SyncSignature: A Simple, Efficient, Parallelizable Framework for Tree Similarity Joins

Nikolai Karpov
Indiana University Bloomington
Bloomington, IN
nkarpov@iu.edu

Qin Zhang
Indiana University Bloomington
Bloomington, IN
qzhangcs@indiana.edu

ABSTRACT

This paper introduces SyncSignature, the first fully parallelizable algorithmic framework for tree similarity joins under edit distance. SyncSignature makes use of implicit-synchronized signature generation schemes, which allow for an efficient and parallelizable candidate-generation procedure via hash join. Our experiments on large real-world datasets show that the proposed algorithms under the SyncSignature framework significantly outperform the state-of-the-art algorithm in the parallel computation environment. For datasets with big trees, they also exceed the state-of-the-art algorithms by a notable margin in the centralized/single-thread computation environment. To complement and guide the experimental study, we also provide a thorough theoretical analysis for all proposed signature generation schemes.

PVLDB Reference Format:
Nikolai Karpov and Qin Zhang. SyncSignature: A Simple, Efficient, Parallelizable Framework for Tree Similarity Joins. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/nkkarpov/syncsignature.

1 INTRODUCTION

In this paper we consider the problem of tree similarity joins under edit distance (or, tree similarity joins for short), where given a collection of trees $\mathcal{T} = \{T_1, \ldots, T_N\}$ and a distance threshold K, we want to find all similar pairs of trees (T_i, T_j) such that $\mathtt{TED}(T_i, T_j) \leq K$, where TED denotes the tree edit distance defined as the minimum number of node insertions, deletions, and substitutions to transfer one tree to the other. We consider rooted, ordered trees with labeled nodes. A node substitution is simply a node relabeling. When we delete a node v, we relink all the children of v to v's parent. When we insert a node v as a child of a node u, we relink a contiguous segment of u's children under v.

Tree-structured data is essential in modern data representation and analysis, and is widely used in data management (e.g., XML/JSON), code analysis and natural language processing (e.g., abstract syntax trees), bioinformatics (e.g., RNA secondary structures), and many other areas. An important primitive in these settings is to identify similar items under certain distance measurement, which is critical to various data analytics such as clustering, classification, and nearest neighbor queries. The tree edit distance is the most widely used distance function for measuring the similarity between a pair of trees.

Tree similarity joins has already attracted much attention in the past two decades [1, 2, 4, 5, 13, 15, 17, 30, 37]. Most of these works adopt the *filter-then-verify* framework: We first extract a set of candidate pairs trees such that this candidate set is a *superset* of the final output, and then verify for each pair of trees in the candidate set whether their tree edit distance is no more than the distance threshold by an exact tree edit distance computation. Different methods vary in their ways of performing the filtering and verification steps. A popular direction is to transform trees to simpler objects such as strings [1, 2, 13] and sets [4, 5, 17, 37], and then compute some distance (e.g., Euclidean distance, L_1 distance, string edit distance, etc.) on the resulting pair of strings/sets which serves as a lower bound of the tree edit distance of the original pair of trees. This lower bound can be computed much more efficiently than tree edit distance, and can be used in a pruning step to quickly filter out tree pairs that should not be in the final output. The performance of these algorithms depends on the effectiveness of the pruning steps and the time for computing the lower bound. The difficulty with this approach is that obtaining a tight lower bound via the corresponding strings or sets is difficult because much of the tree structure is lost in the transformation.

Tang et al. [30] took a partition-based approach. The idea is to build an index containing a collection of subtrees. At each step when we process a new tree T, we search in the index for the (partial) subtree rooted at *each* node of T to find other trees that share the same subtree. After that, we partition T to subtrees of balanced sizes and add all the partitions to the index. The correctness of the algorithm is guaranteed by the pigeonhole principle, that is, for any pair of trees whose tree edit distance is no more than the threshold, they share at least one common subtree, which can be guaranteed if each tree is partitioned into sufficiently many subtrees. Although various optimizations are provided to speed up the search for common subtrees, the main algorithm in [30] is an index nested loop join and the number of subtree comparisons can be massive when the number and/or size of trees is large.

The state-of-the-art algorithm by Hütter et al. [15] also uses an index nested loop join approach. In their algorithm, more structures of the trees are taken into consideration when dynamically building the index. In addition, an efficient verification scheme is presented to speed up the overall join operation.

A critical issue with the index nested loop joins is that it does not support parallel execution. Recall that in index nested loop join algorithms, we process input trees one at a time, use the index to

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

look for similar trees in the set of processed trees, and then update the index with the newly processed input tree. As a result, algorithms proposed [15, 30] are not fully scalable in multicore architecture or massive parallel computation environments. In this paper, we propose a simple, efficient, yet fully parallelizable algorithmic framework for tree similarity joins that scales much better in parallel computation environments.

Our Approach and Contributions. We propose an algorithmic framework named `SyncSignature` for tree similarity joins (Section 2). The high-level idea of our approach is to generate for each tree a set of signatures and then perform a hash join using these signatures. A crucial property of our signature-based join is that the set of signatures for each tree is generated *individually*, which makes the overall framework fully parallelizable. Different from the previous partition-based approach [30], our signature generation process is "implicitly synchronized" between different trees. Intuitively speaking, the process guarantees that if two trees share a large enough common subtree, then with a good probability there will be common signatures generated from this common subtree on both trees, even if the signature generation process on each tree is performed individually without any coordination. This implicit synchronization feature facilitates an effective candidate-producing procedure via hash join.

Under the above framework, we propose two signature generation schemes that fit tree similarity joins. Both schemes are randomized since it may *not* be possible for deterministic partition schemes to achieve effective implicit synchronization. Consequently, our algorithms may miss a small number of valid tree pairs, but with the proper choices of parameters, they can reach near-perfect accuracy in theory *and* in most datasets we studied. Through an extensive set of experiments, we show that our proposed algorithms not only exceed the state-of-the-art algorithm in the centralized model on big datasets, but that they can also be fully parallelized, giving them a significant advantage in parallel computation environments.

Other Related Work. We briefly summarize other related works.

String similarity joins. Similarity joins under edit distance has also been studied extensively on strings [3, 7, 9, 12, 16, 19, 21, 26, 34–36, 38, 39]. Among these works, the one that is most relevant to ours is [39]. To some extent, our approach can be thought of as a generalization and extension of the `MinJoin` algorithm used for string similarity joins in [39], but there are multiple challenges when applying the idea in `MinJoin` to trees. We will discuss these challenges in Section 3.1.

The computation of tree edit distance in RAM. Tree edit distance can be computed in polynomial time in the RAM model. Tai [29] proposed the first polynomial time algorithm with a running time $O(n^6)$ where n is the size of (the larger of) the two trees. Zhang and Shasha [40] improved Tai's algorithm to $O(n^4)$ by observing that not all possible subproblems in the dynamic programming need to be solved. Klein [18] further improved the result to $O(n^3 \log n)$ using *heavy path decomposition*. This line of research culminated in the algorithm by Demaine et al. [10] with a running time $O(n^3)$. Pawlik and Augsten also proposed an $O(n^3)$ algorithm named RTED [23], which demonstrates better practical performance. Later, the same authors proposed APTED [24] with the same running time but a better space usage. For the threshold version of the tree edit distance problem where we only need to decide whether the distance is at most K or larger than K, Touzet [31] proposed an algorithm with running time $O(nK^3)$. Recently, Pawlik and Augsten [25] designed an algorithm for threshold tree edit distance with improved practical performance. All these algorithms, however, cannot be applied directly to tree similarity joins without an effective candidate generation step.

Notation	Definition
$[n]$	$[n] \triangleq \{1, 2, \ldots, n\}$
K	edit distance threshold
$\mathcal{T}$	set of input trees
T_i	i-th tree in $\mathcal{T}$
N	number of input trees, i.e., $N = \lvert\mathcal{T}\rvert$
Σ	alphabet of labels of nodes in $\mathcal{T}$
Π	hash function $\Sigma^q \to (0, 1)$ for generating ranks
Γ	hash function $\Sigma^* \to \mathbb{N}$ for computing fingerprints
z	neighborhood size parameter
τ	signature similarity parameter
$N_r(v)$	neighborhood of v of radius r

Table 1: Summary of Notations

Similarity joins in the massive parallel computation model. We notice that parallel/distributed computation has already been studied for set and string similarity joins [6, 8, 27, 28, 32]. However, to the best of our knowledge, no fully parallelizable algorithm has been designed for tree similarity joins.

Preliminaries. We consider rooted, ordered, unweighted, and undirected trees with labeled nodes. For two nodes u, v in a tree T, the distance of u and v is defined to be the length of the unique path between u and v in T. For a node v in T, let $N_r(v)$ be the set of nodes in T that are within a distance of r from v, including v itself. We call $N_r(v)$ the neighborhood of v of radius r, or, the r-neighborhood of v.

We have listed a set of frequently used notations in this paper in Table 1.

2 THE JOIN FRAMEWORK

We begin by presenting our algorithmic framework SyncSignature for tree similarity joins. The framework is described in Algorithm 1.

Let us briefly describe Algorithm 1 in words. Besides the input set of trees $\mathcal{T}$ and the distance threshold K, Algorithm 1 takes two more parameters z and τ. The former determines the neighborhood size which will be used in implicitly synchronized signature generation schemes, and the later is a parameter for comparing the similarity of two set of signatures.

Our framework consists of three components.

(1) *Signature generation.* At Line 3-8, for each input tree T_i, we generate a set of signatures S_i. For each $s = (s.key, s.pos) \in S_i$, where $s.key$ is the fingerprint of the signature and $s.pos$ is the position of the signature which will be specified in concrete signature generation schemes, we add $(i, s.pos)$ to the bucket in hash table $\mathcal{D}$ indexed by $s.key$.

(2) *Join.* Line 9-15 is the hash join. To reduce the chance of mismatch, we include a sanity check of two signatures with the same key; the check rejects all pairs of signatures whose

Algorithm 1 SyncSignature($\mathcal{T}, K, z, \tau$)

Input: Collection of trees $\mathcal{T} = \{T_1, \ldots, T_n\}$, distance threshold K, neighborhood size parameter z, signature similarity parameter τ.
Output: $O = \{(T_i, T_j) \mid T_i, T_j \in \mathcal{T} : i \neq j, \mathtt{TED}(T_i, T_j) \leq K\}$

1: $O \leftarrow \emptyset, I \leftarrow \emptyset$
2: Initialize an empty hash table $\mathcal{D}$ and an empty table of counters C; generate a random hash function $\Pi : \Sigma \rightarrow (0, 1)$ where Σ is the node label universe, and a random hash function $\Gamma : \Sigma^* \rightarrow \mathbb{N}$.
3: **for each** $T_i \in \mathcal{T}$ **do**
4: $S_i \leftarrow$ Signature$(T_i, \Pi, z, \Gamma, \tau)$
5: **for each** $s \in S_i$ **do**
6: Add $(i, s.pos)$ into the bucket in $\mathcal{D}$ indexed by $s.key$
7: **end for**
8: **end for**
9: **for each** bucket of $\mathcal{D}$ **do**
10: **for each** pair (i, pos), (j, pos') in the bucket **do**
11: **if** $(\big||T_i| - |T_j|\big| \leq K) \wedge (|pos - pos'| \leq K)$ **then**
12: Increment counter $C(i, j)$
13: **end if**
14: **end for**
15: **end for**
16: **for each** (i, j) with $C(i, j) > 0$ **do**
17: **if** $C(i, j) \geq \tau$ **then**
18: $I \leftarrow I \cup \{(i, j)\}$
19: **end if**
20: **end for**
21: **for each** $(i, j) \in I$ **do**
22: **if** LowerBoundED$(T_i, T_j) \leq K$ **then**
23: **if** UpperBoundED$(T_i, T_j) \leq K$ **then**
24: $O \leftarrow O \cup \{(i, j)\}$
25: **else if** $\mathtt{TED}(T_i, T_j) \leq K$ **then**
26: $O \leftarrow O \cup \{(i, j)\}$
27: **end if**
28: **end if**
29: **end for**

associated tree sizes differ by more than K or their positions in the associated trees differ by more than K. At Line 16-20, we collect candidate pairs of trees whose common signatures are above a predetermined threshold τ.

(3) *Verification.* At Line 21-29, we perform a verification step on each pair of candidates to remove false positives. Before the exact tree edit distance computation, we apply some upper and lower bounds of the tree edit distance to filter out or early accept each candidate pair.

For LowerBoundED() at Line 22, one choice, as we select for our ball signature generation scheme (see Section 3.2), is to use the string edit distance of the preorder traversals of the two trees T_i, T_j, which is known to be at most K if $\mathtt{TED}(T_i, T_j) \leq K$ (see, e.g., [13]). For the Euler-tour embedding based signature generation scheme (see Section 3.3), we will use the *half* of the string edit distance of the Euler-tour embedding of T_i, T_j, which can also be shown to be at most K if $\mathtt{TED}(T_i, T_j) \leq K$. For UpperBoundED() at Line 23, we use the LGM upper bound introduced in [15]. Both check procedures can finish in $O(\max\{|T_i|, |T_j|\}K)$ time.

We will propose a couple of signature generation schemes for Signature() at Line 4 of Algorithm 1 in Section 3.

Grouping. In the SyncSignature framework, we use the same neighborhood size z for all trees in the dataset. This is *not* desirable if the sizes of the trees in the dataset vary significantly. One idea to resolve this issue is to first group input trees according to their sizes before feeding them into Algorithm 1.

More precisely, for each $z = 1, 2, \ldots$, we create a group of trees

$$\mathcal{T}_z = \{T_i \in \mathcal{T} \mid |T_i| \in [zK/c, (z+1)K/c + K]\},$$

where $0 < c \leq 1$ is a constant parameter. To avoid missing any valid output pairs, every two adjacent groups overlap by an interval of length K. After the grouping, we call Algorithm 1 on each $\mathcal{T}_z$, and union the outputs of all groups at the end. The final algorithm is presented in Algorithm 2.

Algorithm 2 Tree-Similarity-Joins($\mathcal{T}, K, c$)

Input: Collection of trees $\mathcal{T}$, distance threshold K, parameter $c \in (0, 1)$, signature similarity parameter τ
Output: $O = \{(T_i, T_j) \mid T_i, T_j \in \mathcal{T} : i \neq j, \mathtt{TED}(T_i, T_j) \leq K\}$

1: Let max be the smallest integer such that the size of the largest tree in $\mathcal{T}$ is no more than $(max + 1)K/c + K$.
2: **for** $z = 0, 1, \ldots, max$ **do**
3: $\mathcal{T}_z = \{T_i \in \mathcal{T} \mid |T_i| \in [zK/c, (z+1)K/c + K]\}$
4: SyncSignature$(\mathcal{T}_z, K, z, K/5)$
5: **end for**

Note that in Algorithm 2, there is only one parameter c, as z runs over $1, 2, \ldots$. When calling SyncSignature, we set $\tau = K/5$. We note in advance that $\tau = K/5$ satisfies the requirement $\tau = O(K)$ in Theorem 3.2 and Theorem 3.4 for our signature generation schemes. Both z and c control the neighbor size. We have $z = c\eta/K$, where η is the smallest tree size in the z-th group. We call c the *(neighborhood) resolution*, and will discuss how to choose a good value for c in Section 4.

In the rest of this paper, we always assume that the size of the input trees are all within a constant factor, since we perform the grouping operation before calling SyncSignature.

2.1 Parallel Implementation

We now describe the parallel implementation of SyncSignature framework (Algorithm 1). Let M be the number of cores/machines. First, the signature generation phase (Line 3-8) can be fully parallelized, since signature generation for each tree is performed independently. We only need to evenly distribute the set of trees to the M machines. To insert the signatures into the hash table, the M machines can first generate a hash table locally, and then merge the M hash tables.

For the join phase (Line 9-15), we try to distribute the workload to the M machines in the following way. Suppose there are t non-empty buckets in the hash table $\mathcal{D}$, let $a_1, a_2, \ldots, a_t$ be the number of signatures that are hashed into the t buckets, and let $b_i = \binom{a_i}{2}$ be the number of signature pairs in the i-th bucket that we need to consider. Let $b = \sum_{i \in [t]} b_i$. We try to partition the t buckets to the M machines so that each machine handles a similar number of signature pairs. More precisely, let i_1 be the smallest index such that $b_1 + \ldots + b_{i_1} \geq b/M$; we assign the first i_1 buckets to the first machine. Let i_2 be the smallest index such that $b_{i_1+1} + \ldots + b_{i_2} \geq b/M$; we assign the $(i_1 + 1)$-th to the i_2-th buckets to the second machine. And so on. At the end, the M machines merge local counters $C(i, j)$.

The amount of work at Line 16-20 is insignificant; we can just execute these lines sequentially using one machine.

The verification phase (Line 21-29) can be fully parallelized. We evenly distribute the set of candidate pairs in I to the M machines, and union all M output sets at the end. We note that the verification step in most algorithms following the filter-then-verify framework can be parallelized in the same way.

3 SIGNATURE GENERATION SCHEMES

In this section, we introduce several signature generation schemes for the `SyncSignature` framework. We first present a direct extension of the *local hash minima partition scheme* (LHM-partition) for trees; LHM-partition was originally designed for string similarity joins [39]. However, the direct extension cannot perform well in practice due to the unbalanced signature sizes. We then propose two effective signature generation schemes whose practical performance are not entirely comparable.

3.1 Local Hash Minima Partition

The direct extension of LHM-partition to tree similarity joins is as follows: For each input tree T, we create a random rank for each node of the tree. We next define for each node x its neighborhood to be all nodes in the ball of radius r centered at x. We designate a node x as *anchor* if x has the smallest rank among all nodes in the neighborhood. We then partition T at all of its anchor nodes, transforming the tree into a collection of subtrees. After the LHM-partition, we select all pairs of trees that share at least one common subtree as candidates of the join output, and conduct a verification step for all candidate pairs at the end.

Note that the partition on each tree is performed individually without any coordination. Intuitively, the anchors (i.e., local hash minima) provide an implicit coordination among all trees so that if two trees share a large common subtree, then with high probability this subtree will be cut at exactly the same set of anchors on both trees, generating a common partition. On the other hand, by the pigeonhole principle we know that if two trees have a small edit distance, then they must share a large common subtree, which gives the correctness of the algorithm.

However, there are several challenges of this direct extension.

(1) Different from the string case, the neighborhood sizes of different nodes in trees can be very different – for a node with degree $\Theta(n)$, the neighborhood size can be $\Theta(n)$ even if the radius $r = 1$! The analysis of LHM-partition naturally requires that the probabilities for nodes being anchors are similar, which necessitates that all nodes' neighborhood sizes be close.
(2) If we force the sizes of different nodes' neighborhoods to be similar, symmetry may break: it is possible for v to be w's neighbor while w is *not* v's neighbor.
(3) A single edit operation may change a large number of partitions, and consequently make the pigeonhole principle inapplicable. While in the string case, one edit can only affect at most two anchors (thus at most three substrings).

To tackle the first problem, we transform each tree to a left-child right-sibling binary tree with the degree of each node bounded by 3. More precisely, for each node v with m ordered children $v_1, \ldots, v_m$, we replace edges $(v, v_1), (v, v_2), \ldots, (v, v_m)$ with edges $(v, v_1), (v_1, v_2), \ldots, (v_{m-1}, v_m)$.

Algorithm 3 PARTITION-SIGNATURE$(T, \Pi, z, \Gamma, \cdot)$

Input: Input tree T, random hash function $\Pi : \Sigma \to (0, 1)$, neighborhood size parameter z, random hash function $\Gamma : \Sigma^* \to \mathbb{N}$
Output: A set of signatures S of T

1: Initialize $S \leftarrow \emptyset$, $A \leftarrow \emptyset$
2: **for each** $v \in T$ **do**
3: Assign rank $r_v \leftarrow \Pi(\ell_v)$ ▷ ℓ_v is the label of node v
4: **end for**
5: Convert T into a left-child right-sibling binary tree T'
6: **for each** $v \in T'$ **do**
7: Choose the minimal radius r of neighborhood v in T' such that $|N_r(v)| \in [z, 2z)$
8: **if** $r_v = \min_{x \in N_r(v)} r_x$ **then**
9: $A \leftarrow A \cup \{v\}$
10: **end if**
11: **end for**
12: **for each** $v \in A$ **do**
13: Cut the tree T' at node v; duplicate v with $\deg(v)$ copies, each connecting to one of the adjacent nodes of v in T'
14: **end for**
15: Let $C_1, \ldots, C_M$ be the set of connected components after cutting T' at nodes in A
16: **for** $i = 1, \ldots, M$ **do**
17: Let $(u_1, \ldots, u_{|C_i|})$ be the preorder traversal of C_i
18: $s_i.key \leftarrow \Gamma(\ell_{u_1} \cdots \ell_{u_{|C_i|}})$
19: $s_i.pos \leftarrow \min_{u \in C_i} \{\text{index of node } u \text{ in the preorder traversal of } T'\}$
20: **end for**
21: **return** $S = \{s_1, \ldots, s_M\}$

FACT 1 (SEE, E.G., [30]). *If we apply the left-child right-sibling binary tree transformation on each input tree, the edit distance between any pair of trees after the transformation will be increased by at most a factor of 2.*

To compensate for the factor 2 transformation distortion, we need to double the distance threshold K to retain the correctness of our algorithm. That is, we need to find pairs of binary trees (after the transformation) with edit distance at most $2K$ for verification. However, such a change will *not* affect the correctness of our algorithms. We will give a detailed explanation in Section 3.4.

The problem of asymmetry is built into the tree structure. Fortunately, we can show that asymmetry has little effect on the algorithm's soundness if the neighborhood sizes of all nodes are close, via a careful analysis which is significantly more complicated than that for the original LHM-partition on strings (see Lemma 3.1).

The third problem can also be resolved via a more careful analysis once the node degrees of the trees are bounded by 3.

Our partition-based signature generation scheme is described in Algorithm 3. At Line 2-4, we assign each node a random rank. We then convert the tree to a left-child right-sibling binary tree (Line 5), and identify all anchors based on the random ranks (Line 6-11). Finally, we partition the tree at all anchors and create a signature for each resulting connected components/subtrees (Line 12-16).

The signature of each connected component C consists of a key and a position. The key of C is simply a hash value of the preorder traversal of nodes of C, and the position of C is the minimum index of nodes in C in the preorder traversal of the input tree.

The following lemma gives a good estimate of the number of anchors that Algorithm 3 produces, and will be used in the analysis in Section 3.2. The proof of this lemma is quite technical and can

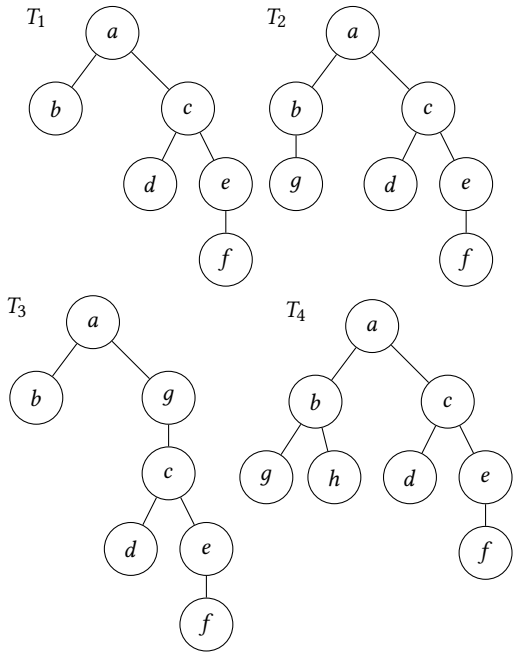

Figure 1: (Example) Input trees

tree	ball signatures
T_1	$(b, 2), (abcdef, 1), (d, 4)$
T_2	$(bg, 2), (abcdef, 1), (d, 5)$
T_3	$(b, 2), (abg, 1), (gcdef, 3), (d, 5)$
T_4	$(bgh, 2), (abcdef, 1), (d, 5)$

Table 2: Signatures produced by Partition-Signature

be skipped during the first reading. Due to space constraints, we leave it to Appendix A.1.

Lemma 3.1. *Let $n \triangleq |T|$ be the size of the tree. If $z = o(n^{1/3})$, then the number of anchors generated by Algorithm 3 falls in the range $(n/(3z), 2n/z)$ with probability $1 - o(1)$.*

An Example. Let us use an example to illustrate how Partition-Signature works.

Figure 1 describes the set of four input trees. We set the distance threshold $K = 1$, two parameters $z = 2$ and $\tau = 1$, and assume that the random hash function Π gives the following order of labels on the trees:

$$\Pi(b) < \Pi(d) < \Pi(g) < \Pi(a) < \Pi(c) < \Pi(e) < \Pi(f) < \Pi(h).$$

Given the random ordering Π, the set of anchors in T_1 is $\{b, d\}$. Since $z = 2$, the neighborhood associated with each node is simply the node itself and all of its adjacent nodes. Cutting T_1 at $\{b, d\}$ and duplicating the anchor nodes, we get three signatures $\{(b, 2), (abcdef, 1), (d, 4)\}$, where for the illustration purpose the fingerprint of each ball is simply represented as a concatenation of the labels of nodes in the ball according to the preorder traversal without applying the hash function Γ. We perform the same operation on other trees; Table 2 summarizes the tree signatures. Table 3 shows the hash table after inserting all signatures.

key	bucket contents
b	$(T_1, 2), (T_3, 2)$
d	$(T_1, 4), (T_2, 5), (T_3, 5), (T_4, 5)$
abg	$(T_3, 1)$
bgh	$(T_4, 2)$
$gcdef$	$(T_3, 3)$
$abcdef$	$(T_1, 1), (T_2, 1), (T_4, 1)$

Table 3: Hash table for Partition-Signature

A Major Issue with Partition-Signature when working with the `SyncSignature` framework. The main issue with Algorithm 3 is that it may produce subtrees with very different sizes, and many of them are small. Particularly, since at least have of the nodes are leaves, many of them would become singletons after the partition under the random ranking scheme (see, e.g., nodes b, d in Table 2). This phenomenon is very different from the string/path setting, since a path only has two leaves/end-nodes. Small partitions are more likely to get matched. Consequently, the join framework may produce many false positives and set a significant burden to verification step, making the algorithm impractical on large datasets. In our example, all 6 pairs of trees will be candidate pairs for verification.

In the next two subsections, we propose two variants of Algorithm 3 that can generate subtrees/signatures of similar sizes for a much better performance.

3.2 Signatures Using Neighborhoods

The first idea to resolve the issue mentioned above is to simply take the neighborhoods of all anchors as the subtree signatures. We call this scheme Ball-Signature, described in Algorithm 4. Compared with Algorithm 3, the main difference is that in Algorithm 4 we directly take the neighborhoods of anchors as signatures at Line 13. To further speed up the signature generation process, we first sort all nodes of the tree according to their ranks in the increasing order, and then try to find anchors starting from the node with the lowest rank. We end the process when 5τ anchors have been identified, or all anchors have been found. We note that 5τ is just used to facilitate the theoretical analysis; any larger constant than 5 will also work.

An Example. Let us again use the four input trees in Figure 1 to illustrate Algorithm 1 equipped with Ball-Signature. We again set $K = 1$, $z = 2$, and $\tau = 1$. Assume the random hash function Π gives the following order of labels on the trees:

$$\Pi(b) < \Pi(c) < \Pi(g) < \Pi(a) < \Pi(d) < \Pi(e) < \Pi(f) < \Pi(h).$$

Given this random ordering, the set of anchors in T_1 is $\{b, c\}$. The set of signatures of T_1 is $\{(ab, 1), (acde, 1)\}$, where for the illustration purpose the fingerprint of each ball is again represented as a concatenation of the labels of nodes in the ball according to the preorder traversal without applying the hash function Γ. We perform the same operations on other trees; Table 4 summarizes the tree signatures.

We next add all signatures to the hash table $\mathcal{D}$; the results are described in Table 5.

Algorithm 1 checks at Line 11 the following pairs:

$$\{(T_1, 1), (T_3, 1)\}, \{(T_1, 1), (T_2, 1)\}, \{(T_1, 1), (T_4, 1)\}, \{(T_2, 1), (T_4, 1)\}.$$

Algorithm 4 BALL-SIGNATURE$(T, \Pi, z, \Gamma, \tau)$

Input: Input tree T, random hash function $\Pi : \Sigma \to (0,1)$, neighborhood size parameter z, random hash function $\Gamma : \Sigma^* \to \mathbb{N}$, signature similarity parameter τ
Output: A set of signatures S of T

1: Initialize $S \leftarrow \emptyset$
2: **for each** $v \in T$ **do**
3: Assign rank $r_v \leftarrow \Pi(\ell_v)$
4: **end for**
5: Sort nodes in T according to their rank in the increasing order, getting $\{v_1, \ldots, v_{|T|}\}$
6: Convert T into a left-child right-sibling binary tree T'
7: **for** $v = v_1, \ldots, v_{|T|}$ **do**
8: Choose the minimal radius r of neighborhood v in T' such that $|N_r(v)| \in [z, 2z)$
9: **if** $r_v = \min_{x \in N_r(v)} r_x$ **then**
10: Let $(u_1, \ldots, u_{|N_r(v)|})$ be the preorder traversal of $N_r(v)$
11: $s.key \leftarrow \Gamma(\ell_{u_1} \cdots \ell_{u_{|N_r(v)|}})$
12: $s.pos \leftarrow \min_{u \in N_r(v)} \{\text{index of node } u \text{ in the preorder traversal of } T'\}$
13: $S \leftarrow S \cup \{s\}$
14: **end if**
15: **if** $|S| = 5\tau$ **then**
16: **return** S
17: **end if**
18: **end for**
19: **return** S

tree	ball signatures
T_1	$(ab, 1), (acde, 1)$
T_2	$(abg, 1), (acde, 1)$
T_3	$(ab, 1), (gcde, 3)$
T_4	$(abgh, 1), (acde, 1)$

Table 4: Sets of signatures produced by BALL-SIGNATURE

key	bucket contents
ab	$(T_1, 1), (T_3, 1)$
$acde$	$(T_1, 1), (T_2, 1), (T_4, 1)$
abg	$(T_2, 1)$
$gcde$	$(T_3, 3)$
$abgh$	$(T_4, 1)$

Table 5: Hash table for BALL-SIGNATURE

All pairs except $\{(T_1, 1), (T_4, 1)\}$ pass the check. The final set of candidates for verification is $\{(T_1, T_3), (T_1, T_2), (T_2, T_4)\}$. The three pairs will pass the verification step and appear in the final output because each has an edit distance of 1.

Analysis of Algorithm 4. In BALL-SIGNATURE, the neighborhoods of anchors may overlap. Compared with disjoint signatures generated from a partition scheme, overlapping signatures introduces a new challenge: a single edit may affect many signatures. For example, consider a star tree T with one central node and 100 leaf nodes. Let $K = 1$, $z = 2$, and $\tau = 1$. Under a random hash ranking, about half of the leaf nodes will be anchors, whose neighborhoods will contain the central node and itself. Consequently, a single edit on the central node (getting another tree T') will affect about half of the signatures. This could cause the similarity check at Line 17 of Algorithm 1 to fail even that $\mathrm{TED}(T, T') \leq 1$.

It appears to be impossible to prove that `SyncSignature` equipped with BALL-SIGNATURE will not miss any valid pairs under an adversarial set of K edits. However, if the K edits occur at random locations, we can still show that our join algorithm succeeds with a good probability.

THEOREM 3.2. *For binary trees T_1, T_2 with $|T_1| = n$ and $\mathrm{TED}(T_1, T_2) \leq K$, let S_1 and S_2 be the sets of signatures outputted by Algorithm 4 on T_1 and T_2 respectively. For any $z = o(\min\{n/K, n^{1/3}\})$ and $\tau = O(K)$, if the (at most) K edits take place at random nodes, then the probability that S_1 and S_2 shares at least τ common signatures is at least $1 - o(1)$. The running time of Algorithm 4 is $O(nz \log n)$.*

Proof: By Lemma 3.1, we know that with probability $1 - o(1)$, the number of anchors generated by Algorithm 4 is at least $n/(3z)$, which is at least 5τ for $\tau = O(K)$ given $z = o(n/K)$. Therefore, with probability $1 - o(1)$, Algorithm 4 returns S containing exactly 5τ signatures (early termination at Line 15-16).

We next consider the number of signatures that will be affected by K edits. Observe that on a binary tree, one edit can affect at most $O(1)$ nodes. For each signature s_i (corresponding to a subtree of size at most $2z$), let Y_i be the indicator variable for the event that at least one of the K edits affects s_i. If the K edits are randomly selected, then we have

$$\mathbf{Pr}[Y_i = 1] \leq K \cdot O(1) \cdot 2z/n = O(Kz/n).$$

Let $Y = \sum_{i=1}^{5\tau} Y_i$. Given $z = o(n/K)$, we have

$$\mathbf{E}[Y] \leq 5 \cdot O(\tau K z/n) = o(\tau).$$

By Markov's inequality we have $\mathbf{Pr}[Y \leq \tau] \geq 1 - o(1)$, and consequently $\mathbf{Pr}[5\tau - Y \geq \tau] \geq 1 - o(1)$. That means, for a pair of trees T_1, T_2 with $\mathrm{TED}(T_1, T_2) \leq K$, they will share at least τ common signatures with probability $1 - o(1)$.

For the running time, for each node we spent $O(z)$ time to construct a ball neighborhood and $O(z \log n)$ for computing the signature. The time cost of other steps is of lower-order term. Thus the total running time is $O(zn \log n)$. □

3.3 Signatures Using Euler-Tour Embedding

The second idea to resolve the issue of unbalanced signature sizes is to first embed trees to strings, use LHM-partition on strings, and then map substrings back to subtrees to produce signatures. The signature generation scheme based on Euler-tour embedding is described in Algorithm 5. At Line 1 we convert the tree to a string which is the Euler-tour of the tree. Next, we use LHM-partition to partition the string to substrings (Line 2-8). Finally, for each substring, we retrieve the corresponding subtree (to be explained in the next paragraph) and generate a signature of the subtree.

The Euler-tour translates a tree T of size n to a string of length $2n$. Each element in the string consists of the label of the corresponding tree node and a sign. Each node of the tree T appears twice in the string with different signs ('+' and '-'). The details are described in Algorithm 6. Each substring in the Euler-tour can be inversely mapped to a subtree in T as follows: we first identify the corresponding node in T for each letter in the substring, getting a set F. We then remove duplicated nodes in F (recall that each node in T corresponds to 2 letters in the Euler-tour). The set of nodes in F must be connected due to the construction of the Euler-tour. The

Algorithm 5 EULER-SIGNATURE$(T, \Pi, z, \Gamma, \cdot)$

Input: Input tree T, distance threshold K, random hash function $\Pi : \Sigma \to (0, 1)$, neighborhood size parameter z, random hash function $\Gamma : \Sigma^* \to \mathbb{N}$

Output: A set of signatures S of T

1: $E \leftarrow$ EULER-TOUR(root of T, T)
2: For i-th element y_i in E, assign a rank $r_i \leftarrow \Pi(y_i)$
3: Let I be the subset of $\{z+1, \ldots, |E|-z\}$ such that $r_i = \min_{j \in \{i-z, \ldots, i+z\}} \{r_j\}$; denote $I = (i_1, \ldots, i_k)$ with $i_1 < \ldots < i_k$
4: Initialize $P \leftarrow \{y_1 \ldots y_{i_1} - 1\}$
5: **for** $j = 1, \ldots, k-1$ **do**
6: $\quad P \leftarrow P \cup \{y_{i_j} \ldots y_{i_{j+1}} - 1\}$
7: **end for**
8: $P \leftarrow P \cup \{y_{i_k} \ldots y_{|E|}\}$
9: **for each** $p \in P$ **do**
10: $\quad$ Let Q be the corresponding subtree of p in T
11: $\quad$ Let $(u_1, \ldots, u_{|Q|})$ be the preorder traversal of Q
12: $\quad s.key \leftarrow \Gamma(\ell_{u_1} \cdots \ell_{u_{|Q|}})$
13: $\quad s.pos \leftarrow \lfloor$(index of the first element of p in E)$/2 \rfloor$
14: $\quad S \leftarrow S \cup \{s\}$
15: **end for**
16: **return** S

Algorithm 6 EULER-TOUR(v, T)

Input: Input tree T and a vertex v in T

Output: Euler-tour of the subtree rooted at v

1: **if** v is leaf **then**
2: $\quad$ **return** $\ell_v^+ \ell_v^-$
3: **else**
4: $\quad$ Let $v_1, \ldots, v_k$ be the children of v
5: $\quad$ **return** $\ell_v^+ \circ$ EULER-TOUR$(v_1, T) \circ \ldots \circ$ EULER-TOUR$(v_k, T) \circ \ell_v^-$
6: **end if**

tree	Euler-tours
T_1	$a^+b^+b^-c^+d^+d^-e^+f^+f^-e^-c^-a^-$
T_2	$a^+b^+g^+g^-b^-c^+d^+d^-e^+f^+f^-e^-c^-a^-$
T_3	$a^+b^+b^-g^+c^+d^+d^-e^+f^+f^-e^-c^-g^-a^-$
T_4	$a^+b^+g^+g^-h^+h^-b^-c^+d^+d^-e^+f^+f^-e^-c^-a^-$

Table 6: Euler-tours of T_1, T_2, T_3, and T_4.

subtree of the inverse mapping is simply the induced subtree in T of the nodes in F.

An Example. We will again use the input trees in Figure 1 as an example to illustrate Algorithm 1 equipped with EULER-SIGNATURE. We again set the distance threshold $K = 1$, and two parameters $z = 2, \tau = 1$. We assume that the random hash function Π gives the following order of labels:

$$\Pi(h^-) < \Pi(g^-) < \Pi(b^-) < \Pi(f^+) < \Pi(f^-) < \Pi(a^+) < \Pi(a^-) < \Pi(b^+) < \Pi(c^+) < \Pi(c^-) < \Pi(d^+) < \Pi(d^-) < \Pi(e^+) < \Pi(e^-) < \Pi(g^+) < \Pi(h^+).$$

The algorithm first generates the Euler-tours of the four trees, presented in Table 6. We then partition each Euler-tour string and generate signatures for each tree; see Table 7. The corresponding sets of signatures of the partitions are presented in Table 8; for simplicity, instead of mapping back to the subtrees and using the random hash function Γ, we just utilize the substrings as signature keys. The corresponding hash table is presented in Table 9.

tree	partitions of Euler-tours
T_1	$a^+b^+ \mid b^-c^+d^+d^-e^+ \mid f^+f^-e^-c^-a^-$
T_2	$a^+b^+g^+ \mid g^-b^-c^+d^+d^-e^+ \mid f^+f^-e^-c^-a^-$
T_3	$a^+b^+ \mid b^-g^+c^+d^+d^-e^+ \mid f^+f^-e^-c^-g^-a^-$
T_4	$a^+b^+g^+g^-h^+ \mid h^-b^-c^+d^+d^-e^+ \mid f^+f^-e^-c^-a^-$

Table 7: Partitions of Euler-tours.

tree	Euler signatures
T_1	$(ab, 0), (bcde, 1), (acef, 4)$
T_2	$(abg, 0), (bgcde, 2), (acef, 5)$
T_3	$(ab, 0), (bgcde, 1), (agcef, 4)$
T_4	$(abgh, 0), (bhcde, 3), (acef, 6)$

Table 8: Set of signatures produced by EULER-SIGNATURE.

key	bucket contents
ab	$(T_1, 0), (T_3, 0)$
$bcde$	$(T_1, 1)$
$acef$	$(T_1, 4), (T_2, 5), (T_4, 6)$
abg	$(T_2, 0)$
$bgcde$	$(T_2, 2), (T_3, 1)$
$agcef$	$(T_3, 4)$
$abgh$	$(T_4, 0)$
$bhcde$	$(T_4, 3)$

Table 9: Hash table for EULER-SIGNATURE.

Next, Algorithm 1 checks at Line 11 the following five pairs:

$$\{(T_1, 0), (T_3, 0)\}, \{(T_1, 4), (T_2, 5)\}, \{(T_1, 4), (T_4, 6)\}, \{(T_2, 5), (T_4, 6)\}, \{(T_2, 2), (T_3, 1)\}.$$

All pairs except $\{(T_1, 4), (T_4, 6)\}$ pass the check. The verification step will further reject the candidate pair (T_2, T_3) whose edit distance is greater than $K = 1$. The final output is again $\{(T_1, T_3), (T_1, T_2), (T_2, T_4)\}$.

Analysis of Algorithm 5. The following lemma states that Euler-tour embedding will not increase the edit distance by more than a factor of 2. Due to space constraints, the proof can be found in Section A.2.

LEMMA 3.3. *Let E_1 and E_2 be the Euler-tour embedding of trees T_1 and T_2, respectively. We have* $\mathsf{ED}(E_1, E_2) \le 2 \cdot \mathsf{TED}(T_1, T_2)$.

Now we present the main theorem for Algorithm 5.

THEOREM 3.4. *For binary trees T_1, T_2 with $|T_1| = n$ and* $\mathsf{TED}(T_1, T_2) \le K$, *let S_1 and S_2 be the sets of signatures outputted by Algorithm 5 on T_1 and T_2 respectively. For any $z = o(n/K)$ and $\tau = O(K)$, the probability that S_1 and S_2 shares at least τ common signatures is at least $1 - o(1)$. The running time of Algorithm 5 is $O(n \log n)$.*

Proof: We make use of the following lemma which states that the number of partitions created by LHM-partition is tightly concentrated to its mean.

LEMMA 3.5 ([39]). *Given an input Euler-tour E and a neighborhood size parameter z, for any $c > 0$, the size of set I ($|I| = X$) created by Algorithm 5 satisfies* $\mathbf{Pr}\left[|X - \mathbf{E}[X]| > \sqrt{c\mathbf{E}[X]}\right] < 1/c$, *where* $\mathbf{E}[X] = \frac{|E|-2z}{2z+1}$.

By Lemma 3.3, we know that after the embedding, the two resulting strings E_1 and E_2 has (string) edit distance at most $2K$.

We now consider a particular edit operation O on E_1. It is easy to see that the number of anchors that O can affect is bounded by 2. Consequently, the number of partitions O can affect is bounded by 3. Given that there are at most $2K$ edit operations, the total number of partitions of E_1 that may be affected by edit operations is bounded by $6K$. By Lemma 3.5, if $z = o(n/K)$, setting $c = n^{0.1} = \omega(1)$ and recalling that $|E_1| = 2n$ and $\tau = O(K)$, we have with probability $1 - o(1)$, there are at least

$$(1 - o(1)) \cdot \frac{|E_1| - 2z}{2z + 1} - 6K \geq 0.9 \cdot \frac{|E_1|}{2z} - 6K \geq \tau$$

unaffected partitions after the $2K$ edits.

For the running time, the Euler tour can be constructed in $O(n)$ time. The set of partitions P can be constructed in $O(n)$ time by a linear scan. Same as before, the fingerprint computation costs $O(n \log n)$ time. The time cost of other steps is of lower-order term. Thus the total running time is $O(n \log n)$. □

3.4 Plugging Signature Generation Schemes to the SyncSignature Framework

Our signature generation schemes Ball-Signature and Euler-Signature are both *Monte Carlo* randomized algorithms, which has a small error probability. More precisely, Theorem 3.2 (for Ball-Signature) and Theorem 3.4 (for Euler-Signature) state that for each pair of binary trees whose edit distance is at most K, the pair will be include in the final output with probability at least $1 - \delta$ for a $\delta = o(1)$. The standard *parallel repetition* method can further reduce the error δ all the way to zero. More precisely, we repeat the signature generation step and the hash join step for $\log N$ times (N is the number of input trees), each time using independent random ranks. We then union the $\log N$ sets of candidate pairs for verification. Now, each pair of trees with edit distance at most K will be included in the output with probability at least $1 - \delta^{\log N} \geq 1 - 1/N^{100}$. By a union bound on $O(N^2)$ pairs of trees, all pairs of trees with edit distance at most K will be included in the output with probability at least $1 - 1/N$. On the other hand, the verification phase in the SyncSignature framework guarantees that there is no false positive in the output.

By Fact 1, the binary tree transformation may increase the edit distance between a pair of trees by at most a factor of 2, to compensate which we need to find pairs of binary trees (after the transformation) with edit distance at most $2K$ for verification. However, this change will *not* affect the correctness of Theorem 3.2 (for Ball-Signature) and Theorem 3.4 (for Euler-Signature) – the statements and the proofs of the theorems still hold if we replace K with $2K$ everywhere. We note that the final verification step is performed on the *original* input trees with distance threshold K.

We have the following theorems for SyncSignature (Algorithm 1) equipped with our signature generation schemes. Recall that after grouping, we can assume that the sizes of all input trees are within a constant factor. Let n is the size of the smallest tree in the input.

Theorem 3.6. *For any $z = o(\min\{n/K, n^{1/3}\})$ and $\tau = O(K)$, equipped with Ball-Signature (with $\log N$ parallel repetition), Algorithm 1 solves the tree similarity joins exactly with probability at least $1 - 1/N$.*

Theorem 3.7. *For any $z = o(n/K)$ and $\tau = O(K)$, equipped with Euler-Signature (with $\log N$ parallel repetition), Algorithm 1 solves the tree similarity joins exactly with probability at least $1 - 1/N$.*

Performing parallel repetition will naturally increase the running time. In our experiments, we do *not* use parallel repetition, thus our algorithms may have false negatives (in theory, δ fraction of false negatives in expectation). Our experiments in Section 4 demonstrate that on the datasets we have tested, our algorithms achieve almost perfect accuracy in most cases without parallel repetition. In the worst case, our algorithms have a false negative rate of 1.3% (Figure 2 and Figure 6). We also show in Section 4.1 that three parallel repetitions decrease the false negative rate to 0%.

We would also like to comment that a small accuracy loss due to randomization is universal in big data analytics, for example, in practically all data stream algorithms [22], locality sensitive hashing for nearest neighbor search [11], large graph analysis [33], etc. For the problem of tree similarity joins, first, a small number of false negatives may not affect the downstream applications. For example, if the subsequent task is clustering and each similar pair forms an edge, a few (random) missing edges in the input graph may not have a notable impact on the quality of the clustering output. To support this statement, we have conducted some experiments on clustering (delayed to Appendix B.4 in the full version due to space constraints). Our experiments show that the false negatives of our join schemes (without parallel repetition) will only cause at most 0.02% percent of nodes to be misclustered. Second, noise in the input data may obscure the algorithm's small accuracy loss. For example, in biological applications, the data reads obtained using the most advanced SMRT technology can have up to 12-18% errors.

4 EXPERIMENTS

In this section, we present our experimental studies of the SyncSignature framework and the two signature generation schemes.

The Setup. We abbreviate Algorithm 2 using Algorithm 4 for the signature generation as BJoin, and that using Algorithm 5 as EJoin. We set the neighborhood resolution c, the only parameter of our algorithms, to be 0.3 by default. We will discuss the influence of c on the running time and accuracy of the algorithms shortly.

We compare BJoin and EJoin with the previous best algorithm for the tree similarity join [15] (denoted by TJoin). The implementation of TJoin is available online.[1] We note that the partition-based algorithm proposed in [30], as well as algorithms in earlier works [13, 20, 37], have a considerably longer running time; these algorithms cannot finish in 10 hours in all of our experiments, and do not fit our figures if we want to provide a high resolution comparison between BJoin, EJoin, and TJoin. We thus choose to leave other algorithms out of the figures, and refer readers to [15] for a comparison.

[1] The implementation can be obtained from https://frosch.cosy.sbg.ac.at/mpawlik/ted-experiments.

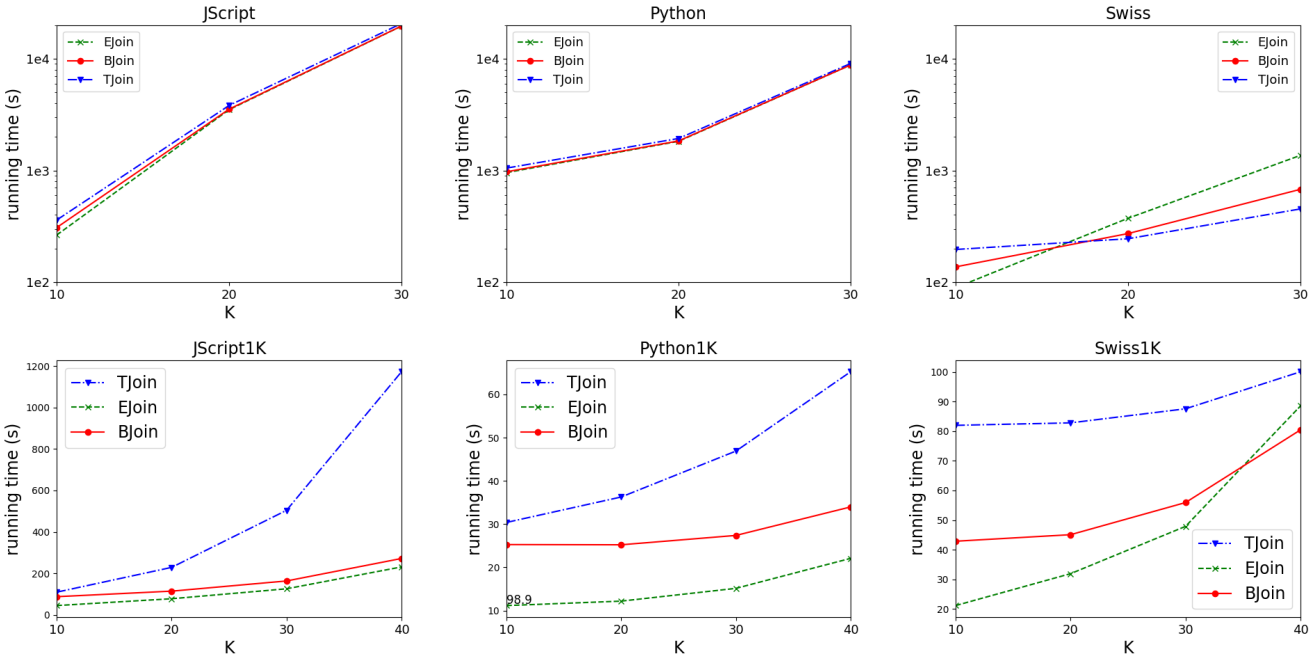

Figure 2: Running time comparisons in the single-thread setting. The number labels associated with points represent the accuracy; points without number labels have accuracy at least 99%

name	#trees	min. size	max. size	avg. size
Swiss1K	122, 772	1, 000	48, 286	1, 902
Swiss	565, 254	105	48, 286	917
Python1K	35, 958	1, 000	46, 481	3, 016
Python	148, 270	1	46, 481	948
JScript1K	40, 795	1, 000	1, 716, 813	9, 006
JScript	142, 373	4	1, 716, 813	2, 619

Table 10: Statistics of datasets.

The theoretical analysis of our algorithms necessitates a tree size of at least $\Omega(K)$. In our experiments, we test for K up to 40. Thus, for trees of sizes smaller than 100, we will use TJoin to find similar pairs. We note that a similar action was taken for the partition-based join algorithm Tang; see the discussion in the experimental section of [15].

We use three large real-world datasets: (1) Swiss[2] : protein sequence data. (2) Python[3]: abstract syntax trees of Python files. (3) JScript[4]: abstract syntax trees of JavaScript files. To better appreciate the advantage of our algorithms on large trees, we also created datasets Swiss1K, Python1K, and JScript1K, which are obtained from Swiss, Python, and JScript after filtering out trees of sizes smaller than 1000. The statistics of the datasets we use in our experiments are summarized in Table 10.

This section consists of four parts. In the first part, we compare BJoin and EJoin with TJoin in the single-thread (sequential) computation setting. In the second part, we show the influence of the tree size and the parameter c on the accuracy and running time of BJoin and EJoin. After that, we compare all algorithms in the multi-thread (parallel) computation setting. Finally, we briefly compare BJoin and EJoin. We also test how false negatives of our algorithms affect a downstream application *clustering*, but due to space constraints, we leave this part to Appendix B.4.

All algorithms are implemented in C++. All experiments are conducted on PowerEdge R740 server equipped with 2× Intel Xeon Gold 6248R 3.0GHz (24-core/48-thread per CPU) and 256GB RAM. All results are medians of 11 runs.

4.1 Comparisons in the Single-Thread Setting

We first compare BJoin and EJoin with the previous best algorithm TJoin [15] in the single-thread setting. The results are described in Figure 2.

The performance of three algorithms on full datasets (JScript, Python, and Swiss) are very similar. This is due to the fact that the number of output pairs is very large when we include all small trees. Consequently, the verification of the output pairs takes up the majority of the running time; In all plots for JScript and Python, the verification takes more than 98% of the total time. This phenomena is why we would like to test on truncated datasets to demonstrate that BJoin and EJoin outperform TJoin on large trees.

Due to the decreased output sizes, all algorithms on truncated datasets run substantially faster than that on full datasets. On all truncated datasets (JScript1K, Python1K, and Swiss1K), both BJoin and EJoin perform much better than TJoin, typically by a factor 2-4. On JScript1K and Python1K, the ratios increase when K increases, but the situation is different on Swiss1K, where the running times of BJoin and EJoin grow faster when K increases. This discrepancy could be related to the amount of output pairs produced by various datasets using different distance thresholds.

[2]https://www.uniprot.org/downloads
[3]https://files.sri.inf.ethz.ch/data/py150.tar.gz
[4]https://files.sri.inf.ethz.ch/data/js_dataset.tar.gz

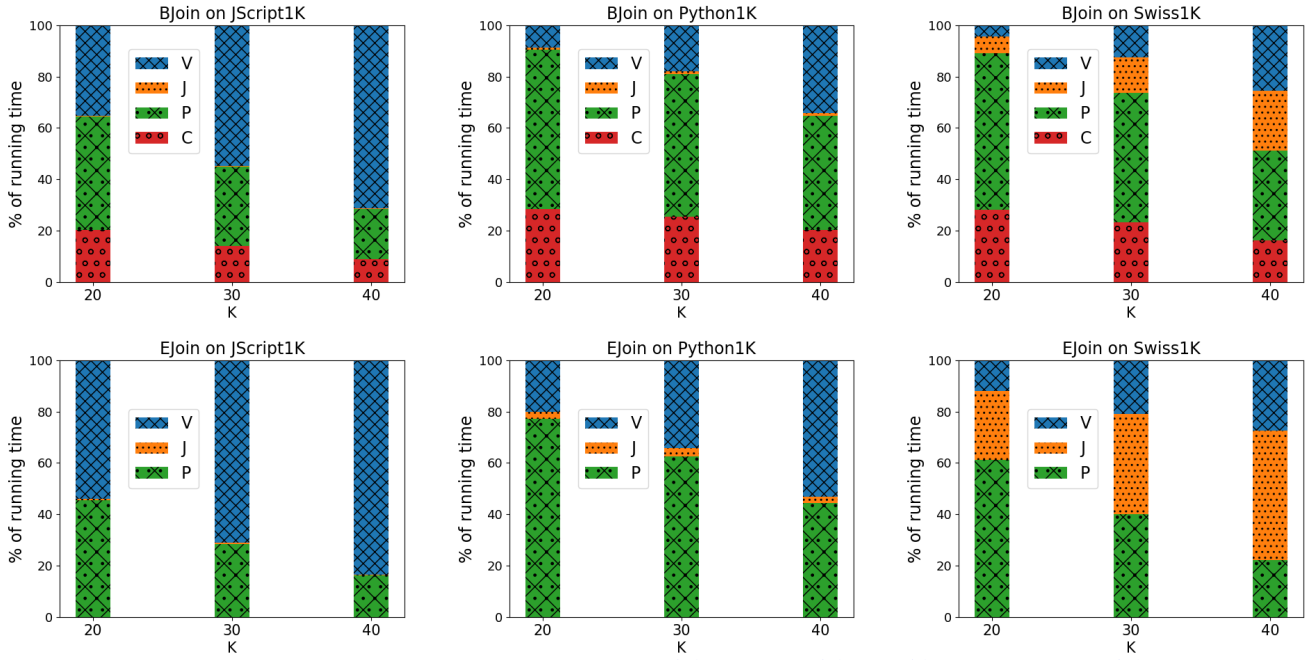

Figure 3: Running time percentages of binary-tree conversion (C), partition (P), join (J) and verification (V) of BJoin and EJoin

Overall, BJoin and EJoin have better performance than TJoin even in the single-thread setting.

The accuracy of BJoin and EJoin is almost perfect (at least 99.9%) on the JScript, JScript1K, Swiss, and Swiss1K datasets, and is at least 98.9% on Python and Python1K.

Running Time Distributions. In Figure 3, we present the running time of *binary-tree conversion*, *partition*, *join*, and *verification* of BJoin and EJoin on the three datasets. We can see that the verification step becomes more expensive when the distance threshold K increases, which is because the number of candidate pairs increases fast when K grows. On JScript and Python, the running time of the join step is negligible, while on Swiss, the join step makes a significant contribution when K is large. This may be because Swiss (biological data) is a different type of dataset compared with JScript and Python (abstract syntax trees). The binary-tree conversion step in BJoin takes a non-trivial percentage of the running time, which decreases percentage-wise as K increases. Note that EJoin does not have a binary-tree conversion step.

Parallel Repetition. Figure 4 shows the effect of parallel repetition on BJoin and EJoin. We find that repeating the signature generation and join steps twice increases the accuracy of both BJoin and EJoin to at least 99.7%, and repeating the steps three times already boosts the accuracy of both algorithms to 100%.

4.2 Influence of Tree Size and Resolution c

Influence of the Tree Size. Our next set of experiments study the influence of the tree sizes on the performance of the three algorithms. Generally speaking, our algorithms have better and more stable performance when the tree sizes are large (say, at least $50K$ where K is the distance threshold). Due to space constraints, we leave this part to Appendix B.1.

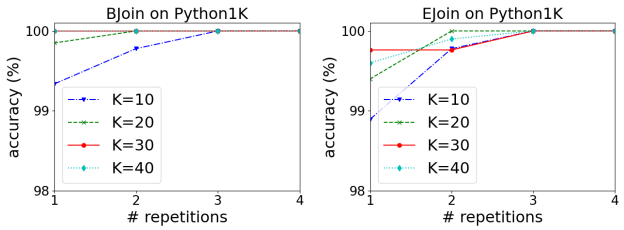

Figure 4: Accuracy Improvement using Parallel Repetition

Influence of Resolution c. Figure 5 presents the influence of neighborhood resolution c on the accuracy of BJoin and EJoin. We utilize truncated datasets rather than full datasets because the verification on the output pairs consumes the majority of the processing time on full datasets, making the influence of c harder to discern.

The experimental results are consistent with the theoretical prediction: the smaller the value of c (therefore z), the less likely we are to overlook a similar pair of trees. In Figure 5, we can see that both BJoin and EJoin achieve nearly perfect accuracy on JScript1K and Swiss1K datasets. While on Python1K, BJoin slightly outperforms EJoin, and both are above 98%.

Due to space constraints, we leave the experimental results on the influence of resolution c on the running time of the algorithms to Appendix B.2. Generally speaking, the value of c does *not* have a significant impact on the running time of BJoin and EJoin.

4.3 Comparisons in the Multi-Thread Setting

As described in Section 2, the main advantage of our SyncSignature framework is that it is fully parallelizable. We have implemented

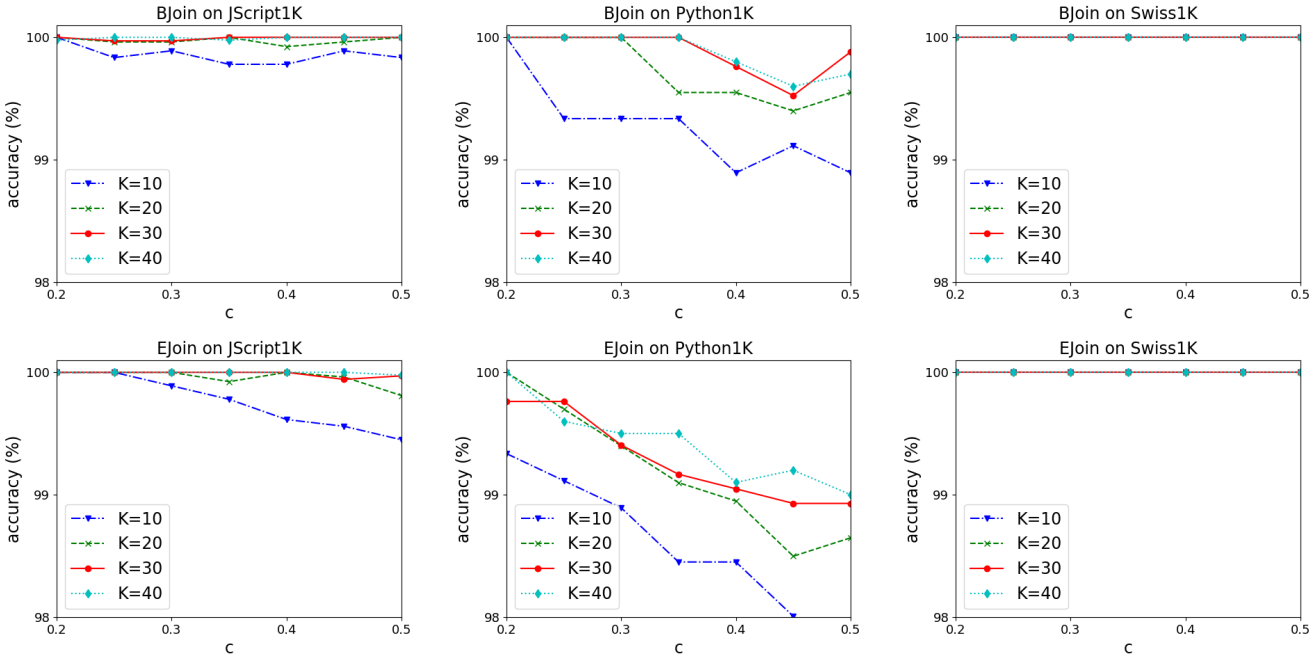

Figure 5: Influence of neighborhood resolution c on the accuracy of `EJoin` and `BJoin`

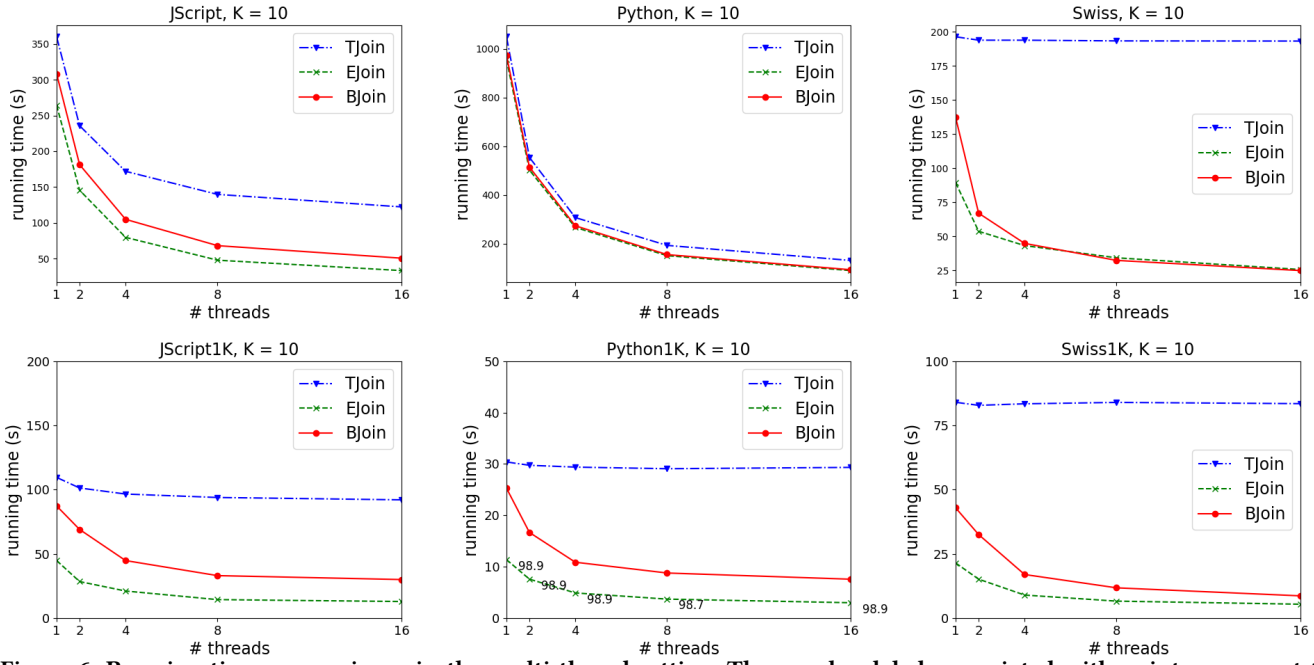

Figure 6: Running time comparisons in the multi-thread setting. The number labels associated with points represent the accuracy; points without number labels represent accuracy of at least 99%

the multi-threading version of `BJoin` and `EJoin` and compare them with `TJoin`. To be fair, we also parallelize the verification step of `TJoin`. Recall that `TJoin` uses an index nested loop join which cannot be parallelized.

The results on parallel executions of the algorithms are presented in Figure 6. We have delayed some results (for a different threshold distance K) in Figure 12 in Appendix B.3 due to space constraints. The x-axis of these figures stands for the number of threads we use in the parallel implementation. We observe that the running

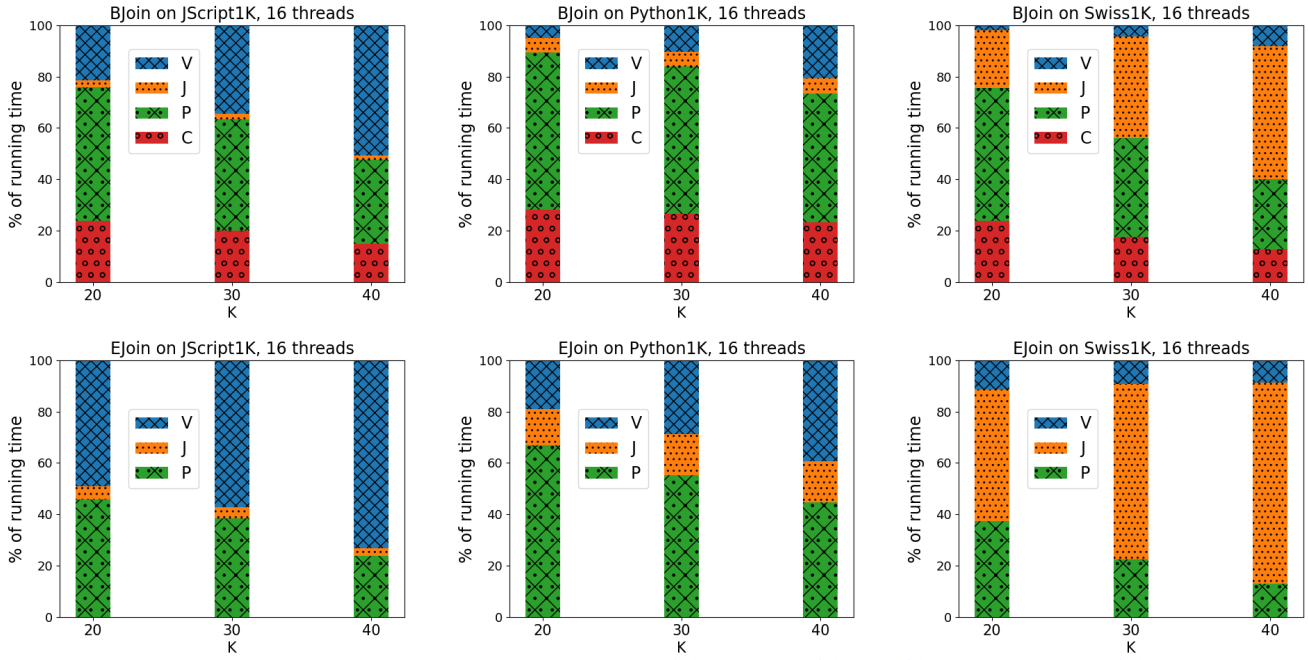

Figure 7: Running time percentages of binary-tree conversion (C), partition (P), join (J), and verification (V) in BJoin and EJoin using 16 threads

times of EJoin and BJoin decrease quickly when the number of threads increases. The trend slows when the number of threads becomes large, which is because the time spent on synchronizing/aggregating results from different threads becomes non-trivial.

On Python, since the verification step takes up the majority of the time in all algorithms, the performance of the three algorithms are close. The situation is different on JScript, on which BJoin and EJoin outperform TJoin by a notable margin. On Swiss, the advantage of BJoin and EJoin is more significant.

On truncated datasets JScript1K, Python1K, and Swiss1K, the advantage of BJoin and EJoin over TJoin becomes significant. For $K = 10$, using 16 threads, BJoin and EJoin outperform TJoin by ratios of approximately 3 and 7 on JScript1K, ratios of 4 and 10 on Python1K, and ratios of 10 and 15 on Swiss1K. We note that the curves for TJoin on Swiss is quite flat, which is due to the fact that the join step in TJoin dominates the running time, and this step cannot be parallelized.

Figure 7 present the running time of *binary-tree conversion*, *partition*, *join*, and *verification* of BJoin and EJoin on the three datasets in the multi-thread setting. The running time distribution in the multi-thread setting is similar to that in the single-thread setting (see Figure 3), except that the time percentage of join is larger in the parallel setting, which is because the hash tables generated by different threads are merged sequentially in our implementation.

4.4 A Brief Comparison of BJoin and EJoin

Finally, we give a brief comparison between BJoin and EJoin. From what we have observed, the practical performance of the two algorithms are not entirely comparable. Generally speaking, EJoin performs slightly better than BJoin in terms of the running time, while the accuracy of BJoin is slightly better than that of EJoin.

There are two reasons why the running time of BJoin could be larger than that of EJoin. First, in BJoin we need to perform an additional binary-tree conversion step that is not needed in EJoin. This conversion step has a non-trivial contribution to the total running time, as seen in Figure 3. Second, we know from Theorem 3.2 and Theorem 3.4 that that the running time of *Ball-Signature* is $O(nz \log n)$, whereas that of *Euler-Signature* is only $O(n \log n)$. The actual difference of the two signature generation schemes is not as large as z because the constants hidden in the big-O notation are different. We also observe that sometimes BJoin performs better than EJoin when K is large, which is because $z \propto 1/K$ and $O(nz \log n)$ becomes smaller when K increases.

Regarding the accuracy, it could be due to some structural information loss in EJoin when finding the anchors in the Euler-tour of each tree. Fortunately, the accuracy loss is not significant and can be offset by two or three parallel repetitions as previously discussed.

5 CONCLUSION

In this paper, we propose a synchronized signature based algorithmic framework for solving tree similarity joins under edit distance, together with two concrete signature generation schemes. Our synchronized signature based algorithms are simple, efficient, yet parallelizable. They exceed the previous best algorithms for tree similarity joins in terms of running time in the centralized/single-thread environment at a small loss of accuracy, and have a substantially larger lead in the parallel/multi-thread context.

ACKNOWLEDGMENTS

This work was supported in part by the in part by NSF CCF-1844234.

REFERENCES

[1] Tatsuya Akutsu. 2006. A relation between edit distance for ordered trees and edit distance for Euler strings. *Inf. Process. Lett.* 100, 3 (2006), 105–109.

[2] Tatsuya Akutsu, Daiji Fukagawa, and Atsuhiro Takasu. 2006. Approximating Tree Edit Distance Through String Edit Distance. In *ISAAC (Lecture Notes in Computer Science)*, Vol. 4288. Springer, 90–99.

[3] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *PVLDB*. 918–929.

[4] Nikolaus Augsten, Michael H. Böhlen, and Johann Gamper. 2005. Approximate Matching of Hierarchical Data Using pq-Grams. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 301–312.

[5] Nikolaus Augsten, Michael H. Böhlen, and Johann Gamper. 2010. The *pq*-gram distance between ordered labeled trees. *ACM Trans. Database Syst.* 35, 1 (2010), 4:1–4:36.

[6] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. 2010. Document Similarity Self-Join with MapReduce. In *ICDM*. IEEE Computer Society, 731–736.

[7] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *WWW*. 131–140.

[8] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in MaPreduce. In *SIGMOD*. ACM, 975–986.

[9] Thomas Bocek, Ela Hunt, Burkhard Stiller, and Fabio Hecht. 2007. *Fast similarity search in large dictionaries.* University.

[10] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. 2009. An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms* 6, 1 (2009), 2:1–2:19.

[11] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. [n.d.]. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.

[12] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate String Joins in a Database (Almost) for Free. In *PVLDB*. 491–500.

[13] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. [n.d.]. Approximate XML joins. In *SIGMOD*. 287–298.

[14] Erez Hartuv and Ron Shamir. 2000. A clustering algorithm based on graph connectivity. *Inf. Process. Lett.* 76, 4-6 (2000), 175–181.

[15] Thomas Hütter, Mateusz Pawlik, Robert Loschinger, and Nikolaus Augsten. 2019. Effective Filters and Linear Time Verification for Tree Similarity Joins. In *ICDE*. 854–865.

[16] Wang Jiannan, Li Guoliang, and Feng Jianhua. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. *SIGMOD* (2012), 85–96.

[17] Karin Kailing, Hans-Peter Kriegel, Stefan Schönauer, and Thomas Seidl. 2004. Efficient Similarity Search for Hierarchical Data in Large Databases. In *EDBT (Lecture Notes in Computer Science)*, Vol. 2992. Springer, 676–693.

[18] Philip N. Klein. 1998. Computing the Edit-Distance between Unrooted Ordered Trees. In *Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy, August 24-26, 1998, Proceedings (Lecture Notes in Computer Science)*, Vol. 1461. Springer, 91–102.

[19] Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *ICDE*. 257–266.

[20] Fei Li, Hongzhi Wang, Jianzhong Li, and Hong Gao. 2013. A survey on tree edit distance lower bound estimation techniques for similarity join on XML data. *SIGMOD Rec.* 42, 4 (2013), 29–39.

[21] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. PASS-JOIN: A Partition-based Method for Similarity Joins. *PVLDB* 5, 3 (2011), 253–264.

[22] S. Muthukrishnan. 2005. Data Streams: Algorithms and Applications. *Found. Trends Theor. Comput. Sci.* 1, 2 (2005).

[23] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: A Robust Algorithm for the Tree Edit Distance. *Proc. VLDB Endow.* 5, 4 (2011), 334–345.

[24] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Inf. Syst.* 56 (2016), 157–173.

[25] Mateusz Pawlik and Nikolaus Augsten. 2020. Minimal Edit-Based Diffs for Large Trees. In *CIKM*. ACM, 1225–1234.

[26] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*. 1033–1044.

[27] Chuitian Rong, Chunbin Lin, Yasin N. Silva, Jianguo Wang, Wei Lu, and Xiaoyong Du. 2017. Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics. In *ICDE*. IEEE Computer Society, 1059–1070.

[28] Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du, Yueguo Chen, and Anthony K. H. Tung. 2013. Efficient and Scalable Processing of String Similarity Join. *IEEE Trans. Knowl. Data Eng.* 25, 10 (2013), 2217–2230.

[29] Kuo-Chung Tai. 1979. The Tree-to-Tree Correction Problem. *J. ACM* 26, 3 (1979), 422–433.

[30] Yu Tang, Yilun Cai, and Nikos Mamoulis. 2015. Scaling Similarity Joins over Tree-Structured Data. *Proc. VLDB Endow.* 8, 11 (2015), 1130–1141. https://doi.org/10.14778/2809974.2809976

[31] Hélène Touzet. 2007. Comparing similar ordered trees in linear-time. *J. Discrete Algorithms* 5, 4 (2007), 696–705.

[32] Rares Vernica, Michael J. Carey, and Chen Li. 2010. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*. ACM, 495–506.

[33] Hanzhi Wang, Zhewei Wei, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2020. Exact Single-Source SimRank Computation on Large Graphs. In *SIGMOD*. ACM, 653–663.

[34] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2010. Trie-Join: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints. *PVLDB* 3, 1 (2010), 1219–1230.

[35] Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. 2013. VChunkJoin: An Efficient Algorithm for Edit Similarity Joins. *IEEE Trans. Knowl. Data Eng.* 25, 8 (2013), 1916–1929.

[36] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.

[37] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. 2005. Similarity Evaluation on Tree-structured Data. In *SIGMOD*. ACM, 754–765.

[38] Haoyu Zhang and Qin Zhang. 2017. EmbedJoin: Efficient Edit Similarity Joins via Embeddings. *KDD* (2017), 585–594.

[39] Haoyu Zhang and Qin Zhang. 2019. MinJoin: Efficient Edit Similarity Joins via Local Hash Minima. In *SIGKDD*. 1093–1103.

[40] Kaizhong Zhang and Dennis E. Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262.

A MISSING PROOFS

A.1 Proof of Lemma 3.1

Let $n = |T|$. Let X_i be the indicator variable of the event that node i is an anchor, and let $X = \sum_{i\in[n]} X_i$. The random variable X is exactly the number of anchors that will be generated by Algorithm 3. In the result of the proof, we show that X will concentrate to its mean $\mathbf{E}[X]$ with a good probability by analyzing its variance $\mathbf{Var}[X]$.

For convenience, let $B(i,z) \triangleq N_r(i)$ where r is the minimal radius such that $|N_r(i)| \in [z, 2z)$. Let $z_i = |B(i,z)|$; thus $z_i \in [z, 2z)$.

Based on the choices of random ranks, we have $\mathbf{E}[X_i] = \frac{1}{z_i}$ for each $i \in [n]$. Therefore,

$$\mathbf{E}[X] = \sum_{i\in[n]} \mathbf{E}[X_i] = \sum_{i\in[n]} \frac{1}{z_i} \in \left(\frac{n}{2z}, \frac{n}{z}\right]. \tag{1}$$

We next analyze the variance of X. By definition we have

$$\mathbf{Var}[X] = \sum_{i\in[n]} \mathbf{Var}[X_i] + \sum_{i\neq j} \mathbf{Cov}[X_i, X_j] \tag{2}$$

Since $X_i \in \{0,1\}$ for any $i \in [n]$, we have

$$\sum_{i\in[n]} \mathbf{Var}[X_i] \le \sum_{i\in[n]} \mathbf{E}[X_i^2] = \sum_{i\in[n]} \mathbf{E}[X_i] = \mathbf{E}[X] \in \left(\frac{n}{2z}, \frac{n}{z}\right]. \tag{3}$$

We next analyze $\mathbf{Cov}[X_i, X_j]$ for every pair (X_i, X_j). We do this by investigating three cases concerning the relationship between ball $B(i,z)$ and ball $B(j,z)$; see Figure 8 for an illustration.

Case I: $B(i,z)$ and $B(j,z)$ are disjoint. In this case X_i and X_j are independent. Thus, $\mathbf{Cov}[X_i, X_j] = 0$.

Case II: $B(i,z)$ and $B(j,z)$ intersect, but $i \notin B(j,z)$ and $j \notin B(i,z)$. Let $\alpha = |B(i,z)\backslash B(j,z)|$, $\beta = |B(i,z) \cap B(j,z)|$, and $\gamma = |B(j,z)\backslash B(i,z)|$.

We first analyze $\mathbf{E}[X_iX_j]$. Let Y be the indicator variable for the event that the rank of i is smaller than all γ nodes in $B(j,z)\backslash B(i,z)$.

$$\begin{aligned}
\mathbf{E}[X_iX_j] &= \mathbf{Pr}[X_i = 1] \cdot \mathbf{Pr}[X_j = 1 \mid X_i = 1] \\
&= \frac{1}{z_i} \cdot \big(\mathbf{Pr}[Y = 0 \mid X_i = 1] \cdot \mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 0] \\
&\quad + \mathbf{Pr}[Y = 1 \mid X_i = 1] \cdot \mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 1]\big) \\
&= \frac{1}{z_i} \cdot \left(\frac{\gamma}{\gamma + z_i} \cdot \frac{1}{\gamma} + \frac{z_i}{\gamma + z_i} \cdot \frac{1}{z_j}\right) \\
&= \frac{1}{\gamma + z_i} \left(\frac{1}{z_i} + \frac{1}{z_j}\right).
\end{aligned}$$

By the definition of covariance we have

$$\begin{aligned}
\mathbf{Cov}[X_i, X_j] &= \mathbf{E}[X_iX_j] - \mathbf{E}[X_i]\,\mathbf{E}[X_j] \\
&= \frac{1}{\gamma + z_i} \left(\frac{1}{z_i} + \frac{1}{z_j}\right) - \frac{1}{z_i} \cdot \frac{1}{z_j} \\
&= \frac{1}{z_i z_j} \cdot \frac{\beta}{\alpha + \beta + \gamma} \\
&\le \frac{\beta}{z^3}.
\end{aligned} \tag{4}$$

The following lemma upper bounds the sum of the sizes of the intersections between the ball centered at a fixed node i and the other $(n-1)$ balls (i.e., $\sum_{j\neq i, j\in[n]} |B(i,z) \cap B(j,z)|$). Its proof will be given shortly.

Lemma A.1. *Fix a node $i \in [n]$. For any node $j \neq i$, let $\beta_j = |B(i,z) \cap B(j,z)|$. It holds that $\sum_{j\neq i, j\in[n]} \beta_j \le 6z^4$.*

By (2), (3), (4) and Lemma A.1, we have

$$\begin{aligned}
\mathbf{Var}[X] &= \sum_{i\in[n]} \mathbf{Var}[X_i] + \sum_{i\neq j} \mathbf{Cov}[X_i, X_j] \\
&= \sum_{i\in[n]} \mathbf{Var}[X_i] + \frac{1}{2} \sum_i \sum_{j\neq i} \mathbf{Cov}[X_i, X_j]. \\
&\le \frac{n}{z} + \frac{1}{2} \cdot n \cdot \frac{6z^4}{z^3} \\
&\le 4nz.
\end{aligned} \tag{5}$$

By (1), (5) and Chebyshev's inequality, we have

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \ge c \cdot \sqrt{4nz}] \le 1/c^2,$$

for any constant c. Now if $z = o(n^{1/3})$, then

$$\mathbf{Pr}\left[X \in \left(\frac{n}{3z}, \frac{2n}{z}\right)\right] = 1 - o(1).$$

Case III: $B(i,z)$ and $B(j,z)$ intersect, $i \in B(j,z)$ but $j \notin B(i,z)$ (or symmetrically, $j \in B(i,z)$ but $i \notin B(j,z)$).

We again let $\gamma = |B(j,z)\backslash B(i,z)|$, and let Y be an indicator variable for the event that the rank of i is smaller than all γ nodes in $B(j,z)\backslash B(i,z)$. We have

$$\begin{aligned}
\mathbf{E}[X_iX_j] &= \mathbf{Pr}[X_i = 1] \cdot \mathbf{Pr}[X_j = 1 \mid X_i = 1] \\
&= \frac{1}{z_i} \cdot \big(\mathbf{Pr}[Y = 0 \mid X_i = 1] \cdot \mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 0] \\
&\quad + \mathbf{Pr}[Y = 1 \mid X_i = 1] \cdot \mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 1]\big) \\
&= \frac{1}{z_i} \cdot \left(\frac{\gamma}{\gamma + z_i} \cdot \frac{1}{\gamma} + \frac{z_i}{\gamma + z_i} \cdot 0\right) \\
&= \frac{1}{\gamma + z_i} \cdot \frac{1}{z_i},
\end{aligned}$$

which is less than that in Case II. Consequently, $\mathbf{Var}[X]$ is also smaller than that in Case II.

Lemma 3.1 follows from Case I, II, and III, and the condition that $z = o\left(n^{1/3}\right)$.

Proof of Lemma A.1. Let i be the node that we are considering. For each $u \in B(i,z)$, let $\sigma_u = |j \in [n] \mid u \in B(j,z), j \neq i|$. In words, σ_u is the number of nodes $j \in [n]$ $(j \neq i)$ whose associated ball $B(j,z)$ contains node u. It is clear that

$$\sum_{j\neq i, j\in[n]} \beta_j = \sum_{u\in B(i,z)} \sigma_u. \tag{6}$$

We next upper bound σ_u for each $u \in B(i,z)$. Let $T_u^{(z)}$ be the tree rooted at u containing all nodes j such that $u \in B(j,z)$. We thus have $\sigma_u = \left|T_u^{(z)}\right|$, and $T_u^{(z)}$ is a binary tree except that the degree of the root may be 3. See Figure 9 for an illustration. Note that the degree of u may also be 1 or 2; we use 3 here for the sake of an upper bound. Let v_1, v_2 and v_3 be the three neighbors of u. Let T_{v_1}, T_{v_2} and T_{v_3} be the (binary) subtrees of $T_u^{(z)}$ rooted at v_1, v_2 and v_3 respectively. See Figure 9 for an illustration. We have

$$\sigma_u = \left|T_u^{(z)}\right| = 1 + \left|T_{v_1}\right| + \left|T_{v_2}\right| + \left|T_{v_3}\right| \le 1 + 3 \cdot \left|T^{(z)}\right|, \tag{7}$$

where $T^{(z)}$ is a binary tree of the maximum size such that the ball $B(j,z)$ associated with each node $j \in T^{(z)}$ contains the root of $T^{(z)}$.

We next bound $\left|T^{(z)}\right|$. First, it is clear that the depth of the tree is at most z, since otherwise the ball $B(j,z)$ of some leaf j cannot

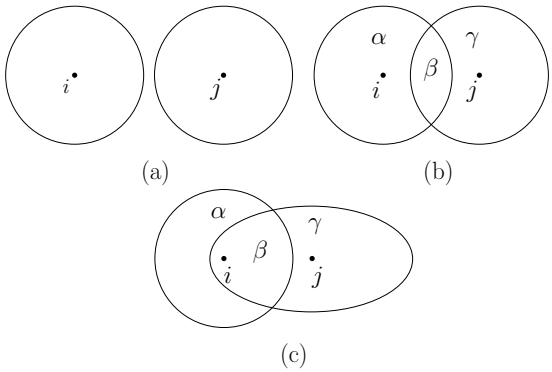

Figure 8: Relationship of two neighborhoods

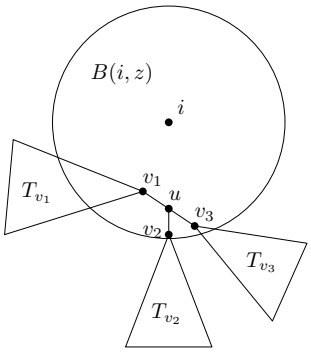

Figure 9: An illustration for the proof of Lemma A.1

reach the root. Thus if we can bound the number of the leaves of $T^{(z)}$ by b, then we have

$$\left|T^{(z)}\right| \leq b \cdot z. \tag{8}$$

Let W be the set of leaves in $T^{(z)}$ and $b = |W|$. For each leaf $w \in W$, grow a ball using BFS until the layer that contains the root; let $D(w)$ denote this ball. We thus have $D(w) \subseteq B(w, z)$, and consequently $|D(w)| \leq |B(w, z)|$. We further have

$$\sum_{w \in W} |D(w)| \leq \sum_{w \in W} |B(w, z)| \leq b \cdot z. \tag{9}$$

For a binary tree with b leaves, it is easy to see that the fully balanced binary tree (i.e., the binary tree with the layer differences among all leaves being at most one) minimizes the quantity $\sum_{w \in W} |D(w)|$. For a fully balanced binary tree (except for the last layer) with w leaves, we have

$$\sum_{w \in W} |D(w)| \geq b^{3/2}. \tag{10}$$

By (9) and (10) we have $b^{3/2} \leq b \cdot z$, which implies $b \leq z^2$. Combining (6), (7) and (8), we have

$$\sum_{j \neq i, j \in [n]} \beta_j = \sum_{u \in B(i,z)} \sigma_u \leq (2z - 1) \cdot (1 + 3bz) \leq 6z^4.$$

A.2 Proof of Lemma 3.3

We show that each edit operation on tree will not increase the string edit distance between two Euler-tour embeddings by more than 2. We consider three types of operations.

- *Substitution:* If we change the label ℓ_x of some node x, then the string obtained by Euler-tour embedding changes two elements ${\ell_x}^+$ and ${\ell_x}^-$.
- *Deletion:* If we delete node x, then the corresponding string deletes two elements ${\ell_x}^+$ and ${\ell_x}^-$.
- *Insertion:* If we insert node x and relink $v_1, \ldots, v_k$ of x's parent's children under x, then the corresponding string inserts two elements ${\ell_x}^+$ and ${\ell_x}^-$; the relink of the children nodes takes no affect.

The lemma follows directly.

B MISSING DETAILS IN SECTION 4

B.1 Influence of the Tree Size

The results are presented in Figure 10. The numbers in the x-axis represent the truncation thresholds. For example, if the number is $6e2$, it means that all trees with sizes less than 6×10^2 have been filtered.

We observe that on `JScript` and `Python`, `EJoin` and `BJoin` consistently outperform `TJoin` by a notable margin on *all* truncation thresholds. On `Swiss`, when the truncation threshold is larger than 6×10^2, `EJoin` and `BJoin` outperform `TJoin`; otherwise, our algorithms underperform. Generally speaking, our algorithms have better and more stable performance when the tree sizes are large (say, at least $50K$ where K is the distance threshold).

B.2 Influence of Resolution c on the Running Time

Figure 11 presents the influence of neighborhood resolution c on the running time of `BJoin` and `EJoin`.

Recall that in Algorithm 2, the smaller the value c, the smaller the neighborhood size we will use for the tree partitioning, resulting in a larger number of signatures generated from each tree. Consequently, the running time of the signature generation and join may increase. On the other hand, a larger number of signatures may boost the efficiency of candidate pair identifications and consequently reduce the verification time. These two opposing forces impact the total running time of `BJoin` and `EJoin`.

From Figure 11, we observe that `EJoin` outperforms `BJoin` in terms of running time on most points, with the exception of `Swiss1K`, where `EJoin`'s running time is higher when c is very small.

B.3 Additional Experiments on Multi-Thread

The running time for $K = 20$ in the multi-thread setting are described in Figure 12. We observe that the trends and relative orders of curves are the same as $K = 10$ (Figure 6) on most datasets.

Figure 13 gives the running time distributions of the three algorithms in the parallel execution using 4 threads. The patterns of the figures are very similar to Figure 7 for 16 parallel threads.

B.4 The Impact of False Negatives on the Downstream Clustering Application

Our next set of experiments study the impact of false positives introduced by our randomized algorithms on a downstream application *clustering*. In our experiments, we make use of three datasets `Python1K`, `JScript1K`, and `Swiss1K`, and test our randomized signature generation schemes `BJoin` and `EJoin`.

On each dataset and for each distance threshold $K \in \{20, 30\}$, we construct a graph G, where each node in the graph G corresponds

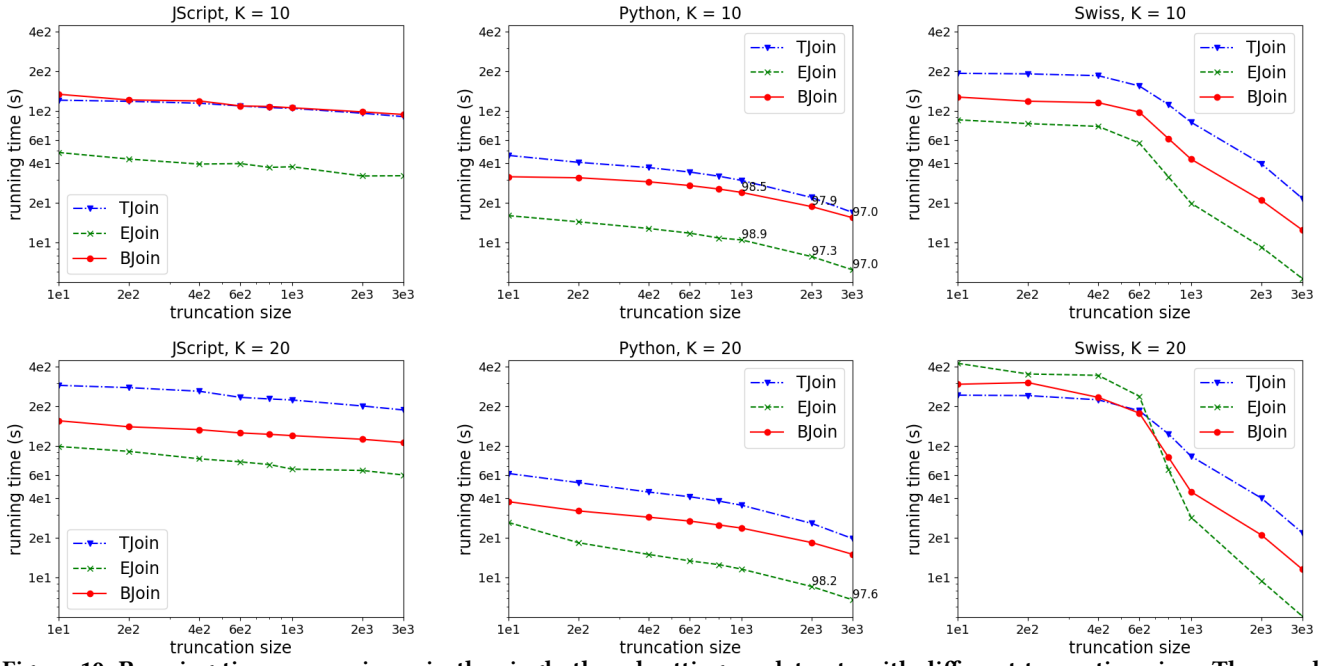

Figure 10: Running time comparisons in the single-thread setting on datasets with different truncation sizes. The number labels associated with points represent the accuracy; points without number labels have accuracy at least 99%

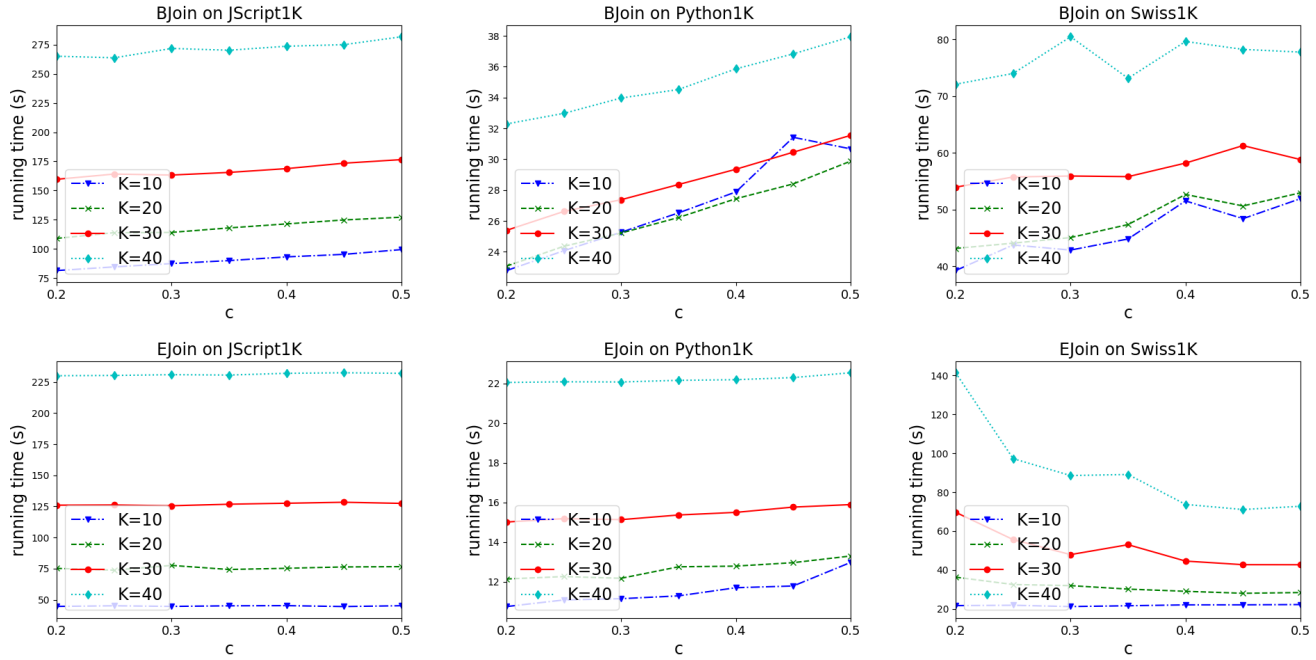

Figure 11: Influence of neighborhood resolution c on the running time of `EJoin` and `BJoin`

to a tree in the dataset, and there is an edge (x, y) if the edit distance between two trees/nodes x and y is no more than K. We construct graphs for each of the three algorithms `TJoin`, `BJoin`, and `EJoin`, and then apply the highly connected clustering algorithm from [14] on the resulting graphs. We use the clustering results on the graphs corresponding to `TJoin` as the ground truth, and calculate the percentage of misclustered nodes on graphs corresponding to `BJoin` and `TJoin`.

The experimental results are presented in Table 11. We observe that the percentage of misclustered nodes is at most 0.02% in all

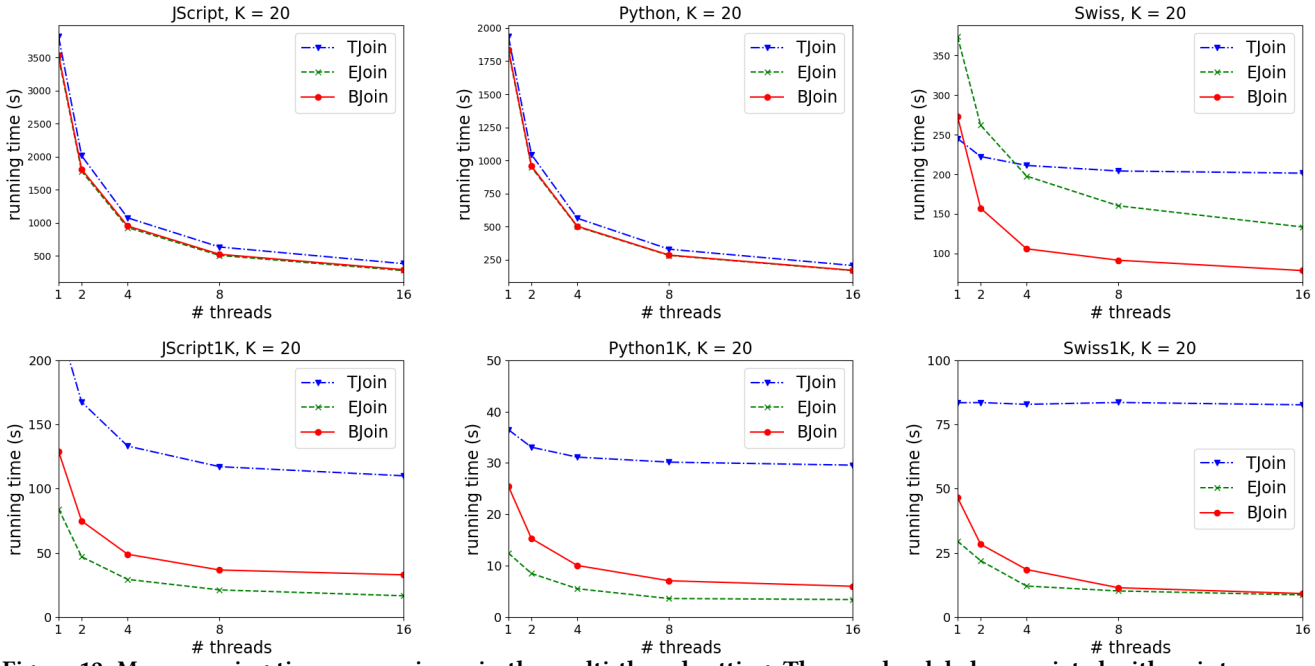

Figure 12: More running time comparisons in the multi-thread setting. The number labels associated with points represent the accuracy; points without number labels represent accuracy of at least 99%

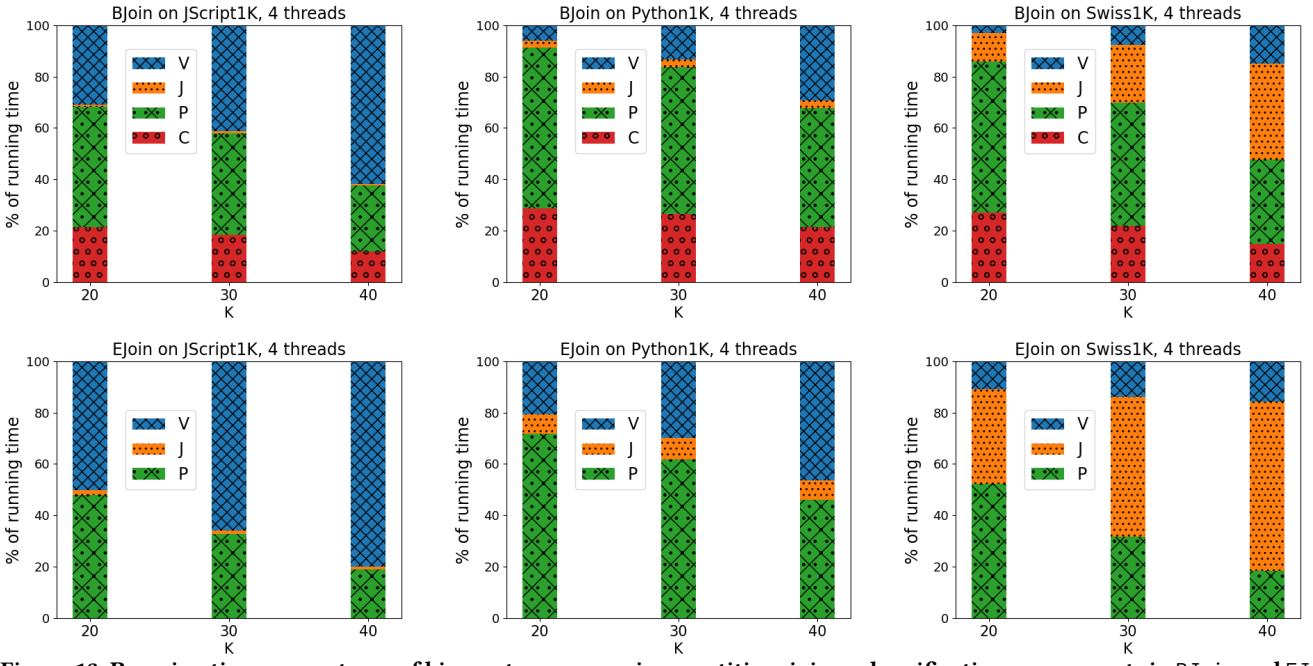

Figure 13: Running time percentages of binary-tree conversion, partition, join and verification components in `BJoin` and `EJoin` using 4 threads

Algos	Python1K		JScript1K		Swiss1K	
	K = 20	K = 30	K = 20	K = 30	K = 20	K = 30
BJoin	0.01%	0.01%	0.01%	0.01%	0%	0%
EJoin	0.02%	0.02%	0.01%	0%	0%	0%

Table 11: The percentage of misclustered elements.

experiments. This means that a small number of false negatives in the join output has negligible impact on the clustering application.