Hashing Tree-Structured Data: Methods and Applications

Shirish Tatikonda and Srinivasan Parthasarathy

Department of Computer Science and Engineering, The Ohio State University
2015 Neil Ave, Columbus, OH 43202, USA
(tatikond,srini)@cse.ohio-state.edu

***Abstract*— In this article we propose a new hashing framework for tree-structured data. Our method maps an unordered tree into a multiset of simple wedge-shaped structures refered to as *pivots*. By coupling our pivot multisets with the idea of minwise hashing, we realize a fixed sized signature-sketch of the tree-structured datum yielding an effective mechanism for hashing such data. We discuss several potential pivot structures and study some of the theoretical properties of such structures, and discuss their implications to tree edit distance and properties related to perfect hashing. We then empirically demonstrate the efficacy and efficiency of the overall approach on a range of real-world datasets and applications.**

I. Introduction

Advances in data collection and storage technology have led to a proliferation of information available to organizations and individuals. This information is often also available to the user in a myriad of formats and multiple media. With the increasing importance given to semantic web [1], [2] and Web 2.0 technologies, languages such as XML and collaborative community systems like DBLife [3], an increasing number of these data stores are housed in (semi-)structured formats. Examples abound ranging from XML data repositories [4] to directory information on modern file systems, from MPEG-7 repositories [5] to linguistic data [6] and from social network data [7] to phylogenetic data [8]. With the increasing use of such structured datasets for housing information, the need for *efficiently* managing, querying and analyzing such data is thus growing.

A fundamental operation, in a number of domains, for example in database systems, data mining, computational geometry and network science, is that of hashing. A hash function is a procedure that maps a large, possibly variable sized piece of information into a small fixed sized datum. Hash functions are ubiquitous in their use and are primarily used as a tool to improve the efficiency of search, for example in the finding items in a database [9], detecting duplicate records [10], [11], and localizing related data for subsequent analysis [12] etc. Although the idea of hashing dates more than 50 years ago, much of the work to date has primarily focused on the hashing of (variable-sized) sets [13], [14], documents [15], sequences [16] and geometric objects [17]. Hashing and sketching tree- and graph- structured data is not so well understood although it has been the focus of recent research [18], [19], [20], [21], [22].

In this article we focus primarily on the problem of hashing and sketching tree-structured data. For expository simplicity we focus on rooted trees although many of our ideas apply for free trees and directed acyclic graphs as well (not discussed further). Our approach relies on two operators – a *transformation* operator that converts a tree-structured dataset into a (multi-)set of pivotal elements, and a *signature-sketch* operator that converts the (multi-)set into a fixed datum that can subsequently employ any standard hash mechanism. We investigate the theoretical properties (e.g. efficiency, perfect hashing, edit-distance bounds) of several possible transformation operators and describe efficient mechanisms to compute them. For the signature-sketch operator we primarily focus on the use of min-wise hashing [15], [13], [14] although alternate strategies (e.g. locality sensitive hashing [23], [24], [25], bloom hashing [26]) may also be options to consider in the future.

In addition to the theoretical analysis we present a comprehensive empirical study to compare and contrast the proposed strategies on multiple synthetic and real datasets. We have evaluated our methods along the axes of efficiency, storage costs and performance from the perspective of the application domain. The benefits of the proposed approach are showcased on different application domains including, XML deduplication, stratified sampling of structure data for mining frequent trees on web log data and phylogenetic data analysis.

To reiterate the key contributions of our study include:

- Novel incremental, transformation operators based on the notion of induced, embedded and constrained and embedded pivot structures that map a given tree-structured datum into a multi-set.
- Signature-sketch operators based on min-wise hashing to convert the resulting multi-set from the transformation operator into a fixed sized datum which can subsequently be leveraged in estimating the similarity between different trees.
- Theoretical results showing that one of the transformation operators we propose can form the basis for a perfect hash function and lower bounds relating the proposed operators to traditional tree edit distance measures.
- Empirical results on a range of datasets and application scenarios demonstrate efficacy and efficiency of the proposed methods.

II. Sketching a Tree Structure

Our approach for constructing signatures for a given tree relies on two main functions viz. a *transformation function* and a *signature function*. While the former is responsible for transforming the tree into meaningful substructures, the latter is accountable for constructing shorthand sketch or fingerprint for the given tree. The transformation function can be adjusted to reflect different notions of tree structure depending on the application requirements. The signature function acts upon the image of the transformation function to construct small sized representation to enable efficient application development. One can then optionally employ any *locality sensitive hashing* technique [23], [24], [25] to perform a variety of operations including nearest neighbor search and grouping similar structured trees. We now present our transformation and signature functions, and illustrate their use in the context of computing the similarity between two given trees. We describe our entire tree sketching method with respect to a particular transformation function (see Sections II-A & II-B). We later show alternative choices for transformation functions in Section II-C.

A. Transformation Function (tf)

A tree structure in terms of set theory can be thought of as a partially ordered set (poset) of elements that are ordered by a relation namely *parent-child*. It induces all other associations among nodes such as ancestor-descendants, siblings etc. Not only that the tree structure imposes different relations among nodes but in converse these relations also precisely define a tree structure. Therefore, one possible way to compare two different tree structures is to examine the relationships that given two trees preserve – the fundamental motivation behind our transformation function. This function maps a given tree into a multiset of substructures known as *pivots* where each pivot individually captures specific relationship present among the nodes involved in the substructure. For the sake of explanation, we define a particular type of pivot structure here, which is called as *embedded pivot* structure. We later show how pivot structures can be adapted to capture different notions of tree similarity based an application demands.

Definition 2.1: **Embedded Pivot**: An embedded pivot substructure containing two nodes $u, v \in T$ is defined as the tuple (lca, u, v) where lca is the lowest common ancestor of u and v.

An embedded pivot evidently encapsulates the association between u and v by denoting the ancestor-descendant relations $lca - u$ and $lca - v$. Since lca may not be the direct parent of u and v, the pivot structure is in fact an embedded subtree of T. The set of all pivots involving a particular node $w \in T$, denoted $\mathcal{S}^T(w)$, describes the tree structure when seen from the perspective of w. This set can further be divided into the ones in which w is the root and the ones in which w is one of the two child nodes of a pivot pattern. We thus have:

$$\mathcal{S}^T(w) = \mathcal{S}^T_{root}(w) + \mathcal{S}^T_{child}(w),\ where$$

$$\mathcal{S}^T_{root}(w) = \{(lca, u, v) | lca = w\}$$

$$\mathcal{S}^T_{child}(w) = \{(lca, u, v) | u = w \vee v = w\}$$

Also, the set of all pivots associated with a tree T is given by $\mathcal{S}(T) = \bigcup_{w \in T} \mathcal{S}^T(w)$

In case of labeled trees, we store the node labels in each pivot. A pivot would then be described as $(l(lca), l(u), l(v))$ rather than (lca, u, v). Since multiple nodes can have the same label, there can be repetitions in $\mathcal{S}(T)$ – making it a pivot *multiset* (or a *bag*) instead of a pivot *set*. Throughput this article, we use the words *multiset* and *set* interchangeably. We also ignore the label function $l(\cdot)$ whenever possible for notational convenience. We use the hyphen ('$-$') as a wildcard symbol in the pivot. For example, $(w, -, -)$ refers to the set $\mathcal{S}^T_{root}(w)$. We further abuse the use of $-$ to omit certain fields in the pivot that are not important for the discussion.

A simple method to construct an embedded pivot multiset is shown as Algorithm 1. It operates on a particular orientation (a particular arrangement of an unordered tree) of the given unordered tree. Every iteration of the outer loop in Line 2 produces all pivots in which u is one of the child node. Also, u is paired up all the other nodes $v \in T$ where v is neither an ancestor nor a descendant of u. Since T is unordered, the resulting pivots are also unordered. In order to ensure different orientations of the same pivot structure are treated in the same way, we swap the node labels in Lines $5-6$. Such an artificial order guarantees that all orientations of an unordered tree produce the same (multi)set of pivots. Since the output size is potentially quadratic in tree size, the computation complexity of Algorithm 1 is $O(n^2)$, where n is the number of nodes in T. We would like to note that it is fairly easy to parallelize Algorithm 1 (see Section IV). Such parallel strategies are of great importance in the context of modern day data servers, which are typically mulicore systems.

Algorithm 1 Transformation function that constructs embedded pivots

```
1: n ← |T|
2: for each u in T in pre-order do
3:     for each v in T[rml(v) + 1 ··· n] do
4:         lca ← lowest common ancestor of (u,v)
5:         if lca != v then
6:             if l(u) > l(v) then
7:                 swap(l(u), l(v))
8:         add pivot ( l(lca), l(u), l(v) ) to S(T)
```

From the above algorithm, we can easily construct a formula to compute the exact number of pivots $\mathcal{N}(T)$ as a function of tree size n. In the following derivation, $pre(v)$ and $|D(v)|$ refers to the pre-order number and the number of descendants

of v, respectively.

$$\mathcal{N}(T) = \sum_v (\ rml(v) + 1 \cdots n\ ) = \sum_v [\ n - pre(v) - |\mathcal{D}(v)|\ ]$$
$$= n^2 - \tfrac{n\cdot(n+1)}{2} - \sum_v |\mathcal{D}(v)| = \tfrac{n\cdot(n-1)}{2} - \sum_v |\mathcal{D}(v)| \quad (1)$$

The total number $\mathcal{N}(T)$ reaches its maximum when the sum of all descendants $\sum_v |\mathcal{D}(v)|$ is minimum, as in the case of bushy tree shown in Figure 1a. Similarly, the size of pivot set has its minimum for *chain* trees where the tree is a single path containing $n-1$ edges (see Figure 1b). Therefore,

$$min(\textstyle\sum_v |\mathcal{D}(v)|) = n-1 \quad \Rightarrow \quad max(\mathcal{N}(T)) = \tfrac{(n-1)\cdot(n-2)}{2}$$

$$max(\textstyle\sum_v |\mathcal{D}(v)|) = \tfrac{n\cdot(n-1)}{2} \quad \Rightarrow \quad min(\mathcal{N}(T)) = 0$$

In order to avoid the special case of chain trees with zero number of pivots, we assume, without loss of generality, in rest of this article that all trees are *branched* i.e., the root node has at least two children. Note that, a chain tree can be made as a branched tree by introducing a *dummy* child node. For branched trees with n nodes, the minimum number of pivots will then be equal to $n-2$, as shown in Figure 1c.

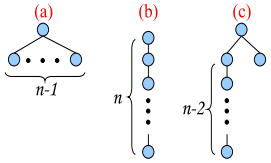

Fig. 1. The cases for minimum and maximum $\mathcal{N}(T)$

We make use of these pivot sets for establishing the similarity between given two trees. This is possible because the tree similarity can be inferred by examining the structural relationships among tree nodes, which are precisely captured in our pivot substructures. We would like to note that simple intuitive transformations are not useful here. Consider a mapping f that maps a tree into a set of node labels or edges i.e., $f : \mathbb{T} \rightarrow \mathbb{E}$ where $f(T) = E_T$ is the set of all edges in T. f is clearly a *surjective* function because many different trees are likely to be mapped to the same edge set i.e., f causes many "collisions". Such functions are not helpful due to the simplistic nature of their substructures. In contrast, our transformation functions capture interesting structural relationships present among tree nodes. Furthermore, they can be made, when specialized with more information, to provide unique mapping from trees to multisets (see Section III). Such unique transformations resemble *perfect hash* functions which are guaranteed to map distinct elements to distinct keys. In cases where the application does not require such powerful unique transformations, one may settle for simple but no simpler pivot substructures. We present some of these alternative pivot structures in Section II-C. Essentially, the exact nature of the transformation primarily depends on the underlying application.

Due to the nature of our pivot substructures, *all the properties given on top of sets can now be provided for trees also*. In particular, the *Jaccard coefficient* that serves as a metric for (pivot) set similarity can now be used as a measure of similarity between the corresponding trees.

$$sim(T_1, T_2) = Jaccard(\mathcal{S}(T_1), \mathcal{S}(T_1)) = \frac{|\mathcal{S}(T_1) \cap \mathcal{S}(T_2)|}{|\mathcal{S}(T_1) \cup \mathcal{S}(T_2)|} \quad (2)$$

The intersection (union, respectively) is generalized from sets to multisets by taking the minimum (maximum, resp.) of the two frequencies of a pivot in the multisets to be intersected (merged, resp.).

B. Signature Function ($\mathbf{sf}$)

Recall that the embedded pivot sets are potentially quadratic in tree size. Merging and intersecting them as in Equation 2 can be computationally expensive. When the sets are so long, all pivot sets may not fit in main memory. We improve the efficiency by making use of a *signature function* that converts these large pivot sets into shorthand *summaries* or *fingerprints*. Subsequent merge and intersection operations are performed on these small signatures. The key idea is to produce signatures which are small enough to fit in main memory and more importantly the similarity between two signatures is roughly the same as the similarity between corresponding pivot sets. There may exist multiple different ways to define a signature function. For example, a fingerprint for a tree T can be produced as a random sample drawn from the pivot multiset $\mathcal{S}(T)$. One may also choose to produce a weighted random sample where the weight of a pivot is proportional to its multiplicity in the multiset $\mathcal{S}(T)$. Here, we design a signature function that is inspired from the popular *MinHashing* technique (short for Minwise Independent Permutation Hashing) [15], [13]. It guarantees that the two sets are similar if and only if the respective signatures are similar [13]. The similarity between two given trees can then be estimated by comparing their respective minhash signatures.

$$sim(T_1, T_2) = \frac{|\mathcal{S}(T_1) \cap \mathcal{S}(T_2)|}{|\mathcal{S}(T_1) \cup \mathcal{S}(T_2)|} \approx \frac{|sig(T_1) \cap sig(T_2)|}{|sig(T_1) \cup sig(T_2)|} \quad (3)$$

Method: For a given tree T, we first construct the pivot multiset $\mathcal{S}(T)$ according to Algorithm 1. We then construct the tree signature $sig(T)$ via minhashing. To this purpose, we hash each pivot $p \in \mathcal{S}(T)$ using the following hash function:

$$ph(p) = (a_1 \cdot lca + a_2 \cdot u + a_3 \cdot v)\ mod\ P \quad (4)$$

where p is an embedded pivot (lca, u, v), P is a large prime number, and $a_1, a_2, a_3 \in \mathbb{Z}_P$. Note that the node labels are used in the above multiplication. Alphanumeric labels are converted to numbers by using methods like Karp-Rabin algorithm [27]. This pivot hash function is sufficiently random and gives low probability of collision [24]. The pivot multiset is essentially a multiset of numbers less than P where each number denotes a single pivot.

The basic idea in signature construction is to randomly permute the universe of pivots, and hash the pivot set under that permutation. For a given permutation π_i, the index of the first pivot that belongs to set $\mathcal{S}(T)$ is produced as its minhash value $h_i(T)$. It has been shown that for a random permutation,

the probability with which two pivot sets produce the same hash value is equal to the Jaccard similarity of those sets [13], [14]. Since the scheme is probabilistic, it is likely to have false positives and false negatives. They are minimized by repeating the process k times, resulting in $k-$MinHashes.

$$sig(T) = \{h_i(T) = \min_{p \in \mathcal{S}(T)} \pi_i(\ ph(p)\ ), 1 \leq i \leq k\}$$

where π_i's are random permutations over the universe of pivots. For a universe of large size, explicit construction of these permutations is very expensive. Broder *et al.* [13] showed that when the universe of elements is $\{0, 1, \cdots, P-1\}$ for some prime P, one can instead consider a family of permutations of the form:

$$\pi_i(x) = a_i \cdot x + b_i \ \ (mod\ M)$$

where $a_i \in \mathbb{Z}_M^*$, $b \in \mathbb{Z}_M$, and M is a prime number that is not smaller than the universe size. They showed that the performance of such linear hash functions is as good as random permutations. Since the universe size is equal to P (see Equation 4), we have $M \geq P$. As shown in Equation 3, the similarity between T_1 and T_2 can then be estimated to be the Jaccard similarity of their signatures $sig(T_1)$ and $sig(T_2)$.

Along with the pivot value that is being hashed, we also store its multiplicity in $h_i(\cdot)$. Therefore, the intersection (union, resp.) of signatures is computed using the multiset extension i.e. by considering the minimum (maximum, resp.) of the two multiplicities involved. The entire procedure to compute the similarity between two trees is summarized as Algorithm 2.

Algorithm 2 Similarity between two trees T_1 and T_2

Input: T_1, T_2, ph, hash family $\mathcal{H} = \{(a_i, b_i), 1 \leq i \leq k)\}$, M

1: construct pivot sets $\mathcal{S}(T_1)$, $\mathcal{S}(T_1)$ using Alg. 1
2: $sig(T_1) \leftarrow sketch(\mathcal{S}(T_1), ph, \mathcal{H})$
3: $sig(T_2) \leftarrow sketch(\mathcal{S}(T_2), ph, \mathcal{H})$
4: compute Jaccard of $sig(T_1)$ and $sig(T_2)$

Algorithm 3 Tree Sketching ($sketch$)

Input: $\mathcal{S}(T)$, ph, hash family $\mathcal{H} = \{(a_i, b_i), 1 \leq i \leq k)\}$, M
Output: $sig(T)$

1: **for each** $p \in \mathcal{S}(T)$ **do**
2: add $ph(p)$ to $\mathcal{S}'(T)$ (Equation 4)
3: **for** $i = 1$ to k **do**
4: $min \leftarrow 0$
5: **for each** $p \in \mathcal{S}'(T)$ **do**
6: $hash \leftarrow a_i \cdot p + b_i$ (mod M)
7: **if** $min < hash$ **then**
8: $min \leftarrow hash$
9: add min to $sig(T)$

C. Different Transformation Functions

A key benefit in the above described approach is that by carefully choosing the transformation function, one can adjust the type of pivot substructure and thereby control the type and amount of tree structure that is to be compared. To exhibit this flexibility in our approach, we now briefly describe some of the alternative definitions for pivot substructure. This list by no means an exhaustive one. The exact choice, as mentioned earlier, is driven by the requirements of underlying application. We present these functions by describing the type of pivot substructures that they consider.

• *Unordered pivots* ($\mathbf{tf_e^u}$): The pivots discussed thus far in this article are unordered embedded pivots where there is no specific order defined on the two children nodes.

• *Ordered pivots* ($\mathbf{tf_e^o}$): Here, there is particular order defined among the two children nodes of the pivot. Typically this order is given by the underlying data. Such ordered pivots are useful in several application domains including computational linguistics, bioinformatics, and document-centric XML where the data is modeled as ordered trees. Such a transformation function is implemented in a similar manner to Algorithm 1 except for the swap operation in Lines $5-6$. Note that when the labels are not swapped, *different* orientations of an unordered tree produce *different* pivot sets.

• *Constrained pivots* ($\mathbf{tf_c}$): The transformation functions considered so far produce pivot sets which are inclusive of all possible pivots. For instance, Embedded pivots discussed earlier present the global structure of the tree since each node is paired with every other node, whenever possible. In contrast, one may be interested only in the local structure around tree nodes. The transformation functions can be adapted to such scenarios by providing additional constraints representing the type of local substructure one wants to focus on. There may exist two types of constraints: *node constraints* and *edge constraints*. Node level constraints allow the user to specify a set of nodes of interest $U = \{u_1, u_2, \cdots\}$ in the tree. The transformation function then produces only those pivots (lca, u, v) where $lca, u, v \in U$. Such node specific constraints can easily be pushed into the construction process in Algorithm 1 (Lines 2 and 4). As we show later in Section IV-C, node-constrained pivots are useful in phylogenetic studies, where the biologists are interested in studying the evolutionary relationships among known taxa that is typically present at leaf nodes (i.e., U).

In case of embedded pivots, each edge denotes some ancestor-descendant relationship. Local structure around tree nodes can be emphasized by providing constraints on the edges. Here, transformation functions are defined to restrict the neighborhood within which the pivot nodes are located. Only those pivots where the root (lca) node is within certain number of hops from the other two nodes. More specifically, this function constructs pivots of the form (lca, u, v) where $|level(lca) - level(u)| \leq \theta$ and $|level(lca) - level(v)| \leq \theta$, for a user defined parameter θ. It is fairly easy to incorporate such *level constraints* in to Algorithm 1 (at Line 4). Focusing on local structure around tree nodes can be useful in many

applications. Consider a graphics or a vision application [28] where the images are represented using some space partitioning data structure like kd-tree. Level-constrained pivots in this case correspond to the tree structure that is localized to some regions or parts of the underlying image. Such localization may be useful, for example, if it is known that one part of the image is unlikely to be correlated to another region that is far away on the image.

• *Induced pivots* ($\mathbf{tf_i}$): In contrast to embedded pivots which preserve ancestor-descendant relationships, induced pivots maintain only parent-child relations among nodes. In an induced pivot (lca, u, v), lca is the parent of both u and v. Note that, induced pivots are a spacial case of level-constrained pivots where $\theta = 1$.

• *Embedded pivots with levels* ($\mathbf{tf_e^l}$): Here embedded pivots are *specialized* by annotating with the level information. The annotation can be done in two ways: each node in the pivot is attached with its level; each edge is attached with the level difference in corresponding nodes. In the former case, each pivot is a 6-tuple $(lca, l_1, u, l_2, v, l_3)$ where $l_1 = level(lca)$, $l_2 = level(u)$, and l_3=$level(v)$. In the case where edges are annotated, each pivot is defined as a 5-tuple (lca,l_1,u,l_2,v) where l_1=$level(u)-level(lca)$ and l_2=$level(v)-level(lca)$. Unlike embedded pivots described earlier, these pivots embody more structural properties present among tree nodes. In fact, this particular transformation function provides a unique mapping between trees and pivot sets. Please see Section III-C for a detailed proof.

III. Theoretical Analysis

We first introduce some terminology used in this section. For a given tree T, we denote its root node by $r(T)$ and its depth or height by $d(T)$. Similarly, the depth of a given node $u \in T$ is denoted by c_u. An orientation of an unordered tree T can be characterized by a *level code* $c_1 c_2 \cdots c_n$ where node i in preorder appears on level c_i. Among all possible orientations of a given unordered tree, the one with lexicographically largest level code is called as the *canonical orientation* and the corresponding code is called as the *canonical level code*. Note that in a canonical orientation, the subtrees in each family are in non-increasing lexicographic order.

A. Bounds on Node Level Pivots

Let $\mathbb{N}(v) = |\mathcal{S}^T(v)|$ denote the number of pivots in which a particular node v presents. As mentioned in Section II-A, this number can be divided into two parts $\mathcal{N}^T_{child}(v) = ||$ and $\mathcal{N}^T_{root}(v)$ As mentioned earlier, the number of pivots involving a particular node $v \in T$ can be partitioned into two subsets $\mathcal{N}^T_{child}(v) = |\mathcal{S}^T_{child}(w)|$ and $\mathcal{N}^T_{root}(v) = |\mathcal{S}^T_{root}(w)|$. Note that $\mathcal{N}^T_{child}(v)$ is equal to the number of nodes with which v can be paired in a pivot, which does not include both its ancestors and descendants. Similarly, $\mathcal{N}^T_{root}(v)$ depends solely on the number of the descendants that v has in T. We thus have,

$$\begin{aligned} \mathcal{N}_{child}(v) &= n - 1 - |\mathcal{A}(v)| - |\mathcal{D}(v)| \\ \mathcal{N}_{root}(v) &\leq \tfrac{|\mathcal{D}(v)|\cdot(|\mathcal{D}(v)|-1)}{2} \quad (from\ Eq\ 1) \end{aligned}$$

By combining these two relations, we get:

$$\begin{aligned} \mathcal{N}(v) &\leq [n - 1 - |\mathcal{A}(v)| - |\mathcal{D}(v)|] + \tfrac{|\mathcal{D}(v)|\cdot(|\mathcal{D}(v)|-1)}{2} \\ &\leq n - 1 - |\mathcal{A}(v)| + \tfrac{|\mathcal{D}(v)|\cdot(|\mathcal{D}(v)|-3)}{2} \end{aligned} \tag{5}$$

B. Incremental Construction

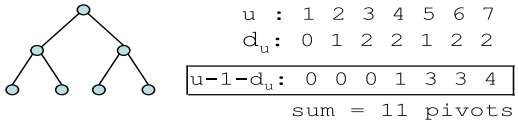

Fig. 2. Computation of $\mathcal{N}(T)$ using Property 3.1

In many XML applications, *document order* i.e. the order that exists among nodes within a single document is very important. For example, operations like XML canonicalization [1] process the nodes in ascending document order. Document order is also important in domains like XSLT. When the nodes are considered in document order, we show that pivot sets can be computed in a much elegant fashion by leveraging the following observation.

Property 3.1: Let T be a tree with associated pivot set $\mathcal{S}(T)$. If a new node u is attached to an existing node $v \in T$ resulting in a new tree T', then $\mathcal{S}(T') = \mathcal{S}(T)$ + $\mathcal{S}^{T'}(u)$.

This property states that the existing pivots are *not* altered when new nodes are attached to the tree (as leaf nodes). For a given pivot (lca, u, v), no addition of a leaf node can alter the ancestor-descendant relations between lca and u, v. Such a property allows us to decompose a pivot set into multiple disjoint sets. For example, if the root node $r = r(T)$ has k children u_1,$\cdots$,u_k (in that document order) then $\mathcal{S}(T)$ can be written as $\mathcal{S}^T(r) + \cup_{i=1}^{k} \mathcal{S}(T(u_i))$, where $T(u_i)$ is the tree rooted at u_i.

While processing a given XML tree T in document order, Property 3.1 allows us to construct $\mathcal{S}(T)$ in an incremental manner i.e., as and when the new nodes are encountered in that document order. If u is the next node in document order then the new set of pivots that are introduced due to u can simply be constructed by pairing u with other existing nodes in T except for its ancestors. The number of such pivots is mainly dependent on the number of nodes which appear before u in document order, and on the depth of node u (see Figure 2). Therefore, we can quickly compute the pivot set size in *linear time* by scanning the tree exactly once, without actually computing the set itself. Such a quick size estimation technique can be helpful in reducing the search space in applications which involve node-by-node processing.

C. Relation to Perfect Hashing

For the transformation $\mathbf{tf_e^l}$ where the pivots are annotated with level information, we prove the following main result:

Theorem 3.2: The function $\mathbf{tf_e^l} : \mathbb{T} \rightarrow \mathbb{S}$ is injective i.e., given T_1 and T_2, if $T_1 \neq T_2$ then $\mathcal{S}(T_1) \neq \mathcal{S}(T_2)$.

[1] http://www.w3.org/TR/xml-c14n

This result guarantees that for every tree there is a unique pivot set – akin to perfect hashing. We first prove this theorem in the context of *unordered, unlabeled* trees, and subsequently extend this result for trees with node labels.

1) Unlabeled Trees: Before we delve into the details of the proof, we introduce two concepts: *tree merge*; and *structural equivalence* between nodes *within* a single tree.

Definition 3.3: Tree Merge: Consider two trees T_1 and T_2 with canonical level codes $a_1 \cdots a_m$ and $b_1 \cdots b_n$, respectively. Say, $\forall i < k \ a_i = b_i$, and $a_k > b_k$ i.e., k is the first location at which both codes differ. We define a merged tree $T_1 \otimes T_2$ to be a tree with the following level code:

$$\begin{aligned} Code(T_1 \otimes T_2) &= Code(T_1) \otimes Code(T_2) \\ &= \begin{matrix} [\, a_1 \cdots a_{k-1} a_k \cdots a_m \\ \otimes \\ b_1 \cdots b_{k-1} b_k \cdots b_n \,] \end{matrix} \\ &= a_1 \cdots a_{k-1} a_k [a_{k+1} \cdots a_m \bowtie b_k \cdots b_n] \end{aligned}$$

where $[a_{k+1} \cdots a_m \bowtie b_k \cdots b_n]$ is some valid combination of level codes. It is guaranteed that such a combination exists. The tree merge operation essentially creates a larger tree with $m+n-k+1$ nodes such that the first k are nodes are similar to T_1, and rest of the nodes from T_1 and T_2 are added in some valid combination. A *simple* case of tree merge operation is illustrated in Figure 3. In this particular example, the valid combination is obtained through simple block movement of level codes.

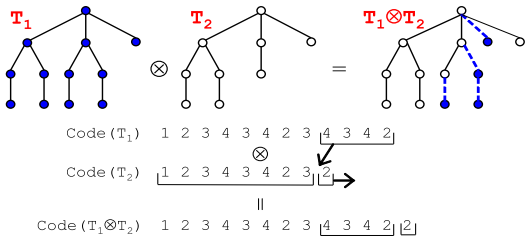

Fig. 3. Tree Merging

Definition 3.4: Structural Equivalence: Two nodes u and v in a tree T are said to be structurally indistinguishable (or structurally equivalent) if and only if the following two conditions hold:

i) Subtrees $T(u)$ and $T(v)$ possess the same structure.

ii) Either u and v are sibling nodes or their parent nodes are structurally indistinguishable.

This definition essentially states that u and v are placed in T in such a way that they can be swapped without affecting the tree structure. By permuting such structurally indistinguishable nodes one can enumerate different *automorphisms* [2] of T. The recursion in the definition is carried out until the lowest common ancestor of the two nodes is encountered, at which point the two parent nodes are siblings. This implies that both u and v are at the same depth in the tree. A pictorial demonstration of the definition is shown in Figure 4. If u and v (whose LCA is w) are structurally equivalent then the branches of $T(w)$ [3] which contain u and v are equivalent.

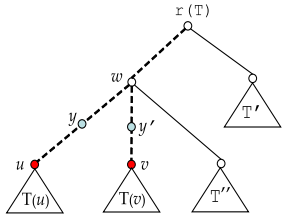

Fig. 4. Structurally indistinguishable nodes (Theorem 3.5)

Theorem 3.5: Two tree nodes u and v in T are structurally indistinguishable if and only if $\mathcal{S}^T(u) = \mathcal{S}^T(v)$.

Proof: This result stems from the fact that the embedded pivots involving a given node capture the entire tree structure when seen from that node. Also, the pivots now are specialized by annotating edges with level difference.

Only If part: Recall that the pivots $\mathcal{S}^T(u)$ involving a single node u can be partitioned into two subsets $\mathcal{S}^T_{root}(u)$ and $\mathcal{S}^T_{child}(u)$. The first condition in Definition 3.4 implies that $\mathcal{S}^T_{root}(u) = \mathcal{S}^T_{root}(v)$. All the remaining pivots in $\mathcal{S}^T_{child}(u)$ are of the form $(-,-,t,-,x)$ where t is either u or v, and x is some node that is neither an ancestor nor a descendant of both u and v. We then have two cases:

- $x \notin T(w)$: In this case, x is a node that is present in T' (see Figure 4. Since the node label and the depth are same for both u and v due to their structural equivalence, all pivots of the form $(-,-,u,-,x)$ and the pivots $(-,-,v,-,x)$ are equivalent.
- $x \in T(w)$: Here, x can either be part of T'' or is a node that is on the path $w \rightsquigarrow u$ or $w \rightsquigarrow v$ (i.e., an ancestor node to either u or v). If $x \in T''$ then one can make the same argument as in previous case ($x \notin T(w)$). If $x \notin T''$ and x is on the path $w \rightsquigarrow u$ then there must exists x' that is on the path $w \rightsquigarrow v$ such that x and x' are structurally equivalent – second condition in Def. 3.4. In such a case, the pivot $(y,-,x,-,v)$ that belongs to $\mathcal{S}^T_{child}(v)$ will have an equivalent pivot $(y',-,x',-,u)$ in $\mathcal{S}^T_{child}(u)$ in such a way that y and y' are structurally equivalent. We can equivalently argue the case where x is on the path $w \rightsquigarrow v$. We thus proveed that $\mathcal{S}^T_{child}(u) = \mathcal{S}^T_{child}(v)$ if u and v are structurally equivalent.

If part: We show that this part of the theorem holds by proving its contrapositive i.e., if u and v are *not* structurally equivalent then $\mathcal{S}^T(u) \neq \mathcal{S}^T(v)$. If they are not equivalent then one of the two conditions in Definition 3.4 must fail.

- $T(u) \neq T(v)$: Let the canonical level codes of $T(u)$ and $T(v)$ be $c_1 \cdots c_m$ and $d_1 \cdots d_n$, respectively. If the subtrees differ in their structure then there must exist $x \in T(u)$ and $x' \in T(v)$ such that $d(x)$=c_k $\neq$ d_k=$d(x')$ for some $k \leq$ min(m,n), where k is the position at which both level codes differ. Now consider the specific pivots p_1 formed by u and the node in

[2] A tree automorphism is an isomorphism from a tree to itself.

[3] $T(w)$ is the subtree of T that is rooted at w.

$T(v)$ i.e., $(w, -, u, d_k, x') \in \mathcal{S}^T_{child}(u)$ and similarly $p_2 = (w, -, v, c_k, x) \in \mathcal{S}^T_{child}(v)$. Since $c_k \neq d_k$, we have $p_1 \neq p_2$.

- Now consider an instance where the second condition of Definition 4 fails. Let the parent nodes of u and v are z and z', respectively. If z and z' are not structurally equivalent then, by a similar argument from preivous case, we can find at least one node in each of $T(z)$ and $T(z')$ such that the nodes differ in their depth. By pairing them with u and v, as we did in previous case, we can find at least one pivot that is in $\mathcal{S}^T_{child}(u)$ but not in $\mathcal{S}^T_{child}(v)$.

We thus proved that nodes u and v are structurally equivalent if and only if $\mathcal{S}^T(u) = \mathcal{S}^T(v)$. ■

We use the above result to prove Theorem 3.2 – the function $\mathbf{tf}^{\mathbf{l}}_{\mathbf{e}}$ is injective.

Proof of Theorem 3.2 Consider two cases: $d(T_1) \neq d(T_2)$; and $d(T_1) = d(T_2)$. In the former case, without loss of generality assume that $d(T_1) > d(T_2)$. Since all trees are branched, there must exist at least one pivot of the form $(r(T_1), d(T_1), -, -, -)$ that is in $\mathcal{S}(T_1)$ but not in $\mathcal{S}(T_2)$. Thus, $\mathcal{S}(T_1) \neq \mathcal{S}(T_2)$.

Now consider the case where $d(T_1) = d(T_2) = d$. We prove this case by contradiction i.e., we assume $\mathcal{S}(T_1) = \mathcal{S}(T_2)$ and argue that it is not possible. Let $Code1$ and $Code2$ are canonical codes for T_1 and T_2. Assume that $Code1 > Code2$ and k be the smallest index at which both the codes differ. Since $d(T_1) = d(T_2)$, we have $k > d$. This instance is summarized below:

$$Code(T_1) = Code1 = a_1 \cdots a_{k-1} a_k \cdots a_m$$
$$Code(T_2) = Code2 = b_1 \cdots b_{k-1} b_k \cdots b_n$$
$$WLOG: Code1 > Code2 \;\; i.e., \;\; \forall i < k, a_i = b_i; \;\; a_k > b_k$$

Since the first $k-1$ nodes in both the trees are same and since we assumed $\mathcal{S}(T_1) = \mathcal{S}(T_2)$, we must have k' such that:

$$\exists k' \in T_2, k < k' \leq n \;\; such\ that \;\; \mathcal{S}^{T_1}(k) = \mathcal{S}^{T_2}(k') \quad (6)$$

Now consider the merged tree $T_3 = T_1 \otimes T_2$ as defined in Def. 3.3. Note that both k and k' belong to the merged tree. From Property 3.1, Equation 6, and from the way merged tree is constructed, we can derive that the pivot sets involving k and k' in the merged tree are same i.e., $\mathcal{S}^{T_3}(k) = \mathcal{S}^{T_3}(k')$. In other words, both k and k' are structurally indistinguishable in T_3 (from Theorem 3.5). However, such a node k' can not exist in T_2 since we assumed that $Code2$ is canonical and $Code2 < Code1$. Therefore, our assumption $\mathcal{S}(T_1) = \mathcal{S}(T_2)$ can not hold true when $T_1 \neq T_2$. □

2) Labeled Trees: We can easily extend the definition of structural equivalence (Def. 3.4) to labeled trees. Based on this definition, we provide the following theorem.

Theorem 3.6: Two nodes u and v in a labeled unordered tree T are indistinguishable if and only if $\mathcal{S}^T(u) = \mathcal{S}^T(v)$.

Intuition The proof is similar to the one described for Theorem 3.5. It however relies on the fact that node labels *specialize* the pivots by making them more distinct. If two pivot sets S_1 and S_2 are *not same* when labels are not considered then they must be different even when labeles are taken into account. In contrast, if $S_1 = S_2$ when there are no labels, then the sets might or might not be the same when labels are considered. Therefore, the structural equivalence between u and v in the unlabeled version of T is a necessary but not sufficient condition for their equivalence in T. □

Theorem 3.7: If T_1 and T_2 are two labeled trees with different structure then $\mathcal{S}(T_1) \neq \mathcal{S}(T_2)$.

Proof: From Theorem 3.5, if two unlabeled trees differ in their structure then their corresponding pivot sets will be different. Since the unlabeled pivot set equivalence is a necessary condition, the labeled pivot sets of the trees are also different. ■

Theorem 3.8: Consider two labeled trees T_1 and T_2 with exact same structure. If $T_1 \neq T_2$ i.e. they differ in node labels then $\mathcal{S}(T_1) \neq \mathcal{S}(T_2)$.

Intuition By using Theorem 3.6, let the structurally equivalent tree nodes of T_1 are clustered into groups $G_1 \cdots G_k$. Permuting nodes within each such group results in different automorphisms of T_1. We can then argue that T_1 and T_2 will have same pivot sets if and only if they have identical group structure i.e., they are automorphisms of each other. □

D. Relation to Tree Edit Distance

The most commonly used distance measure on tree structured data is tree edit distance [29], [30], which denote the the minimum number of basic edit operations (relabel, delete, and insert) to transform one tree into the other. Some researchers have used a variant of this basic measure that includes *subtree moves*, which allow a subtree to be moved under a new node in the tree in one step [22] – see Figure 5. In this section, we briefly present the relation between this variant of tree edit distance (with subtree moves) with our tree similarity measure from Section II-A. Later in Section IV, we empirically evaluate this relation on different datasets.

Theorem 3.9: Consider two trees T_1 and T_2. If $\mathcal{S}(T_1)$=$\mathcal{S}(T_2)$ then the maximum edit distance between them is equal to $m + n - 6$, where $m = |T_1|$ and $n = |T_2|$.

Proof: Since we are interested in the maximum edit distance, we choose two extreme cases discussed in Section II-A for T_1 and T_2. Let T_1 and T_2 be the tree structures similar to the ones shown in Figure 1a & c, respectively. We thus have, $\mathcal{N}(T_1) = \frac{(n-1)\cdot(n-2)}{2}$ and $\mathcal{N}(T_2) = m - 2$.

If $\mathcal{S}(T_1) = \mathcal{S}(T_2)$ then $\frac{(n-1)\cdot(n-2)}{2}$ must be equal to $m-2$ ($m > n$). Note that the choice of T_2 is very important to find the maximum edit distance. We can not choose any other tree structure with more than m nodes for T_2 – there can not exist a larger tree that has the same number of pivots as that of T_1. This is because T_2 is a m-node tree structure with minimum number of pivots. Also, if we consider trees with size less than m then we may not get the *maximum* distance. Therefore, the choice of T_2 is justified to find the maximum edit distance.

Now, consider the edit distance between $T1$ and $T2$. To convert $T1$ in Fig. 1a into $T2$ in Fig. 1c, we need $(n-3)$

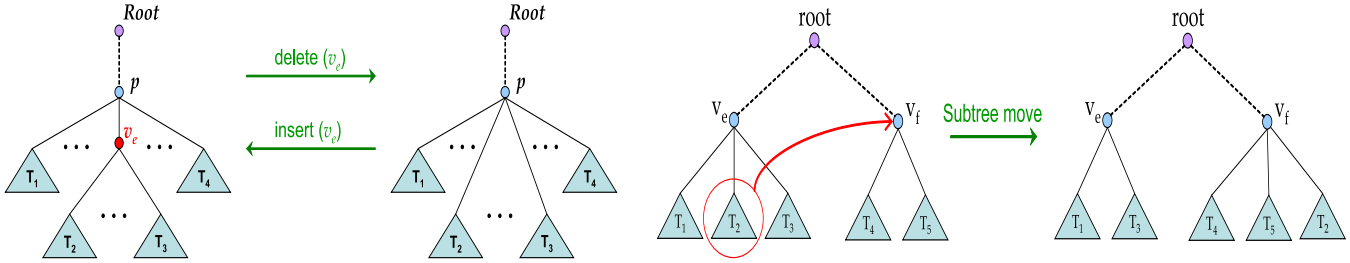

Fig. 5. Edit operations – (a) Insert and Delete (b) Subtree Move

deletions and $(n-3)+(m-n)$ additions. This implies that the edit distance between trees with same pivot sets can not exceed $m+n-6$. ■

For each of the edit operations, we constructed some lower bounds on the number changes in the pivot set, especially in $\mathcal{S}(T_1) \cap \mathcal{S}(T_2)$ [4]. These bounds provide an intuition as to how pivot sets change when different edit operations are applied on a tree.

1) Relabel v_e: When a particular node label is changed then only those pivots which contain the modified node are altered. They include pivots both $\mathcal{S}^{T_1}_{child}(v_e)$ and $\mathcal{S}^{T_1}_{root}(v_e)$, whose number is equal to $\mathcal{N}^{T_1}(v)$ as given in Section III-A. We can then construct a lower bound on the pivot multiset intersection (the numerator in Jaccard similarity) as follows:

$$\begin{aligned} &\mathcal{S}(T_1) \cap \mathcal{S}(T_2) = \mathcal{S}(T_1) - \mathcal{S}^{T_1}_{root}(v_e) - \mathcal{S}^{T_1}_{child}(v_e) \\ &|\mathcal{S}(T_1) \cap \mathcal{S}(T_2)| \\ &= \mathcal{N}(T_1) - \mathcal{N}^{T_1}(v_e) \\ &\geq \mathcal{N}(T_1) - [n-1-|\mathcal{A}(v_e)| + \tfrac{|\mathcal{D}(v_e)|\cdot(|\mathcal{D}(v_e)|-3)}{2}] \\ &\geq \mathcal{N}(T_1) - \tfrac{|\mathcal{D}(v_e)|\cdot(|\mathcal{D}(v_e)|-3)}{2} - n \end{aligned} \tag{7}$$

2) Delete v_e: When v_e is deleted, the children of v_e are attached to the parent of v_e (say, p_{v_e}) in the resulting tree. Suppose $\overline{T_1(v_e)}$ be the tree that does not contain v_e and its descendants i.e., $\overline{T_1(v_e)} = T_1 - T_1(v_e)$. Note that the pivots that belong to $\overline{T_1(v_e)}$ are not affected by the deletion of v_e. Therefore one needs to analyze on the pivots involving nodes from $T_1(v_e)$. Let us first consider v_e itself. All the pivots in which v_e is the child are eliminated, and hence they do not belong to new pivot set i.e., $\mathcal{S}^{T_1}_{child}(v_e) \cap \mathcal{S}(T_2) \neq \phi$. For every pivot in $\mathcal{S}^{T_1}_{root}(v_e)$, the root node is modified to p_{v_e}.

Now consider other nodes $x \in \mathcal{D}(v_e)$. All the pivots of the form $(z,-,x,-,y)$ where $y,z \in \mathcal{D}(v_e)$ are not affected by the deletion. Only those pivots in which x is paired up with nodes $y \in \overline{T_1(v_e)} - \mathcal{A}(v_e)$ are affected. More precisely, a pivot $(-,d_x,x,-,y)$ will be changed to $(-,d_x-1,x,-,y)$ in T_2 – the depth of x is reduced by 1 due to the deletion of v_e. Let this set be denoted as S', whose exact size depends on the number of ancestors and descendants that v_e has in T_1.

$$S' = \{(-,-,x,-,y) | x \in \mathcal{D}(v_e) \wedge y \in \overline{T_1(v_e)} - \mathcal{A}(v_e)\}$$

$$\mathcal{S}(T_1) \cap \mathcal{S}(T_2) = \mathcal{S}(T_1) - \mathcal{S}^{T_1}(v_e) - S'$$

For convenience, let $A = |\mathcal{A}(v_e)|$ and $D = |\mathcal{D}(v_e)|$. We can then bound the intersection size as follows.

$$\begin{aligned} &|S'| = (\ n - |T_1(v_e)| - A\ ) \cdot D \\ &|\mathcal{S}(T_1) \cap \mathcal{S}(T_2)| \\ &= \mathcal{N}(T_1) - \mathcal{N}^{T_1}(v_e) - |S'| \\ &\geq \mathcal{N}(T_1) - (n-1-A-\tfrac{D\cdot(D-3)}{2}) - |S'| \\ &\geq \mathcal{N}(T_1) - (n-A)\cdot(1-D) - 1 - \tfrac{D\cdot(D+5)}{2} \end{aligned} \tag{8}$$

For example, if v_e is a leaf node then the above expression reduces to $\mathcal{N}(T_1) - (n-A-1)$, where $(n-A-1)$ is exactly equal to the number of pivots that v_e has in T_1 i.e. $|\mathcal{S}^{T_1}_{child}(v_e)|$.

3) Insert v_e: Insertion is a dual to deletion that is discussed above. Therefore, the pivots which get eliminated during deletion, for example $\mathcal{S}^{T_1}_{child}(v_e)$, are introduced into T_2 due to insertion. Similarly the pivots which get introduced while deletion are elimanated during insertion. Hence, the change in the number of pivots is same as the bounds shown in previous section except that A and D now refer to the number of ancestors and descendants of v_e in the modified tree T_2, respectively.

4) Move v_e to v_f: Now consider the last edit operation, *move*. Here, a non-root node along with the subtree rooted at node is moved from one tree node to the other. It is depicted in Figure 5b. $T(v_e)$ that is under the node w_1 in T_1 is moved under w_2, resulting in T_2. In such a case, the set $\mathcal{S}^{T_1}_{child}(v_e)$ changes completely, especially when the depth of v_e in both the trees is different. However, the pivots $\mathcal{S}^{T_1}_{root}(v_e)$ are unaffected by the move operation. Similar to delete operation, the pivots involving descendants of v_e and rest of the tree nodes change from T_1 to T_2. They are of the form $(-,-,x,-,y)$ where $x \in T(v_e)$ and $y \in T_1 - T_1(v_e)$. Therefore, the bound derived for this operation will be similar to the one for delete operation except that the set $\mathcal{S}^{T_1}_{root}(v_e)$ that is eliminated during deletion is unaffected during move operation.

[4]Note that these are not the bounds for our Jaccard similarity.

IV. RESULTS

We consider several publicly available[5] tree-structured datasets drawn from a range of real-world applications including bioinformatics (Swissprot), linguistics (Treebank), Web log analysis (CSlogs) and XMark (online auctions), for our evaluation study. Among these, XMark is a synthetic dataset that models the behavior of an online auction site and is useful for controlled experiments. The size of XMark data trees is controlled by the *scaling factor*, an input to the generator. It produces one large single tree. We remove the top-level XML tags such as $<site>$, $<regions>$, $<closed\_auctions>$ to generate a number of small trees. All the results presented in this section are obtained by using signatures with 16 min-hashes. We did not observe any improvements with signature sizes greater than 16.

Basic Construction Costs: In Figure 7a we present some basic statistics on the overall time spent in constructing signatures (including cost of transformation and signature sketching operators) for different transformation functions. As noted earlier the computational cost associated with the induced pivot function $\mathbf{tf_i}$ are expected to be linear in the size of the tree while signatures extracted from than embedded pivot function $\mathbf{tf_e}$, are expected to be quadratic in tree size[6]. Costs for both functions grow linearly with the database size. It should be pointed out that our construction algorithm exhibits embarrassingly parallelism and thus the increased cost of computing embedded pivots can be mitigated significantly by leveraging the capabilities of modern multicore systems. For example $\mathbf{tf_e}$ on Swissprot on a dual quad core system took about 67.3 seconds – a 7.98-fold improvement over a sequential version that takes 535 seconds. Figure 6a demonstrates the the scalability of our hashing algorithms as we vary the tree size. The signature construction time depends directly on the number of tree nodes. Similarly, Figure 6b shows that the run time scales well as we increase the size of the database.

Information Content: If we treat a pivot set as a message that describes the associated tree then the amount of information contained in the message (i.e., pivot set) can be quantified as the *information entropy* ($\sum plogp$) of the set. We can use this measure to compare amongst multiple pivot set strategies. Let T be a tree from the XMark dataset whose labels have been removed. We subsequently assign new labels to T chosen from a set L using the following scheme. We first randomly select a special label $\lambda \in L$. We then assign new labels to tree nodes by biasing the distribution towards the special symbol λ with probability b. The remaining probability $1-b$ is equally divided among other labels i.e. each label other than λ is chosen randomly with probability $\frac{1-b}{|\sum|-1}$. Figure 7b depicts the change in entropy for the main pivot set strategies we have discussed, as the label bias b for a XMark tree of size $31,000$ is varied. The main trends, along expected lines are: i) that the information content of embedded pivots with labels dominate the other two strategies across the board; ii) even when the label bias is 1.0 (all nodes have the same label) the entropy of this strategy is not zero suggesting there is still important information in the levels to help disambiguate amongst trees; and iii) for low label bias values the difference between embedded with levels and embedded without levels becomes insignificant. We have observed similar results for a range of trees of different sizes and shapes.

Edit Distance

In this experiment, we empirically relate the edit distance with four operations (relabel, insert, delete, and subtree move) and the Jaccard distance (1 - similarity) obtained via signatures computed in Section II-B. Unfortunately the problem of computing exact edit distance with moves between two given trees is NP-Hard (see Section III-D). Furthermore, to the best of our knowledge, there are no efficient approximation algorithms especially in the context of unordered trees. We therefore adopt a mechanism where a given tree T is subjected to a series of *random perturbations*. As we perform these perturbations, we monitor the change in Jaccard similarity between the original tree T and the perturbed tree T^p.

For the purposes of this experiment, we selected a tree at random from each of our datasets [7]. For each tree, we prepared an *edit script* ES, which is a sequence of edit operations that are denoted as ($node$,$operation$) pairs [22]. In order to avoid any redundant operations, we make sure that no node is deleted twice, no relabeled node is selected for relabel operation again, and no subtree is moved twice. Such a valid edit script is applied on the selected tree resulting in a new perturbed T^p. We then compare the edit distance $|ES|$ with our signature-based distance, $1-$Jaccard(T,T^p) (see Section II-A).

We compared the performance of the three basic strategies we propose, embedded with levels, embedded without levels and induced with two alternative approaches from the literature. The first strategy is based on pq-grams proposed by Augsten *et al.* [8] [18], [19]. Since our data trees are unordered, we have computed *windowed pq-grams* using their 3-step process [19] – sort the unordered tree, extend the tree with dummy nodes, and compute the windowed pq-gram profile. The results shown in Figure 8 are obtained for the default setting of $p = q = 2$ and $w = 3$[9]. In addition to pq-grams we also examined the performance of a path-based strategy denoted *paths*. Here, the transformation function maps the tree into a set of root-to-leaf paths, which are then hashed using standard string hashing methods such as PJW hash [16]. The resulting set of hash values is then used for computing MinHash signatures.

Figure 8 reports on the performance of these five transfor-

[5]Swissprot, Dblp, and Treebank datasets are obtained from `http://www.cs.washington.edu/research/xmldatasets/`, CSlogs from `http://www.cs.rpi.edu/~zaki/software/` and XMark from `www.xml-benchmark.org/`.

[6]In a controlled experiment when we changed the (XMark) tree size from $1,000$ to $10,000$ to $30,000$ nodes, the time taken by $\mathbf{tf_e}$ increased from 0.02sec to 10sec to 112sec (quadratic).

[7]These experiments are representative and similar results were obtained for other trees and other random perturbations.

[8]We have obtained the source code from the authors.

[9]We found this parametric setting to work the best, in a manner similar to those reported by the authors[19]

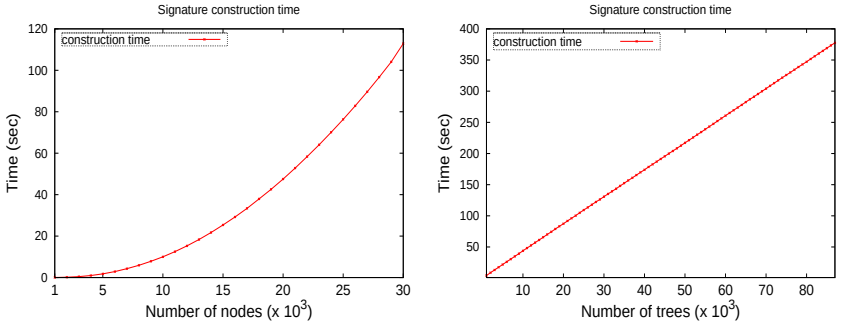

Fig. 6. Construction costs

Data set	# of trees	Tree size		Tree depth		Sketching time (sec)		
		max	avg	max	avg	induced	embedded	emb on 8 cores
Swissprot	50,000	7018	271.5	5	4.8	9.8	535	67.3
Dblp	328,858	4477	32.1	6	3.0	4.4	49	6.2
Treebank	52,851	648	68.0	35	10.4	0.5	31	3.89
Cslogs	59,691	428	12.9	85	3.4	0.5	4	0.48
XMark	280,000	28,000	59.7	10	5.3	6.4	202	25.2

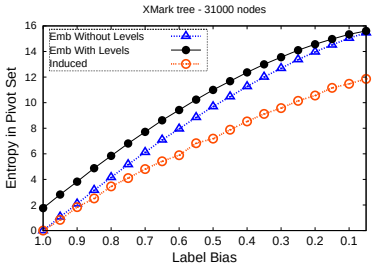

Fig. 7. (a) Datasets and their characteristics, (b) Information content in pivot multisets

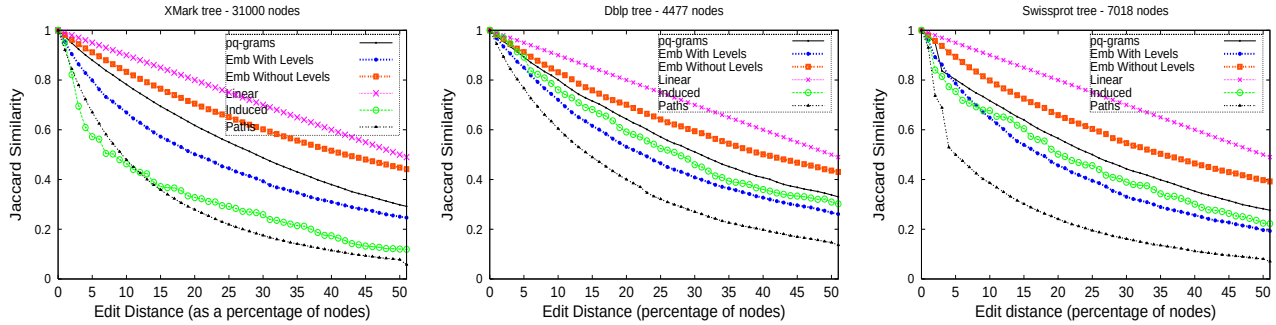

Fig. 8. Approximating Edit Distance

mation strategies for trees from three different datasets. The trend-line labeled "linear" refers to a hypothetical technique that can compute or estimate the exact edit distance, hence a linear relation. The closer we are to this line, the better the approximation of the algorithm.

For all data sets and all perturbation scripts we have evaluated, the transformation function $\mathbf{tf}_{\mathbf{e}}^{\mathbf{u}}$ using embedded pivots without levels dominates, and always provides the best approximation across the board. Similarly the weakest strategy is almost always the path based strategy. In the experiments we have observed the second best strategy is usually a toss up between embedded pivots with levels and the strategy based on pq-grams. When level information is used, the transformation function $\mathbf{tf}_{\mathbf{e}}^{\mathbf{l}}$ behaves similarly to a perfect hash function. Therefore, even a small change in the tree is likely to be exaggerated in the the corresponding pivot set leading to a loss in accuracy in the estimation of the edit distance. Since pq-grams are induced substructures (with some additional structural information) they can only capture the local structure around nodes, they are unable to approximate the actual edit distance as well as the embedded strategy without levels. We should also point out that in terms of execution time, the path-based and basic induced strategies were inexpensive. The embedded strategies were more expensive but were not far off. Computing the pq-gram profiles was found to be the most expensive – on average couple of orders of magnitude more expensive than the other strategies.

A. Case Study I: De-duplication of XML Documents

For our first case study, we consider the application of our hashing algorithms for the purpose of detecting duplicate documents in an XML repository [31], [32]. This problem is related to finding different representations (i.e. duplicates) of a same underlying object. Duplicates are possible in real-world databases due to various reasons like typographical errors, missing data, inconsistent representations.

We performed a detailed evaluation by adopting an approach that is commonly used in evaluating deduplication techniques [32]. We add random *noise* or *artificial duplicates* to the data and then we observe how well our tree signatures can detect these duplicates. For this purpose, we implemented

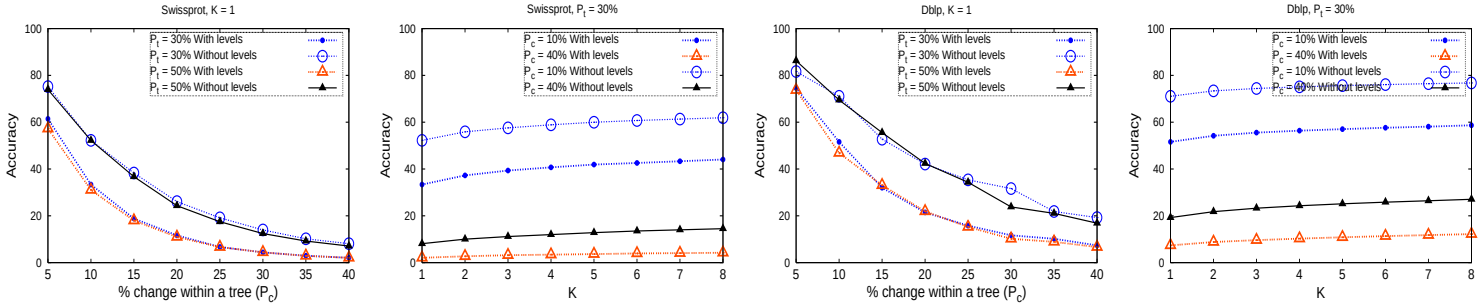

Fig. 9. Accuracy in deduplication: (a & b) Swissprot, (c & d) Dblp

a tool that takes in a data set and generates a data set that is polluted with duplicate documents. Our tool takes two main parameters p_t and p_c where p_t is the percentage of trees that are to be duplicated, and p_c is the amount of change or noise that is added while creating a duplicate. While p_t is denoted as the percentage of the database size, p_c is represented as the percentage of the number of nodes in the tree selected for duplication. For a given p_t and p_c values, the tool carefully generates duplicates by introducing various forms of errors: node deletions representing missing data, node modifications denoting typographical errors or corrupted data; node movements which stand for copy-paste errors; node insertions which correspond to extra noise. Note that the tool's output contains both clean and dirty or noisy documents. For each duplicate document, we obtain top K-nearest neighbors and check if the original document is one of the top nearest neighbor.

Figures 9 & 10 show the evaluation results on Swissprot, Dblp, and XMark datasets for a variety of p_c and p_t values. The y-axis in figures show the percentage of duplicates which are detected (accuracy) by our algorithms. When K is set to 1, we compare a duplicate document with its top nearest neighbor. For such setup, as the amount of change within a tree p_c increases, the accuracy reduces because the duplicate is no longer "similar" to the original document. For all three datasets, the accuracy is not affected by the number of trees that are altered (p_t). For a given p_c, the accuracy is roughly the same for all values of p_t. Notably, the accuracy obtained by $\mathbf{tf_e^l}$ transformation is less than that of $\mathbf{tf_e^u}$ that computes pivots without level information. The performance difference between these two transformation functions is roughly the same for all three data sets. We also show the improvement in accuracy as K is increased from 1 to 8. When $K = 8$, we check whether or not the original document is present in duplicate's top 8 nearest neighbors. The accuracy is increased marginally with K, and it reaches its plateau after $K = 4$, for all datasets.

In this case study, induced pivots produced from $\mathbf{tf_i}$ did not perform very well. For instance, when $p_t = 30\%$, $p_c = 5\%$, and $K = 1$, $\mathbf{tf_i}$ could detect only 28% of duplicates in Swissprot and a mere 5% of duplicates in Dblp. In contrast, embedded pivot transformations $\mathbf{tf_e^u}$ and $\mathbf{tf_e^l}$ detected 75% and 61% on Swissport, and 81% and 75% on Dblp, respectively. We also did not consider pq-grams in this evaluation due to its poor run time performance. We expect their performance to be similar to that of $\mathbf{tf_i}$.

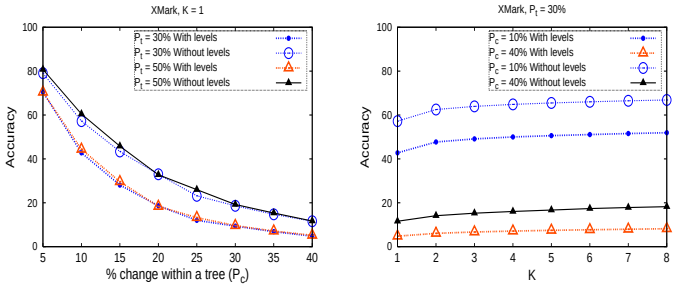

Fig. 10. Duplicate detection in XMark

B. Case Study II: Stratified Sample Generation for Frequent Subtree Mining

For our second case study, we evaluate the use of our algorithms for improving the efficiency of frequent pattern mining algorithms. Hashing helps in grouping similar data records into partitions, and subsequently pattern mining algorithms can run in individual partitions. Such techniques are especially useful when performing large scale analysis using parallel cluster systems [33]. Also, by treating each computed partition as a *strata*, one can generate *stratified* samples of the dataset, which are then used to discover frequent patterns.

Methodology: Here, we consider two popular datasets used in tree mining – Cslogs and Treebank. For each dataset, we compute signatures for all trees (embedded without levels) and then hash similar signatures into K strata ($K = 10$). We perform this grouping by comparing the minhash values in each signature, which is similar to the divide-compute-merge (DCM) algorithm [15]. We then sample from each strata at a given sampling rate (here we fix it to be proportional to the size of the strata), and combine into a single unified sample. We expect the resulting stratified sample to be more robust than one using traditional random sampling, since we are taking the payload into account when sampling over strata. We evaluate this premise by comparing the generated frequent patterns with actual set of patterns obtained from the full dataset, one can compute the precision, recall, and the F-measure. The F-measure is computed as the fraction $\frac{2 \cdot recall \cdot precision}{recall + precision}$.

The first and third charts in Figure 11 compare the effectiveness of stratified sampling when compared to random

Node label	Sequence Name
A	CACHIT
B	PSTCHIT
C	NTACIDCL3
D	S66038
E	CUSSEQ_1
F	CUSSEQ_2
G	CUSSEQ_3
H	VIRECT
I	VURNACH3A
J	ATHCHIA
K	VURNACH3B
L	NTBASICL3

TABLE I

DNA SEQUENCES CONSIDERED IN THE EXPERIMENT FROM SECTION IV-C

sampling in terms of F-measure on Cslogs and Treebank datasets, respectively [10]. The results presented in the figures are averaged over 5 different runs. As expected, stratified sampling using our hashing algorithm, outperforms random sampling from a qualitative perspective across the board for multiple sample sizes. Additionally, since stratified sampling acts upon carefully grouped data records using our proposed hashing schemes, the variance in F-measure is very small when compared to that of random sampling – see second and fourth charts in the figure. High accuracy and small variance of our algorithms make them attractive choices for progressive sampling, where we can use them to quickly find the nature of the learning curve (accuracy vs. sample size) and thereby determine best sample size [34]. The results presented here are representative, we observed similar results for other support levels for both the datasets. For example on Cslogs at 1% support, the variance in F-measure for randomly generated sample was, on average, about 10 times higher than the variance observed when our stratified sampling is used.

C. Case Study III: Phylogenetic analysis

For our final case study, a qualitative one, we demonstrate the use of our algorithms in the context of another application domain *Phylogenetics*. In particular here we will leverage the feature of constrained pivots discussed in Section II-C. Biologists make use of phylogenetic trees in order to study and understand the evolutionary relationships present among given set of organisms. These are usually unordered trees where the leaf nodes represent the given organisms and the internal nodes refer to some ancestor organisms. Phylogenetic analysis often produce a number of candidate trees. Biologists resolve the conflict among them by computing the consensus of these trees. They help in summarizing the most common relationships among output trees [8]. However, the computational complexity in constructing these trees is very high. One possible solution is to split the set of all phylogenies into partitions and construct consensus for each partition. Here, we empirically show that our algorithms are useful in creating effective partitions of phylogenies.

[10]For Treebank, higher support values are chosen due to high associativity present in the dataset.

Methodology: We consider the case of chitinase (digestive enzyme) genes in plants. We take 12 aligned protein coding sequences from these genes [11] (see Table I). From these sequences, we generate 20 different phylogenies by using different algorithms from the Phylip [12] toolbox.

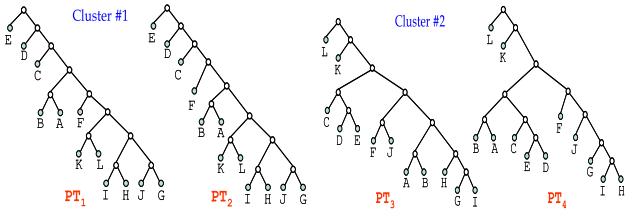

Fig. 12. Case study: Analysis of phylogenetic trees

Since most of the phylogenies are only leaf labeled, we used constrained transformation function $\mathbf{tf_c}$ to construct node constrained pivot sets. We compute signatures for these 20 trees, and use them to group similar trees into clusters. A domain expert manually examining these clusters, found that the trees in each cluster present very similar evolutionary relationships. Two such clusters (each with two trees) are shown in Figure 12. The 12 protein sequences are shown as A, B, $\cdots$, L for simplicity (see Table I). The trees in *cluster #1* are produced by the same algorithm when run with different parameters. On the other hand, the trees PT_3 and PT_4 in *cluster #2* are produced from two different algorithms. For these particular phylogenies, even an induced transformation function $\mathbf{tf_i}$ provided good clustering arrangements. However, in general, constrained embedded pivots are preferable for domain experts since the trees are leaf labeled. Our algorithms were thus able to effectively find the similarities present among different phylogenies. As Stockham *et al.* recently pointed out, the consensus trees obtained over such clusters with similar trees are more resolved than single-tree consensus trees [8]. Furthermore, computation of consensus trees over these clusters is cheaper and effective than computing a single, global, often non-informative consensus tree suggesting that our hashing methods can be an effective preprocessing step for such methods.

V. RELATED WORK

Tree similarity has been studied extensively in the context of tree editing distance [29], [30], which is a natural extension of string edit distance. Similar to the case of string comparisons, there exist a number of ways to compare different trees – largest common subtree [35], smallest common super-tree [36], tree alignment [37], to name a few. Exact computation of conventional tree edit distance is computationally expensive. The fastest known solution takes at least $O(n^3)$ time for ordered trees [29] and it is $\mathcal{NP}$-hard for unordered trees [38]. Even for ordered trees, the problem is $\mathcal{NP}-$hard when the additional "subtree move" operation is

[11]`http://home.cc.umanitoba.ca/~psgendb/GDE/phylogeny/parsimony/chitIII.mrtrans.gde`

[12]`http://evolution.genetics.washington.edu/phylip.html`

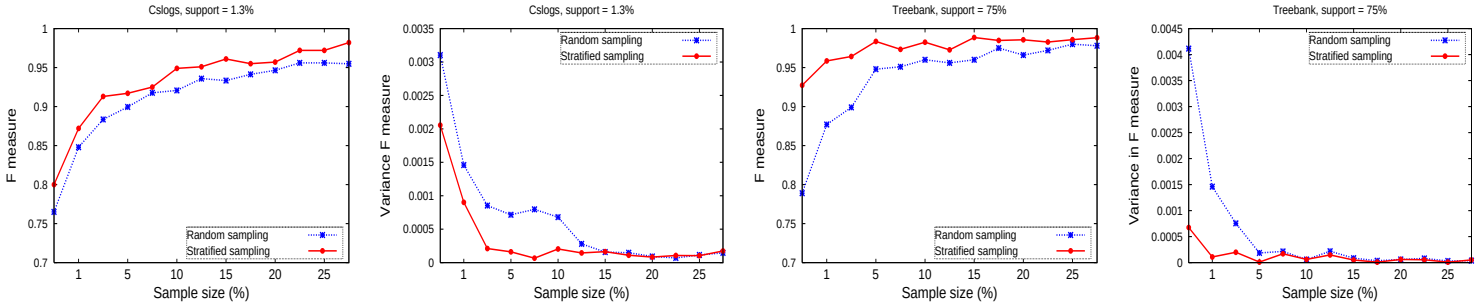

Fig. 11. Evaluation of stratified sampling that is powered by our tree hashing methods

introduced [39]. There has been some efforts in developing approximate algorithms for tree edit distance [18], [19], [20], [22], [21].

Yang *et al.* match two ordered trees by using L_1 distance between corresponding vectors of binary branches. Binary branches are similar to induced pivots produced by our transformation $\mathbf{tf_i}$ as they capture only parent-child relationships around a given tree node. Therefore, they are likely to be ineffective (see Section IV). Augsten *et al.* use pq-grams to measure tree similarity [18]. These pq-grams can be thought of as extensions of binary branches, and they are small subtrees of specific shape, which is controlled by parameters p and q. They recently extended pq-grams for unordered trees by considering different permutations of sibling nodes [19]. However, as we show in Section IV, they do not perform as well as our transformation functions. Furthermore, the parameters p and q must be tuned based on the data. Here, a parameter setting that is suited for tree in the database may not suit the others, especially because of recursive and repetitive structure of XML tags. Guha *et al.* presents a framework for approximate XML joins based on edit distance between ordered trees [31]. Garofalakis and Kumar proposed XML stream processing algorithms for embedding tree edit distance into L_1 space while providing some bounds on the distortion [22]. However unlike our methods, their algorithms focus only on ordered trees. In general, approximating edit distance is a hard problem. Andoni and Krauthgamer recently proved that the computational hardness of estimating string edit distance itself is significantly very high when compared to Hamming distance [40], for which efficient approximate algorithms are known [25]. Recently, Gollapudi and Panigrahy proposed sketching techniques for hierarchical data, which are subsequently leveraged in providing LSH methods under Earth Mover's Distance measure [20]. These methods are developed only for leaf labeled trees. It is not easy to extend them for general node labeled trees.

VI. Conclusions

In this article we have presented a simple yet effective framework for hashing tree structured data. The synopsis of the proposed approach entailed transforming the tree-structured datum into a multi-set of pivot structures and subsequently relying on min-wise hashing to yield a fixed length datum suitable for standard hashing techniques. We examined the performance of induced, embedded without levels, embedded with levels, and constrained pivots from a theoretical perspective and proved that one of them can form the basis for constructing a perfect hash function and also proved lower bounds connecting these strategies with traditional tree edit distance operations. To further enhance the efficacy of the proposed transformations we realized parallel implementations on modern multicore systems with linear time speedups. We demonstrated the utility of the hashing framework on several applications and case studies drawn from the domains of XML deduplication, frequent tree mining, phylogenetic data analysis and linguistic data analysis. As part of future work we are interested in extending this work for a broader range of structured data (free trees, networks, directed acyclic graphs). We plan to extend our hash-based sampling methods to several other applications including anomaly detection in tree structured data. We are also interested in the applicability of these ideas for a broad class of problems drawn from the software engineering community.

References

[1] M. Daconta, L. Obrst, and K. Smith, *The Semantic Web: a guide to the future of XML, Web services, and knowledge management*. Wiley, 2003.

[2] S. Decker, S. Melnik, F. Van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks, "The semantic web: The roles of XML and RDF," *IEEE Internet computing*, vol. 4, no. 5, pp. 63–73, 2000.

[3] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan, "DBLife: A community information management platform for the database research community," *demo). In CIDR-07, 2007*.

[4] A. Brazma, H. Parkinson, U. Sarkans, M. Shojatalab, J. Vilo, N. Abeygunawardena, E. Holloway, M. Kapushesky, P. Kemmeren, G. Lara, *et al.*, "ArrayExpress–a public repository for microarray gene expression data at the EBI," *Nucleic Acids Research*, vol. 31, no. 1, p. 68, 2003.

[5] U. Westermann and W. Klas, "An analysis of XML database solutions for the management of MPEG-7 media descriptions," *ACM Computing Surveys*, vol. 35, no. 4, pp. 331–373, 2003.

[6] E. Charniak, "Tree-bank grammars," in *Proceedings of the National Conference on Artificial Intelligence*, 1996, pp. 1031–1036.

[7] M. Tsvetovat, J. Reminga, and K. Carley, *DyNetML: Interchange format for rich social network data*. Carnegie Mellon University, School of Computer Science,[Institute for Software Research International], 2004.

[8] C. Stockham, L. Wang, and T. Warnow, "Statistically based postprocessing of phylogenetic analysis by clustering," *Bioinformatics*, vol. 18, no. 3, pp. 465–469, 2002.

[9] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: a system for keyword-based search over relationaldatabases," in *Proceedings 18th International Conference on Data Engineering*, 2002, pp. 5–16.

[10] A. Elmagarmid, P. Ipeirotis, and V. Verykios, "Duplicate record detection: A survey," *IEEE Transaction on Knowledge and Data Engineering*, vol. 19, no. 1, pp. 1–16, 2007.

[11] Y. Ke, R. Sukthankar, and L. Huston, "An efficient parts-based near-duplicate and sub-image retrieval system," in *Proceedings of the 12th annual ACM international conference on Multimedia*, 2004, pp. 869–876.

[12] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *Proceedings of the international conference on Web search and web data mining*. ACM New York, NY, USA, 2008, pp. 95–106.

[13] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher, "Min-wise independent permutations (extended abstract)," in *Proceedings of the thirtieth annual ACM Symposium on Theory of Computing*, 1998, pp. 327–336.

[14] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang, "Finding interesting associations without support pruning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 1, pp. 64–78, 2001.

[15] A. Broder, S. Glassman, M. Manasse, and G. Zweig, "Syntactic clustering of the web," *Computer Networks and ISDN Systems*, vol. 29, no. 8-13, pp. 1157–1166, 1997.

[16] A. Aho, R. Sethi, and J. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986.

[17] H. Wolfson and I. Rigoutsos, "Geometric hashing: An overview," *IEEE Computational Science & Engineering*, vol. 4, no. 4, pp. 10–21, 1997.

[18] N. Augsten, M. Böhlen, and J. Gamper, "Approximate matching of hierarchical data using pq-grams," in *Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 301–312.

[19] N. Augsten, M. Bohlen, C. Dyreson, and J. Gamper, "Approximate joins for data-centric XML," in *IEEE 24th International Conference on Data Engineering, 2008. ICDE 2008*, 2008, pp. 814–823.

[20] S. Gollapudi and R. Panigrahy, "The power of two min-hashes for similarity search among hierarchical data objects," in *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2008, pp. 211–220.

[21] R. Yang, P. Kalnis, and A. Tung, "Similarity evaluation on tree-structured data," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 2005, pp. 754–765.

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

[23] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of ACM*, vol. 51, no. 1, pp. 117–122, 2008.

[24] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of 25th International Confererence on Very Large Data Bases*, 1999, pp. 518–529.

[25] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

[26] A. Kirsch and M. Mitzenmacher, "Distance-sensitive bloom filters," in *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments and the Third Workshop on Analytic Algorithmics and Combinatorics*. Society for Industrial Mathematics, 1987, p. 41.

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

[28] H. Samet, *Applications of spatial data structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[29] P. Bille, "A survey on tree edit distance and related problems," *Theoretical computer science*, vol. 337, no. 1-3, pp. 217–239, 2005.

[30] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, p. 1245, 1989.

[31] S. Guha, H. Jagadish, N. Koudas, D. Srivastava, and T. Yu, "Approximate XML joins," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 2002, pp. 287–298.

[32] M. Weis and F. Naumann, "DogmatiX tracks down duplicates in XML," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 2005, pp. 431–442.

[33] T. Shintani and M. Kitsuregawa, "Parallel mining algorithms for generalized association rules with classification hierarchy," in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 1998, pp. 25–36.

[34] S. Parthasarathy, "Efficient progressive sampling for association rules," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, 2002, pp. 354–361.

[35] T. Akutsu and M. Halldorsson, "On the approximation of largest common subtrees and largest common point sets," *Theoretical Computer Science*, vol. 233, no. 1-2, pp. 33–50, 2000.

[36] F. Rossello and G. Valiente, "An algebraic view of the relation between largest common subtrees and smallest common supertrees," *Theoretical Computer Science*, vol. 362, no. 1-3, pp. 33–53, 2006.

[37] T. Jiang, L. Wang, and K. Zhang, "Alignment of treesan alternative to tree edit," *Theoretical Computer Science*, vol. 143, no. 1, pp. 137–148, 1995.

[38] 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.

[39] D. Shapira and J. Storer, "Distance with Move operations," in *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, 2002, pp. 85–98.

[40] A. Andoni and R. Krauthgamer, "The computational hardness of estimating edit distance," in *IEEE Symposium on the Foundations of Computer Science (FOCS)*. Citeseer, 2007, pp. 724–734.