Approximate Joins for Data Centric XML

Nikolaus Augsten [1], Michael Böhlen [1], Curtis Dyreson [2], Johann Gamper [1]

[1]*Faculty of Computer Science, Free University of Bozen-Bolzano*
Dominikanerplatz 3, Bozen, Italy
{augsten,boehlen,gamper}@inf.unibz.it

[2]*Department of Computer Science, Utah State University*
Logan, UT 84322-4205, U.S.A.
curtis.dyreson@usu.edu

***Abstract*— In data integration applications, a *join* matches elements that are common to two data sources. Often, however, elements are represented slightly different in each source, so an *approximate* join must be used. For XML data, most approximate join strategies are based on some *ordered* tree matching technique. But in data-centric XML the order is irrelevant: two elements should match even if their subelement order varies.**

In this paper we give a solution for the approximate join of unordered trees. Our solution is based on *windowed* pq*-grams*. We develop an efficient technique to systematically generate windowed pq-grams in a three-step process: sorting the unordered tree, extending the sorted tree with dummy nodes, and computing the windowed pq-grams on the extended tree. The windowed pq-gram distance between two sorted trees approximates the tree edit distance between the respective unordered trees. The approximate join algorithm based on windowed pq-grams is implemented as an equality join on strings and avoids to evaluate the distance between every pair of input trees. Our experiments with synthetic and real world data confirm the analytic results and suggest that our technique is both useful and scalable.

I. Introduction

The amount of data that is stored and exchanged in XML is increasing. When XML data from different sources is integrated in a single data collection, data items that correspond to the same real world object must be matched. But exact matches often fail due to inconsistent representations and missing global keys, so approximate matching techniques must be applied. For instance, when companies merge, their customer data will need to be integrated, but the companies may have different ways to represent customer data. As another example, an internet shop may want to enrich its product description with data provided by third parties, which each have slightly different descriptions for the same product.

One way to approximately match a pair of XML documents is to compute the *minimal edit distance* between the documents [1]–[3]. An XML document can be modeled as an *ordered*, labeled tree. The edit distance between two such trees is the minimum number of node insertions, deletions, and/or renamings that transform one tree into the other. Though order is important in document-centric scenarios (e.g., paragraph tags in XHTML), most data-centric applications ignore the sibling order when considering whether two data items are the same. Data-centric XML items are usually modeled as *unordered*, labeled trees. While the minimal tree edit distance between ordered trees can be computed in polynomial time, the problem has been shown to be NP-complete for unordered trees [4].

This paper develops an efficient approximate join between data-centric XML. Our solution is based on *windowed* pq*-grams,* which are small subtrees of a specific shape. We develop a technique to systematically generate the set of windowed pq-grams in a three-step process: sorting the unordered tree, extending the sorted tree with dummy nodes, and computing the windowed pq-grams on the extended tree. Intuitively, two unordered trees are similar if their sorted trees have many windowed pq-grams in common. The windowed pq-gram distance between sorted trees approximates the tree edit distance between the respective unordered trees. The tree sorting approach is not applicable to other common distances between ordered trees. In particular, it is not possible to approximate the edit distance between unordered trees with the edit distance between sorted trees.

A windowed pq-gram consists of a stem and a base. The stems are invariant to order, and the main challenge is to compute the bases. Our bases satisfy the following core properties: all non-root nodes appear in the same number of bases; the Jaccard distance between two sibling sets is preserved; and node moves to other parents are detected.

We provide an algorithm to compute the windowed pq-gram distance in $O(n \log n)$ time (n is the number of tree nodes) and approximately join unordered trees using windowed pq-grams. Most joins based on distance measures, such as the edit distance, must evaluate the distance between every pair of input trees. There is no effective way to sort sets of trees or partition them into buckets with a hash function. A nested loop join must be applied. Our algorithm reduces the approximate join to an equality join on strings (windowed pq-grams are serialized and represented as strings) that takes advantage of well-known join optimization techniques.

The rest of the paper is organized as follows. Section II presents related work. We motivate the approximate join of data-centric XML in Section III and we discuss the impact of the sibling order on the tree distance computation in Section IV. Windowed pq-grams are introduced in Section V. Section VI discusses core properties of windowed pq-grams, and we tune windowed pq-grams to optimize these properties

in Section VII. Section VIII provides algorithms, which are experimentally evaluated in Section IX. In Section X we draw conclusions and point to future work.

II. Related Work

Most papers that compare similar XML documents represent the XML data as trees. Labels or (label,value)-pairs are assigned to the tree nodes. Tree matching techniques are applied to compute the similarity between trees. A well known distance function for trees is the tree edit distance, which is defined as the minimum number of edit operations (node insertion, node deletion, and renaming) that transforms one tree into another [5]. The best known tree edit distance algorithms [6]–[9] for ordered trees have at least $O(n^3)$ runtime for trees with n nodes. The problem is NP-complete for unordered trees [4].

Guha et al. [2] present an approximate XML join based on the tree edit distance between ordered trees. They give upper and lower bounds for the tree edit distance that can be computed in $O(n^2)$ time and use reference sets to take advantage of the fact that the tree edit distance is a metric, thus reducing the actual number of distances to compute in a join. Guha et al. [2] do not address joins of *unordered* XML.

Garofalakis and Kumar [10] discuss approximate joins in the context of data streaming applications. They focus on performing a match in a limited amount of space and present an efficient approximation of the tree edit distance; but their approximation assumes ordered trees.

pq-Grams were introduced by Augsten et al. [11] as an effective and efficient approximation of the tree edit distance between *ordered* trees. In this paper, we present windowed pq-grams that extend pq-grams to approximate the edit distance between unordered trees, and we develop an efficient join technique based on windowed pq-grams.

In change detection scenarios two versions of the same document are given and the difference is computed. Most research in this area assumes that the trees are ordered [1], [3], [12]. Cobéna et al. [1] take advantage of existing element IDs, which can not be assumed for joins of data from different sources. Chawathe et al. [13] present a heuristic solution for unordered trees that runs in $O(n^3)$ time and for many cases in $O(n^2)$. The X-Diff algorithm by Wang et al. [14] allows leaf and subtree insertion and deletion, and node renaming. To achieve $O(n^2 \times f_{max} \log f_{max})$ runtime (f_{max} is the maximum fanout of the nodes) they match only nodes with the same path to the root node. Our windowed pq-gram distance has $O(n \log n)$ runtime complexity. The distance measures proposed for change detection are evaluated between pairs of documents. Used as a join predicate there is no obvious way to avoid an expensive nested loop join. We transform the distance-based join to an equality join on windowed pq-grams and can apply well-known join optimization techniques.

Weis and Naumann [15] propose an XML similarity measure for a duplicate detection framework. In the worst case, all pairs of elements must be compared. Puhlmann et al. [16] improve the efficiency by applying the Sorted Neighborhood method to nested objects. Both approaches assume a known, common schema of the matched documents and require a configuration step. No join algorithm using the proposed similarity measure is presented.

Sanz et al. [17] develop a similarity-based inverted index to identify regions of XML documents that are similar to a given pattern. Adjacent regions are merged into new regions if the new region better matches the pattern than each of the merged regions. The merging algorithm assumes ordered trees. Joins are not addressed.

A core operation in XML query processing is to find all occurrences of a twig pattern [18], [19], which, in common with approximate join techniques, concerns identifying patterns in a tree. But we split the tree into subtrees in order to calculate the distance between trees, not to answer queries.

Several papers deal with the related, but different problem of detecting the structural similarity between XML documents [20]–[22]. Two documents are considered structurally similar if they are valid for a similar DTD. The text content of the elements and the values of the attributes are ignored.

III. Motivation

In our application scenario we consider building an online database about music CDs that integrates data from two sources: a song lyric store and CD warehouse.[1] The integrated database will store the artists and songs of an album, information about individual songs such as the lyrics, guitar tabs, and information about the artists.

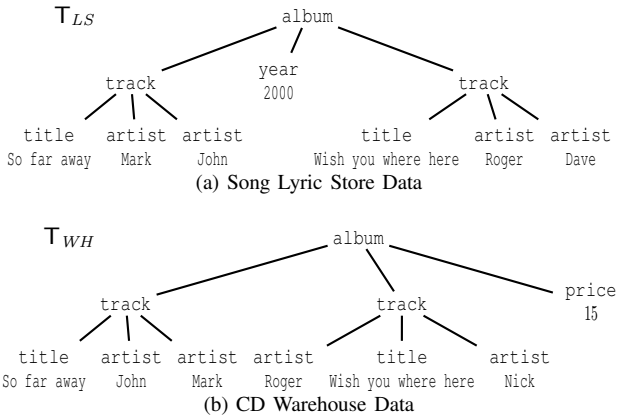
Example 3.1: Figure 1 shows tree representations of two different XML documents. Intuitively, both represent data about the same song album. Yet exact ordered tree matching would not consider the items as the same for a number of reasons. The song lyric store has an element `year` that is absent from the CD warehouse. The CD warehouse has a price for the album. For one track the databases list different artists. Also the document order of elements differs, i.e., the two documents have different sibling orders.

One way to match items from the two sources is to *join* the documents. The join attribute is (the part of) the XML document that represents the album. Two albums match if they are "similar." The join condition can not be equality, as the data items representing the same album in the different databases may not match exactly.

The following XQuery expression returns all `album` pairs that are within distance `$tau`. The distance function, `dist`, is a user-defined function that returns the distance between a pair of XML documents.

```
for $a in doc("lyricstore.xml")//album,
    $b in doc("warehouse.xml")//album
where dist($a,$b) <= $tau
return <match>{$a}{$b}</match>
```

[1] We do not assume that the sources use a common schema, but we assume a common vocabulary to describe the data; the problem of integrating data vocabularies or ontologies is separate from matching the data. Terms in one source can be converted to the vocabulary of the second source prior to matching. We focus on the data matching problem.



(a) Song Lyric Store Data

(b) CD Warehouse Data

Fig. 1. Two XML Trees Representing the Same Album.

In the XQuery expression, `$a` and `$b` are bound to elements of the sets `doc("lyricstore.xml")//album` and `doc("warehouse.xml")//album`, respectively. Each `album` element is a (small) XML document itself. We define the approximate XML join between two sets of XML documents as follows [2].

Definition 3.1 (Approximate XML Join): Given two sets of XML documents, $\mathbf{F}_1$ and $\mathbf{F}_2$, a distance measure, $\mathrm{dist}(\mathsf{T}_i, \mathsf{T}_j)$, between the documents $\mathsf{T}_i \in \mathbf{F}_1$ and $\mathsf{T}_j \in \mathbf{F}_2$, and a threshold τ. The approximate XML join computes all pairs $(\mathsf{T}_i, \mathsf{T}_j) \in \mathbf{F}_1 \times \mathbf{F}_2$, such that $\mathrm{dist}(\mathsf{T}_i, \mathsf{T}_j) \leq \tau$.

Our goal is to find a distance function for *unordered* trees that is effective for data-centric XML and can be computed efficiently. We use this function as the basis of a scalable approximate join.

IV. Ordered vs. Unordered Tree Matching

In this section we introduce basic concepts and discuss the impact of the sibling order on the tree distance computation.

a) XML and Trees: In order to make approximate tree matching applicable for XML, we represent an XML document as a rooted, labeled tree. The tree is unordered in the case of data-centric XML. Each node in the tree is a triple (i, l, v), where i is the node index, l is the node label, and v is the node's value. A node in the tree represents an XML element (or attribute). The node index is any number that identifies the node in the document, such as the ordinal position of the element (or attribute) in document order. The node is labeled with the name of the element (or attribute). The value of a node represents the text content of the corresponding element (or the value of the corresponding attribute). If the corresponding element contains only sub-elements and no content, then the node value is the empty string, ϵ. An edge connects an element node with each of its subelements (or attributes).

The function $\lambda(\mathsf{n})$ maps a node $\mathsf{n}{=}(i, l, v)$ to the pair (l, v) of label and value. While nodes are unique within a tree, the (label,value)-pairs are not. To simplify the discussion we refer to a node by its label and omit node indexes and empty values.

b) Ordered and Unordered Trees: In an *ordered* tree the children of a node form a sequence, in an *unordered* tree the children of a node are not ordered and form a set. Ordered trees that differ only in the sibling order are *permutations* of each other. In a *sorted* tree the siblings are lexicographically ordered by their node labels and values. An unordered tree is transformed to an ordered tree by ordering the siblings. Graphically we represent an unordered tree as a set consisting of a node and the subtrees rooted in the node's children.

Example 4.1: The trees T_0 and T'_0 in Figure 2 are ordered (but not sorted), the trees T_1 and T_2 are unordered. T'_0 is a permutation of T_0. The unordered tree T_1 can be transformed to the ordered tree T_0 (or any of its permutations) by choosing the appropriate sibling order. T_1 differs from T_2 in that the node with label `g` is moved between the trees. There is no sibling order that transforms T_2 to T_0.

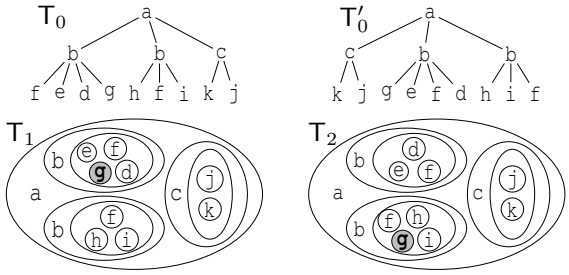


Fig. 2. Ordered and Unordered Trees.

c) Approximating the Edit Distance between Unordered Trees: The edit distance between two trees (ordered or unordered) is the minimum number of node edit operations that transform one tree into the other. The problem is substantially harder for unordered trees (NP-complete [4]) as the distance algorithm can not rely on a sibling order and must consider all sibling permutations. The edit distance between two *unordered* trees is approximated by the smallest edit distance between permutations of the respective *ordered* trees. Transforming an unordered tree to an ordered tree is straightforward. The key issue is to find the permutations of two ordered trees that yield the smallest edit distance. This is non-trivial as illustrated below.

It is obviously not feasible to compute the edit distance between all permutations of two ordered trees. Alternatively, consider an approach that sorts the siblings of both trees by their string labels and values. This heuristic fails for the tree edit distance between ordered trees. When the siblings of a tree are sorted, also the subtrees rooted in the siblings are sorted. But two subtrees that should match may appear in a different order in the two sorted trees. For example, if a subtree root is renamed between two trees, it may result in a different sort position. The edit distance between ordered trees then moves back permuted subtrees node by node. A subtree rooted in a sibling can be of size $O(n)$, where n is the number of tree nodes. Even if the edit distance between two unordered trees is zero or a small constant (for example, a single renamed node), the edit distance between the respective sorted trees may be $O(n)$.

Example 4.2: Consider the two trees T and T' in Figure 3. The children of the root node have the same label such that

the label sort is not unique. Although both trees are sorted, the subtrees t_1 and t_2 are permuted between T and T'. The edit distance between the respective unordered trees is zero as they differ only in the sibling order. The edit distance between the ordered trees is about the tree size as the subtrees t_1 and t_2 must be moved back node by node.

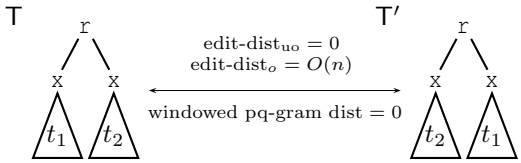


Fig. 3. Edit Distance and Windowed pq-Grams Distance.

By contrast, sorting trees is a valid approach for the windowed pq-grams introduced in the following section. We show that the permutation of a constant number of siblings changes only a constant number of windowed pq-grams. Thus, we can sort the unordered trees and compute the windowed pq-grams on the sorted trees. The resulting distance approximates the edit distance between the *unordered* trees, and the windowed pq-gram distance computed from identical unordered trees is always zero (see Figure 3).

V. Windowed pq-Grams

In this section we introduce windowed pq-grams. We define properties that we require for our solution, and we show that sorting trees is a valid approach for windowed pq-grams. Proofs are omitted due to space constraints.

A. Requirements for Windowed pq-Grams

pq-Grams were introduced by Augsten et al. [11] as an approximation of the edit distance between *ordered*, labeled trees. Intuitively, a pq-gram is a small subtree of a specific shape composed of two parts: a *stem* that consists of an anchor node with $p-1$ ancestors and a *base* that consists of q consecutive children of the anchor node. For example, consider the ordered tree T_0 in Figure 2. The stem $(\mathtt{a}, \mathtt{c})$ with anchor node $\mathtt{c}$ and the base $(\mathtt{k}, \mathtt{j})$ form a pq-gram with $p=q=2$.

Stems are node chains of length p. They are invariant to order, and the strategy for choosing stems in ordered trees carries over to unordered trees. The *bases* in ordered trees are formed by consecutive siblings. This strategy is not applicable to unordered trees, since no sibling order is defined. A different strategy is required.

A *sibling set* is the set of all children of a tree node. In an unordered tree the bases are formed from subsets of sibling sets. A strategy to choose all possible sibling subsets of size q weights nodes differently. There are $\binom{f}{q}$ subsets of size q in a sibling set of f siblings. pq-Grams produced from large sibling sets disproportionally contribute to the total number of pq-grams. Changes covered by these pq-grams are amplified, other changes are disregarded.

Bases that consist of a single node ignore the sibling order. However, pq-grams with such bases fail to detect sibling moves to another parent if the ancestors in the old and the new position have identical labels and values. For example, they cannot distinguish between the trees T_1 and T_2 in Figure 2. The ancestors of the moved node $\mathtt{g}$ have identical labels (and empty values), resulting in identical stems, $(\mathtt{a}, \mathtt{b})$. Ancestors with identical labels and values are frequent in data-centric XML (e.g., all `title` elements have the ancestors `track` and `album` in the XML of Figure 1).

Larger bases encode sibling information and can detect sibling moves, as nodes with homonymous ancestors may have siblings with different labels and/or values. In our example, $\mathtt{g}$ has a sibling $\mathtt{i}$ in T_2 but not in T_1. A base $(\mathtt{g}, \mathtt{i})$ exists only in T_2 and distinguishes it from T_1.

A sibling order may be given implicitly, for example, by the XML document order. This order is random for data-centric XML. Bases formed over randomly ordered sibling sets may be very different even for identical sibling sets.

In our approach we sort the trees and use a window to control the computation of the bases. We seek to build bases with the following properties:

P1: Equal Base-Node Frequency. Each non-root node of the tree appears in the same number of bases, independent of the number of siblings.

P2: Preservation of the Sibling Distance. For bases build from two different sibling sets the percentage of overlap between the bases is equal to the percentage of overlap between the sibling sets. In Figure 2 there is a 50% overlap between the two sibling sets that contain node $\mathtt{g}$, hence also 50% of the bases should match.

P3: Detection of Node Moves to Other Parents. In Fig. 2, node $\mathtt{g}$ is moved to another parent with the same label $(\mathtt{b})$ and value (ϵ). All pq-grams with anchor node $\mathtt{b}$ have the same stem. To distinguish T_1 from T_2 the bases must differ.

B. Solution

We introduce windowed pq-grams that have the required properties for the bases. We proceed in three steps:

a) sort the unordered tree,
b) extend the sorted tree, and
c) compute the windowed pq-grams on the extended tree.

a) Sorting the Unordered Tree: In the first step we sort the trees by imposing a horizontal order among siblings. The siblings are sorted by node label and value. Due to nodes with identical (label,value)-pairs an unordered tree can be sorted in different ways. But all possible sorts of the same unordered tree yield identical pq-grams and so are equivalent for our purpose. Figure 4(a) shows T_1^{sort}, the sorted example tree T_1.

Definition 5.1 (Sorted Tree): A tree T is *sorted* if its siblings are ordered and for each sibling pair, $\mathsf{n} = (i, l, v)$ and $\mathsf{n}' = (i', l', v')$, the order satisfies

$$l < l' \vee (l = l' \wedge v < v') \Rightarrow \mathsf{n} < \mathsf{n}'.$$

b) Extending the Sorted Tree: The next step extends the sorted tree with *dummy nodes* $(\bullet)$. Dummy nodes have a special (label,value)-pair, which is the same for all dummy nodes, $\lambda(\bullet_i) = (\star, \star) = \lambda(\bullet_j)$ for all i, j. The number of

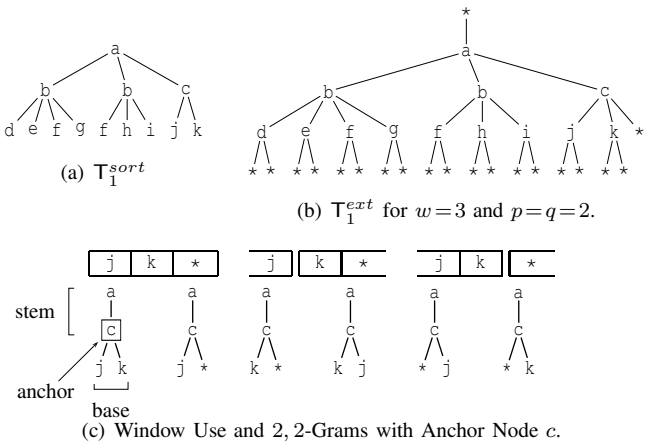


Fig. 4. (a) Sorted Tree, (b) Extended Tree, and (c) Windowed pq-Grams.

dummy nodes depends on the size of the window that we shift over the tree for a systematic generation of the windowed pq-grams. Figure 4(b) shows the extended tree T_1^{ext} for $w=3$ and $p=q=2$.

Definition 5.2 (Extended Tree): Let T^{sort} be a sorted tree, $p > 0$ and $q > 0$ be the parameters determining the shape of the windowed pq-grams, $w \geq q$ be the window size, and f denote the fanout of a node. The *extended tree*, T^{ext}, is defined as T^{sort} extended with dummy nodes as follows:

- *root:* $p-1$ ancestors are prepended to the root node;
- *leaves:* q children are added to each leaf node;
- *siblings:* $w-f$ siblings are appended to each sibling sequence $(\mathsf{c}_1, \ldots, \mathsf{c}_f)$ of size $0<f<w$, yielding $(\mathsf{c}_1, \ldots, \mathsf{c}_f, \bullet_1, \ldots, \bullet_{w-f})$.

c) Computing the pq-Grams: We give a definition of windowed pq-grams based on the extended tree.

Definition 5.3 (Windowed pq-Grams): Let T be an unordered tree with extended tree T^{ext}, n be a node of T, c_i be the i^{th} child of n in T^{ext} $(1 \leq i \leq f^{ext})$, and window $W_i = (\mathsf{c}'_i, \mathsf{c}'_{i+1}, \ldots, \mathsf{c}'_{i+w-1})$, $\mathsf{c}'_k = \mathsf{c}_{(k-1) \bmod f^{ext}+1}$, be a node sequence of length $w \geq q$ that is wrapped around the right border. A *windowed pq-gram* $(p > 0,\ q > 0)$ of T with anchor node n is defined as an ordered subtree of T^{ext} that is composed of a stem and a base. The *stem* is a node chain $(\mathsf{a}_{p-1}, \ldots, \mathsf{a}_1, \mathsf{n})$, where a_k is n's ancestor at distance k. The *base* is a sequence of mutually different siblings $(\mathsf{c}'_i, \mathsf{b}_2, \ldots, \mathsf{b}_q)$ chosen from a window W_i preserving the node order; c'_i is the first node of the window, $\{\mathsf{b}_2, \ldots, \mathsf{b}_q\} \subseteq \{\mathsf{c}'_{i+1}, \ldots, \mathsf{c}'_{i+w-1}\}$. If n is a leaf in T, the base is formed by q dummy nodes. Each base that satisfies these constraints produces a windowed pq-gram with anchor node n. The set of windowed pq-grams of all nodes of T is called the *windowed pq-gram profile* of T.

We use a linear encoding and represent a windowed pq-gram as a tuple $g = (\mathsf{a}_{p-1}, \ldots, \mathsf{a}_1, \mathsf{n}, \mathsf{b}_1, \ldots, \mathsf{b}_q)$. With $\lambda(g) = (\lambda(\mathsf{a}_{p-1}), \ldots, \lambda(\mathsf{n}), \ldots, \lambda(\mathsf{b}_q))$ we denote a pq-gram's node labels and values, called its *label-value tuple*. While a windowed pq-gram is unique within a tree, different windowed pq-grams may yield identical label-value tuples.

The bases are systematically computed by producing for each window W_i only the bases that contain the first window node. For each window position, $\binom{w-1}{q-1}$ bases are produced.

Example 5.1: Figure 4(c) shows all windowed pq-grams for $p = q = 2$ that can be formed in T_1^{ext} for the anchor node with label c. Initially, the window covers the nodes $\boxed{\mathtt{j}, \mathtt{k}, \star}$, which according to the above procedure yields two bases of size 2 and produces the first two windowed pq-grams with label-value tuples $((\mathtt{a}, \epsilon), (\mathtt{c}, \epsilon), (\mathtt{j}, \epsilon), (\mathtt{k}, \epsilon))$ and $((\mathtt{a}, \epsilon), (\mathtt{c}, \epsilon), (\mathtt{j}, \epsilon), (\star, \star))$. Next, the window is moved right and covers $\boxed{\mathtt{k}, \star, \mathtt{j}}$. Notice that the window is wrapped around. Two other windowed pq-grams are produced. The final position of the window covers $\boxed{\star, \mathtt{j}, \mathtt{k}}$.

Theorem 5.1 (Profile Size): If T is an unordered tree with n nodes, then the size of its windowed pq-gram profile, $P(\mathsf{T})$, $q > 1$, is linear in the tree size, $|P(\mathsf{T})| \leq nq\binom{w}{q}$. If T has l leaves, and all other nodes have fanout $f \geq w$, then $|P(\mathsf{T})| = (n-1)\binom{w-1}{q-1} + l$.

Definition 5.4 (Windowed pq-Gram Index): Let T be an unordered tree with windowed pq-gram profile $\mathsf{P}(\mathsf{T})$, $p > 0$, $q > 0$. The *windowed pq-gram index*, I, of tree T is the bag of all label-value tuples of T, i.e.,

$$\mathsf{I}(\mathsf{T}) = \biguplus_{g \in \mathsf{P}(\mathsf{T})} \lambda(g).$$

The windowed pq-gram distance is computed from the number of windowed pq-grams that the indexes of the compared trees have in common. For two unordered trees, T and T', the *windowed pq-gram distance* is

$$\mathrm{dist}(\mathsf{T}, \mathsf{T}') = 1 - 2\frac{|\mathsf{I}(\mathsf{T}) \cap \mathsf{I}(\mathsf{T}')|}{|\mathsf{I}(\mathsf{T}) \uplus \mathsf{I}(\mathsf{T}')|}.$$

The windowed pq-gram distance is 1 if two trees share no windowed pq-grams, and 0 if they have the same windowed pq-gram index, which does not necessarily imply that the trees are equal.

Sorting trees involves subtree permutations. The windowed pq-gram distance is independent of the size of the permuted subtrees. Only the windowed pq-grams that contain the root nodes of the permuted subtrees in the bases change. This core feature qualifies windowed pq-grams for our approach and sets it apart from other distance measures such as the edit distance between ordered trees.

Theorem 5.2 (Local Effect of Subtree Permutations): Given a sorted tree T (index I) that is transformed to a tree T' (index I') by permuting the order of the $f \geq w$ children of a node n, then the permutation affects only $O(f)$ windowed pq-grams:

$$|\mathsf{I} \setminus \mathsf{I}'| \leq O(f).$$

VI. Properties of Windowed pq-Gram Bases

We discuss the base properties of windowed pq-grams. We denote sibling sets with S, the respective bags of (label,value)-pairs with L, and bases formed over L with B.

P1: Equal Base-Node Frequency. Dummy nodes, windows, and window wrapping guarantee that each node of a tree is in the same number of bases, thus giving each node the

same weight. Dummy nodes prevent a node from appearing twice in the same window when the window is wrapped. Due to the window wrapping each node appears in all w positions of a window exactly once, independent of the number of left and right siblings. Only bases within windows are formed, thus each node is in the same number of bases.

P2: Preservation of the Sibling Distance. We analyze the windowed pq-grams of two anchor nodes that have the same stem and differ only in the bases. The bases represent the sibling sets formed by the children of the anchor nodes. The distance between the bases should approximate the distance between the sibling sets.

Sibling Distance. Let L_1 and L_2 be the bags of (label,value)-pairs of two sibling sets. We use the Jaccard distance [23] (modified for bags) between L_1 and L_2 to compute the distance between the respective sibling sets.

$$J(L_1, L_2) = 1 - \frac{2|L_1 \cap L_2|}{|L_1 \uplus L_2|} \quad (L_1 \neq \emptyset \text{ or } L_2 \neq \emptyset)$$

The sibling distance is 1 if all siblings are different, and 0 if L_1 and L_2 have identical (label,value)-pairs.

Base Error. Let B_1 and B_2 be the bases formed over L_1 and L_2, respectively. We define the *base error*,

$$\varepsilon(L_1, L_2, B_1, B_2) = |J(L_1, L_2) - J(B_1, B_2)|, \tag{1}$$

where $J(B_1, B_2)$ is the Jaccard distance between the bases. The base error ε ranges between 0 and 1, $\varepsilon = 0$ means that the base distance is equivalent to the sibling distance.

Example 6.1: Let[2] $L_1 = \{\mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{f}, \mathtt{g}, \mathtt{i}\}$ and $L_2 = \{\mathtt{a}, \underline{\mathtt{b}}, \mathtt{c}, \mathtt{d}, \underline{\mathtt{e}}, \mathtt{f}, \mathtt{g}, \underline{\mathtt{h}}, \mathtt{i}\}$. For $q = 2$ and $w = 3$, we get $B_1 = \{\mathtt{ac}, \mathtt{ad}, \mathtt{cd}, \mathtt{cf}, \mathtt{df}, \mathtt{dg}, \mathtt{fg}, \mathtt{fi}, \mathtt{gi}, \mathtt{ga}, \mathtt{ia}, \mathtt{ic}\}$ and $B_2 = \{\mathtt{a}\underline{\mathtt{b}}, \mathtt{ac}, \underline{\mathtt{b}}\mathtt{c}, \underline{\mathtt{b}}\mathtt{d}, \mathtt{cd}, \mathtt{c}\underline{\mathtt{e}}, \mathtt{d}\underline{\mathtt{e}}, \mathtt{df}, \underline{\mathtt{e}}\mathtt{f}, \underline{\mathtt{e}}\mathtt{g}, \mathtt{fg}, \mathtt{f}\underline{\mathtt{h}}, \mathtt{g}\underline{\mathtt{h}}, \mathtt{gi}, \underline{\mathtt{h}}\mathtt{i}, \underline{\mathtt{h}}\mathtt{a}, \mathtt{ia}, \mathtt{i}\underline{\mathtt{b}}\}$. With $|B_1 \cap B_2|$=6, $|B_1 \uplus B_2|$=30, and $|L_1 \cap L_2|$=6 the base error is $\varepsilon = \frac{2}{5}$. For $q = w = 3$ no bases match, and $\varepsilon = \frac{4}{5}$.

P3: Detection of Node Moves to Other Parents. We define *base recall* and *base precision* to measure the sensitivity of the bases to node moves. A node move is detected if at least one of the bases changes. We consider bases of size $q = 2$ and discuss larger bases in the next section.

A base without dummy nodes encodes exactly one sibling pair. Due to the window wrapping, the same sibling pair may be encoded twice. Two bases formed from the same sibling pair are called *duplicates*, regardless of the node order. Bases with dummy nodes give no sibling information. Let $\#\text{pairs}(S, B)$ denote the number of unique sibling pairs of S encoded by the bases B, i.e., only bases without dummy nodes and only one copy of each duplicate are counted.

Base Recall. For a sibling set S with $f \geq 2$ nodes, $\binom{f}{2} = \frac{f(f-1)}{2}$ pairs can be formed. Given the respective bases B, we define the *base recall*, ρ, as the ratio of sibling pairs encoded by the bases to the number of possible pairs.

$$\rho(S, B) = 2\frac{\#\text{pairs}(S, B)}{f(f-1)}, \quad f = |S| \geq 2 \tag{2}$$

[2]To simplify the notation of this example, we represent a (label,value)-pair by its label and a base by the concatenation of its node labels, e.g., the (label,value)-pair $(\mathtt{a}, \epsilon)$ is denoted as $\mathtt{a}$, the base $((\mathtt{a}, \epsilon), (\mathtt{c}, \epsilon))$ as $\mathtt{ac}$.

$\rho = 1$ if all possible pairs of S are in B, $\rho = 0$ if none of the possible pairs is encoded. Bases with low recall may not encode relevant sibling pairs and thus miss node moves.

Base Precision. Given a sibling set S and the respective set of bases B, the base precision is the ratio of sibling pairs encoded by the bases to the total number of bases:

$$\pi(S, B) = \frac{\#\text{pairs}(S, B)}{|B|}. \tag{3}$$

$\pi = 1$ if the bases contain no duplicates/dummy nodes. In the original tree there are no dummy nodes. A low precision, i.e., many bases with dummy nodes, decreases the weight of the original nodes.

Example 6.2: Let B over siblings S be the bases in Figure 4(c) ($q = 2, w = 3$). $(\mathtt{j}, \mathtt{k})$ and $(\mathtt{k}, \mathtt{j})$ are duplicates, all other bases contain dummy nodes, thus $\#\text{pairs}(S, B) = 1$. Base recall $\rho(S, B) = 1$ (all pairs of S are encoded by B), base precision $\pi(S, B) = \frac{1}{6}$ (only 1 of 6 bases is relevant for detecting node moves).

VII. Optimal Windowed pq-Grams

In this section we discuss the choice of the base size q and the window size w. Specifically, bases of size $q = 2$ have smaller base error than larger bases (Lemma 7.1), but can detect exactly the same sibling moves (Lemma 7.2). For $q = 2$ we provide base recall and precision (Lemma 7.3). We choose a window size w that optimizes both recall and precision, and we show that all nodes in the resulting bases have equal weight (Theorem 7.4).

Lemma 7.1 (Optimal Base Size): Let S and S' be sibling sets with the bags of (label,value)-pairs L and L', respectively, let S be transformed to S' by one of the following edit sequences:

a) k insertions of new nodes with (label,value)-pairs not in L;
b) k renamings of nodes with new (label,value)-pairs not in L ($k \leq |S|$);
c) k node deletions ($k \leq |S|$).

For a given window size $w \leq \min(|S|, |S'|)$, small bases of size 2 ($B_{q=2}, B'_{q=2}$) have equal or smaller base error than larger bases ($B_{q>2}, B'_{q>2}$):

$$\varepsilon(L, L', B_{q=2}, B'_{q=2}) \leq \varepsilon(L, L', B_{q>2}, B'_{q>2})$$

Lemma 7.2 (Sibling Move Detection): Given the sibling sets S_1 and S'_1 with the bases B_1 and B'_1. We move a node n from S_1 to S'_1 and get the sibling sets S_2 and S'_2 with the bases B_2 and B'_2. For a given window size w, if the sibling move is detected for bases with $q > 2$, i.e., $B_1 \cup B'_1 \neq B_2 \cup B'_2$, then it is also detected for bases with $q = 2$.

Lemma 7.3 (Recall and Precision): Let S be a sibling set with $f \geq 2$ nodes, B be the bases of size $q = 2$ formed over S with windows size w. Base recall, $\rho(S, B)$, and base precision, $\pi(S, B)$, are

$$\rho = \begin{cases} 2\frac{w-1}{f-1} & w < \frac{f+1}{2} \\ 1 & w \geq \frac{f+1}{2} \end{cases} \quad \pi = \begin{cases} 1 & w < \frac{f+1}{2} \\ \frac{f-1}{2(w-1)} & w \geq \frac{f+1}{2} \end{cases}$$

Theorem 7.4: (Optimal Windowed pq-Grams) Given an unordered tree with fixed fanout $f \geq 2$ for the non-leaf nodes. For base size $q = 2$ and window size $w = \frac{f+1}{2}$ we get windowed pq-grams with the following properties:

(a) Each non-root node appears in exactly $2w - 2$ bases.

(b) $\varepsilon \leq \begin{cases} \frac{k}{f} & \text{for rename} \\ \frac{2k}{2f+k} & \text{for insert} \\ \frac{2k}{2f-k} & \text{for delete} \end{cases}$

(c) $\rho = 1 \qquad \text{for } w = \lceil \frac{f+1}{2} \rceil$

(d) $\pi = 1 \qquad \text{for } w = \lfloor \frac{f+1}{2} \rfloor$

The optimal base size w depends on the fanout f. For a degenerated tree (consisting only of the root node and $n-1$ leaves) $w = \frac{f-1}{2} = O(n)$. Even in this case, the windowed pq-gram profile can not grow larger than $O(n^2)$ (Theorem 5.1, $f \geq w$, $q = 2$).

VIII. Algorithms

A. Building the pq-Gram Index

Algorithm 1 computes the windowed pq-gram profile P for $q = 2$ by recursively traversing the tree T in preorder. The algorithm is initialized with the root node n of T, the window size w, a stem of dummy nodes $(\bullet_1, \ldots, \bullet_p)$, and the empty profile $\mathsf{P} = \emptyset$. Whenever the last sibling (in document order) of a sibling set is reached, the siblings are sorted (dummy nodes to the end), and the windowed pq-grams are produced. The runtime is $O(n + f_{max} \log f_{max})$ for documents with n nodes, a maximal fanout of f_{max}, and constant window size. Our experiments confirm the analytic runtime result.

The index, I, is computed by aggregating and counting the label-value tuples of the windowed pq-grams in the profile $\mathsf{P}(treeId, pqg)$: $\mathsf{I} \leftarrow \Gamma_{treeId, \lambda(pqg) \rightarrow pqg, \mathrm{COUNT}(*) \rightarrow cnt}(\mathsf{P})$. The runtime is $O(n \log n)$ (sorting the profile of size $O(n)$). The index of a forest is the union of the indexes of its trees.

To deal with node labels and values of different length, such as element names and text values in XML documents, we use a fingerprint hash function (e.g., the Karp-Rabin fingerprint function [24]) that maps a string s to a hash value $h(s)$ of fixed length that is unique with a high probability. Instead of storing the label-value tuples of windowed pq-grams, we store the concatenation of the hashed labels and values. Note that the only operation we need to perform on the (label,value)-pairs is to check equality.

Example 8.1: Figure 5 shows an example hash function and part of the windowed pq-gram indexes of the two XML documents in Figure 1, the music albums from the song lyric store (T_{LS}) and the CD warehouse (T_{WH}). We choose $p = q = 2$, $w = 3$, $\lambda(\bullet) = (\star, \star)$. The label-value tuple $((\star, \star), (\texttt{album}, \epsilon), (\texttt{track}, \epsilon), (\texttt{track}, \epsilon))$ with hash value `9999 4100 3200 3200` appears twice in the index of T_{LS} and has two matches in the other index. The label-value tuple $((\texttt{album}, \epsilon), (\texttt{year}, 2000), (\star, \star), (\star, \star))$ with the hash value `4100 5497 9999 9999` appears only once in the index of T_{LS} and has no match in the index of T_{WH}.

Algorithm 1: `getPQGrams`$(\mathsf{T}, \mathsf{n}, w, \mathrm{stem}, \mathsf{P})$

```
stem ← dequeue-first-element(stem) ∘ n;
if n is a leaf then return P ∪ {(T, stem∘(•,•))};
C ← ∅;
foreach child c of n do
    C ← C ∪ {c};
    P ← P ∪ getPQGrams(T, c, w, stem, P);
end
C ← C ∪ ⋃_{i=1}^{w-f} {•};
a ← sort-by-label-value(C);
for i ← 0 to |a| − 1 do
    for j ← i + 1 to i + w − 1 do
        P ← P ∪ {(T, stem ∘ a[i] ∘ a[j mod |a|])};
    end
end
return P;
```

B. Approximate XML Join

Algorithm 2 computes the approximate join of two sets of unordered trees, $\mathbf{F}_1$ and $\mathbf{F}_2$, given their windowed pq-gram indexes, I_1 and I_2, and the threshold, τ. All pairs $(\mathsf{T}_i, \mathsf{T}_j) \in \mathbf{F}_1 \times \mathbf{F}_2$ that satisfy $\mathrm{dist}(\mathsf{T}_i, \mathsf{T}_j) \leq \tau < 1$ are returned. PS_i is initialized with the profile sizes for the trees in forest $\mathbf{F}_i$.

Algorithm 2: `pqGramJoin`$(\mathsf{I}_1, \mathsf{I}_2, \tau)$

```
foreach I_i do
    I_i ← ρ_{treeId/treeId_i, cnt/cnt_i}(I_i);
    PS_i ← Γ_{treeId_i, SUM(cnt_i)→size_i}(I_i);
end
return π_{treeId_1, treeId_2}(σ_{1−2 cnt/(size_1+size_2) ≤ τ}(
        Γ_{treeId_1, treeId_2, SUM(min(cnt_1, cnt_2))→cnt}(I_1 ⋈ I_2)
    ⋈ PS_1 ⋈ PS_2))
```

As pointed out by Guha et al. [2], hash and sort-merge joins do not carry over to approximate tree joins that use the edit distance, since the distance function must be evaluated between every input pair. There is no effective way to sort trees or partition them into buckets with a hash function. The only approach readily applicable is the nested loop join [2].

This does not hold for the windowed pq-gram distance. For the calculation of the windowed pq-gram distance a tree is represented by its windowed pq-gram index. Instead of computing the distance between each pair of trees directly, we check for each windowed pq-gram in which pairs of trees it appears. We transform the distance-based join to an equality join on all windowed pq-grams represented as strings. We can apply well known techniques to optimize this join (e.g., sort-merge and hash join). The approximate join is computed by counting windowed pq-grams in the join result.

s	$h(s)$
*	99
ϵ	00
album	41
track	32
title	02
artist	11
year	54
price	19

s	$h(s)$
So far away	67
Mark	86
John	15
2000	97
15	73
Wish you where here	42
Roger	26
Dave	09
Nick	37

(a) Hash Function.

$treeId$	pqg	cnt
...	...	...
T_{LS}	9999 4100 3200 5497	2
T_{LS}	9999 4100 3200 3200	2
T_{LS}	9999 4100 5497 3200	2
T_{LS}	4100 5497 9999 9999	1
T_{LS}	4100 3200 0267 1186	1
T_{LS}	4100 3200 0267 1115	1
T_{LS}	4100 3200 1186 1115	1
...	...	...

(b) pq-Gram Index of the Song Lyric Store.

$treeId$	pqg	cnt
...	...	...
T_{WH}	9999 4100 3200 3200	2
T_{WH}	9999 4100 3200 1973	2
T_{WH}	9999 4100 1973 3200	2
T_{WH}	4100 1973 9999 9999	1
T_{WH}	4100 3200 0267 1115	1
T_{WH}	4100 3200 0267 1186	1
T_{WH}	4100 3200 1115 1186	1
...	...	...

(c) pq-Gram Index of the CD Warehouse.

Fig. 5. Implementation of the Windowed pq-Gram Index.

In the worst case the joined forests consist of identical copies of the same tree. Let N be the cardinality of the forest, n the number of nodes per tree. The indexes are of size $O(Nn)$ for a constant window size. In a sort-merge join the complexity of sorting the relations is $O(Nn\log(Nn))$. Each windowed pq-gram in one index matches $O(N)$ tuples in the other index. The overall complexity is $O(Nn(N+\log n))$. Note that for this worst case scenario the join result is of size $O(N^2)$, thus no algorithm can improve on the quadratic runtime.

Different from the nested loop join, our join algorithm can take advantage of the diversity of trees in a forest. In the best case, when no two trees in the forest share windowed pq-grams, the runtime is $O(Nn\log(Nn))$ for the index size $O(Nn)$. In our experiments we show the performance advantages of the optimized join for large forests.

IX. Experiments

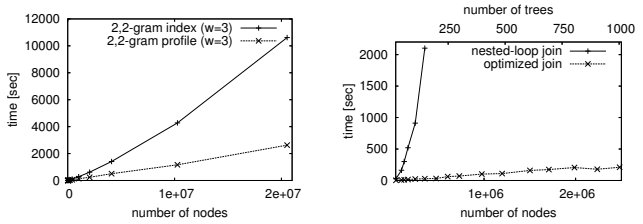
A. Profile and Index Computation.

We analyze the scalability of the windowed pq-gram index computation. Our test data are XML documents that range between 100kB and 1.2GB (2k to 20M nodes), $p = q = 2$ and $w = 3$. The index computation in Figure 6(a) includes the profile computation (Algorithm 1) and the aggregation of duplicate pq-grams within each tree. The index computation scales to very large trees. The test documents are generated with `xmlgen`, provided by the XML benchmark project XMark[3].

[3]`http://monetdb.cwi.nl/xml/`

B. Approximate Join Based on Windowed pq-Grams.

We compare the scalability of the "optimized join" (Algorithm 2) with the scalability of a join that computes the windowed pq-gram distance between each pair of documents ("nested loop join"). We join two sets of synthetic XML documents. Each set consists of 1000 documents with 100 to 17000 nodes and stores 58MB of data. The documents within a set are different, each document has a match in the other set. Figure 6(b) shows the results. The optimized join computes only the distance between documents that have pq-grams in common. Unlike the nested loop join, it can take advantage of the diversity of the trees that result in a small join results set. The runtime is close to linear.



(a) Windowed pq-Gram Index Computation. (b) Approximate Join Based on Windowed pq-Grams.

Fig. 6. Index Creation and Join Scalability.

C. Quality of Matches.

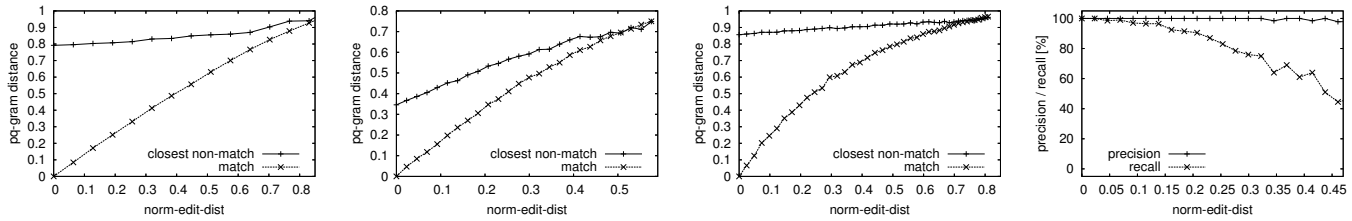
We use real world XML data sets and add noise (spelling mistakes and missing elements). We approximately join the original and the noisy set.

The Data Sets. We use the DBLP[4] (bibliography), the SwissProt[5] (protein sequence database), and the Treebank[6] (parts of speech tagged English sentences) XML databases. We split each database into a set of (sub)documents by deleting the root node, and we randomly choose 200 of the resulting documents for our experiments (requiring their size to be larger than the number of errors we introduce).

The resulting document sets are structurally very different: DBLP contains small and flat documents (15 nodes and depth 1.9 on average) with about ten times more elements than attributes, the SwissProt documents are larger and deeper with almost the same number of attributes and elements (104 nodes and depth 3.5 on average), the Treebank documents have deep recursive structure (49 nodes and depth 6.9 on average, with a maximum depth of 30).

Adding Noise. We modify the *original documents* by deleting and renaming random nodes. Node deletions simulate missing elements or attributes and modify the document structure. Renamed nodes represent different tag names or spelling mistakes in the text values. The resulting noisy document is the *match* of the original document, all other noisy documents are *non-matches*. In our figures we show the percentage of changed nodes (*norm-edit-dist*).

[4]`http://dblp.uni-trier.de`
[5]`http://us.expasy.org/sprot/`
[6]`http://www.cis.upenn.edu/~treebank/`



(a) DBLP (b) SwissProt (c) Treebank (d) 1:1 Matches for SwissProt

Fig. 7. (a-c) Distance between Matches and Non-Matches. (d) 1:1 Matches for SwissProt.

Distance between Matches and Non-Matches. Each original document has exactly one match. Figures 7(a)–7(c) show the average distance of the original documents to their match and to the closest non-match. The SwissProt documents are more similar to each other than the DBLP and Treebank documents. The windowed pq-gram distance to the matches is almost linear to the number of modified nodes. It effectively approximates the edit distance. All documents are modified, thus also the distance to the non-matches increases with the number of changed nodes.

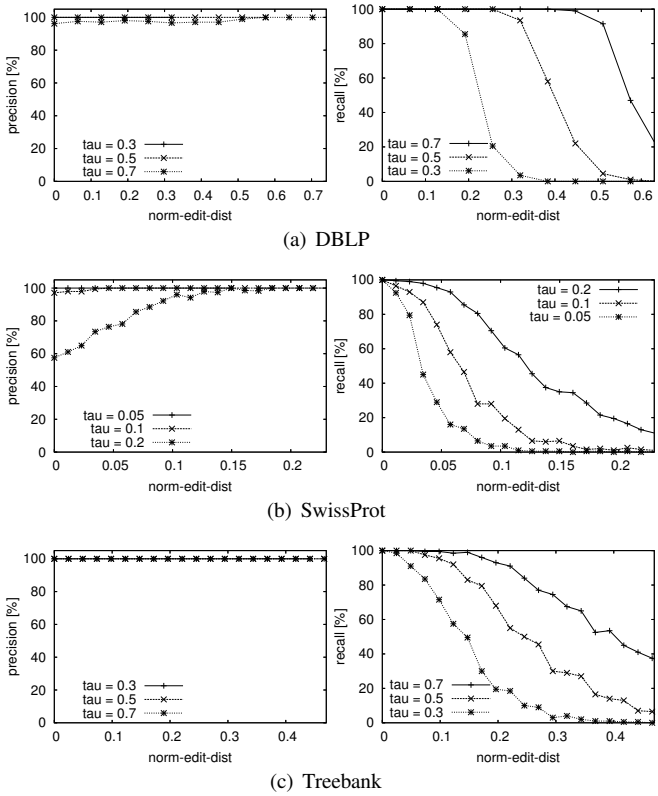
Precision and Recall. Our join algorithm matches each original document to one or more noisy documents. We count *correct* and *incorrect* matches. With $possible$ we denote the maximum number of correct matches that exist for a dataset. In our setting, $possible$ is equal to the number of documents in the dataset. We compute $precision = \frac{correct}{correct+incorrect} \times 100\%$ and $recall = \frac{correct}{possible} \times 100\%$. The precision is high if the returned matches are correct, the recall is high if the algorithm does not miss correct matches.

Figure 8 shows precision and recall for different thresholds τ. Moving up the threshold decreases the precision and increases the recall. Precision and recall for DBLP and Treebank are almost 100%, even for very noisy documents.

For SwissProt the precision drops as we increase the threshold. The SwissProt documents are clustered into groups of very similar documents (protein variants). For example, two documents with 64 elements have exactly the same structure and vary only in 6 text values. The clustering of the data is evident from the precision values in Figure 8(b) for *norm-edit-dist* $= 0$ (approximate self join): Already for $\tau = 0.2$ many documents match other documents than themselves. We improve the result for SwissProt using a variable threshold. Each document is matched to its nearest neighbor. If a document has more than one nearest neighbor, no match is returned. Figure 7(d) shows the results for the SwissProt database. The algorithm returns precise matches, and even for errors of 20% we miss only about 10% of the matches.

X. Conclusion

When XML data from different sources is integrated in a single data collection, data items that represent the same real world object must be recognized. Exact matches, however, often fail in such applications (elements may be missing in one database, content values may not match due to different coding conventions and spelling mistakes, and the data may



(a) DBLP

(b) SwissProt

(c) Treebank

Fig. 8. Matching with Different Thresholds.

be arranged in a different structure). Approximate matching techniques must be applied.

Previous research developed approximate join operations based on ordered tree-matching, but for data-centric XML applications the order of siblings should not matter. Data-centric XML can be represented as unordered trees. In this paper we propose an approximate join technique for data-centric XML based on windowed pq-grams.

pq-Grams were developed for the approximate matching of ordered trees [11]. We introduce *windowed* pq-grams to approximately match unordered trees in a three-step process: sorting the tree, extending the sorted tree with dummy nodes, and computing the windowed pq-grams on the extended tree. Windowed pq-grams consist of a stem and a base. The stems are invariant to order, and the main challenge is to compute the bases. Our bases enjoy the following important properties:

all non-root nodes appear in the same number of bases; the Jaccard distance between sibling sets is preserved, and node moves to other parents are detected. The windowed pq-gram distance between sorted trees approximates the tree edit distance between the respective unordered trees.

We show that the permutation of a constant number of siblings changes only a constant number of windowed pq-grams. This core feature makes windowed pq-grams eligible for our tree sorting approach and rules out other common distances such as the edit distance between ordered trees.

We provide an efficient algorithm for the approximate join of unordered trees, which is implemented as an equality join on windowed pq-grams and can take advantage of well known join optimization techniques. To the best of our knowledge, this is the first work to address the problem of approximately joining *data-centric* XML, where the distance algorithm can not take advantage of a predefined document order. Extensive experiments on both synthetic and real world data confirm the analytic results and suggest that our technique is both useful and scalable.

Future work includes the investigation of persistent, updatable index structures for the windowed pq-gram join. As windowed pq-grams store local information, a document modification (e.g., an altered text value) affects only a limited number of windowed pq-grams. The index should be updated incrementally by substituting the affected pq-grams only, thus avoiding the recomputation of all windowed pq-grams from scratch.

Further we plan to combine our approximate join on data-centric XML with approximate string matching techniques. The string values of some elements or attributes may be particularly important to identify a data item, for example, the title of an article is very significant in a XML database about publications. We would like to include both the similarity of the XML structure and the similarity of selected string values into our approximate join.

ACKNOWLEDGEMENTS

The work has been done in the framework of the project *eBZ – Digital City*, which is funded by the Municipality of Bolzano-Bozen. We wish to thank our colleagues at the municipality, in particular Franco Barducci, Walter Costanzi, Roberto Loperfido, and Danila Sartori.

REFERENCES

[1] G. Cobéna, S. Abiteboul, and A. Marian, "Detecting changes in XML documents," in *Proceedings of the International Conference on Data Engineering (ICDE)*. San Jose, California: IEEE Computer Science Press, 2002, pp. 41–52.

[2] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu, "Approximate XML joins," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Madison, Wisconsin: ACM Press, 2002, pp. 287–298.

[3] K.-H. Lee, Y.-C. Choy, and S.-B. Cho, "An efficient algorithm to compute differences between structured documents," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 16, no. 8, pp. 965–979, Aug. 2004.

[4] K. Zhang, R. Statman, and D. Shasha, "On the editing distance between unordered labeled trees," *Information Processing Letters*, vol. 42, no. 3, pp. 133–139, 1992.

[5] K.-C. Tai, "The tree-to-tree correction problem," *Journal of the ACM (JACM)*, vol. 26, no. 3, pp. 422–433, July 1979.

[6] W. Chen, "New algorithm for ordered tree-to-tree correction problem," *Journal of Algorithms*, vol. 40, no. 2, pp. 135–158, Aug. 2001.

[7] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann, "An optimal decomposition algorithm for tree edit distance," in *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP 2007)*, Wroclaw, Poland, 2007.

[8] P. N. Klein, "Computing the edit-distance between unrooted ordered trees," in *Proceedings of the 6th European Symposium on Algorithms*, ser. Lecture Notes in Computer Science, vol. 1461. Venice, Italy: Springer, 1998, pp. 91–102.

[9] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[10] M. Garofalakis and A. Kumar, "XML stream processing using tree-edit distance embeddings," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 279–332, 2005.

[11] N. Augsten, M. Böhlen, and J. Gamper, "Approximate matching of hierarchical data using pq-grams," in *Proceedings of the International Conference on Very Large Databases (VLDB)*. Trondheim, Norway: Morgan Kaufmann Publishers Inc., Sept. 2005, pp. 301–312.

[12] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Montreal, Canada: ACM Press, June 1996, pp. 493–504.

[13] S. S. Chawathe and H. Garcia-Molina, "Meaningful change detection in structured data," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Tucson, Arizona, United States: ACM Press, May 1997, pp. 26–37.

[14] Y. Wang, D. J. DeWitt, and J. Cai, "X-Diff: An effective change detection algorithm for XML documents." in *Proceedings of the International Conference on Data Engineering (ICDE)*. Bangalore, India: IEEE Computer Science Press, Mar. 2003, pp. 519–530.

[15] M. Weis and F. Naumann, "DogmatiX tracks down duplicates in XML," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Baltimore, Maryland, USA: ACM Press, June 2005, pp. 431–442.

[16] S. Puhlmann, M. Weis, and F. Naumann, "XML duplicate detection using sorted neighborhoods," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, ser. Lecture Notes in Computer Science, vol. 3896. Munich, Germany: Springer, Mar. 2006.

[17] I. Sanz, M. Mesiti, G. Guerrini, and R. Berlanga, "Fragment-based approximate retrieval in highly heterogeneous XML collections," *Data & Knowledge Engineering*, vol. 64, no. 1, pp. 266–293, Jan. 2008.

[18] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: Optimal XML pattern matching," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Madison, Wisconsin: ACM Press, June 2002, pp. 310–321.

[19] H. Jiang, W. Wang, H. Lu, and J. X. Yu, "Holistic twig joins on indexed XML documents," in *Proceedings of the International Conference on Very Large Databases (VLDB)*. Berlin, Germany: Morgan Kaufmann Publishers Inc., Sept. 2003, pp. 273–284.

[20] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. Sellis, "A methodology for clustering XML documents by structure," *Information Systems*, vol. 31, no. 3, pp. 187–228, May 2006.

[21] S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese, "Fast detection of XML structural similarity," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 17, no. 2, pp. 160–175, Feb. 2005.

[22] A. Nierman and H. V. Jagadish, "Evaluating structural similarity in XML documents," in *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, USA, June 2002.

[23] C. J. van Rijsbergen, *Information Retrieval*, 2nd ed. Butterworth-Heinemann, Mar. 1979, ch. 3.

[24] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, Mar. 1987.