Effective Filters and Linear Time Verification for Tree Similarity Joins

Thomas Hütter, Mateusz Pawlik, Robert Löschinger, Nikolaus Augsten
Department of Computer Sciences, University of Salzburg, Austria
{thomas.huetter,mateusz.pawlik,nikolaus.augsten}@sbg.ac.at, robert.loeschinger@porscheinformatik.at

Abstract—The tree similarity join computes all similar pairs in a collection of trees. Two trees are similar if their edit distance falls within a user-defined threshold. Previous algorithms, which are based on a filter-verify approach, suffer from the following two issues. First, ineffective filters produce a large number of candidates that must be further verified. Second, the candidates are verified by computing the tree edit distance, which is cubic in the number of tree nodes. Thus, these techniques fail to scale to large tree collections and are infeasible even for small collections when the trees are large. In this paper, we present a scalable solution for the tree similarity join that is based on (1) an effective indexing technique that leverages both the labels and the structure of trees to reduce the number of candidates, (2) an efficient upper bound filter that moves many of the candidates directly to the join result without additional verification, (3) a linear time verification technique for the remaining candidates that avoids the expensive tree edit distance. We are the first to propose an efficient, non-trivial upper bound and linear time verification for the tree edit distance. Unlike previous solutions, our approach scales to collections with millions of trees. We improve the overall join time by up to two orders of magnitude w.r.t. the state of the art.

Index Terms—tree similarity join, tree edit distance

I. INTRODUCTION

Often data are naturally represented as trees, witnessed by the popularity of hierarchical data formats like XML or JSON. Trees store data items or metadata in the node labels and encode their relation in the structure. Exact matches are often ineffective for trees due to small differences in structure or labels, thus small errors should be tolerated. The classical error measure for trees is the edit distance [1], which counts the minimum number of node edits (insertion, deletion, rename) that transform one tree into the other.

In this paper, we study the tree similarity join, which identifies all tree pairs in a collection of trees \( \mathcal{T} \) that are within a user-defined edit distance threshold \( \tau \). This problem is of high interest for various applications, e.g., data integration and duplicate detection [2]–[5], code analysis using abstract syntax trees [6], [7], the analysis of merger trees in astrophysics [8], and the RNA secondary structures in biology [9], [10], or pairwise clustering of shock graphs in shape recognition [11].

The problem is challenging for two reasons. First, standard techniques to avoid the infeasible nested loop join (like sorting or hashing) fail for trees. New filter techniques must be developed to produce a so-called candidate set, which should be a small superset of the join result. Second, verifying the candidates is expensive. The fastest tree edit distance algorithm [12] runs in cubic time and quadratic space in the number of tree nodes. This is a serious limitation since all candidates must undergo verification, i.e., the tree edit distance is not only computed for the false positive candidates (that are rejected) but also for the true positives. When the trees are large (e.g., up to 48k nodes in the Swissprot datasets), verifying even a single candidate pair is a challenge.

The fastest solutions for tree similarity joins [2], [13]–[15] are based on a filter-verify approach: the filter phase returns a set of candidates, and the candidates must be verified using the tree edit distance. While such an approach substantially reduces the number of edit distance computations w.r.t. the nested loop join, there are still too many candidates for large datasets like DBLP with millions of trees. In fact, none of the previous works joins large real-world datasets, e.g., in the most recent work by Tang et al. [14] only a small sample of DLBP with 100k trees was used. Moreover, the problem of large, individual trees is not addressed by previous works.

We propose a new tree similarity join algorithm that scales to both large trees and datasets with millions of trees.

- Our novel candidate index produces a small candidate set and is up to two orders of magnitude faster than previous works. Only \( O(|\mathcal{T}| \tau) \) space is required to index \( |\mathcal{T}| \) trees, i.e., we can index trees of arbitrary size in main memory.
- The upper bound filter is executed on all candidates and identifies result pairs; these pairs need no further verification. This is the first non-trivial upper bound that runs in linear \( O(n \tau^2) \) time for trees with \( n \) nodes.
- Verification: Previous algorithms compute the edit distance \( D \) and then check the join condition \( D \leq \tau \) to verify a candidate pair. We avoid the cubic edit distance and leverage the so-called bounded structural modification distance [16] to verify a tree pair in \( O(n \tau^2) \) time.
- An empirical evaluation on four real-world datasets suggests that our technique outperforms its competitors in terms of index lookup time, number of candidates, and overall runtime, often by orders of magnitude.

The paper is organized as follows. After introducing the problem in Section II, we outline our join algorithm in Section III. In Sections IV, V, and VI we discuss candidate index, upper bound, and efficient verification, respectively. The related work is discussed in Section VII. We empirically evaluate our solution in Section VIII and conclude in Section IX.
II. Preliminaries and Problem Definition

Basic definitions. A tree $T$ is a directed, acyclic, connected graph with nodes $N(T)$ and edges $E(T) \subseteq N(T) \times N(T)$, where each node has at most one incoming edge. The size $|T|$ of tree $T$ is the number of its nodes. In an edge $(v, w)$, $v$ is the parent and $w$ the child, $p(w) = v$; children of the same node are siblings. Node $x$ is an ancestor of node $v$ iff $x = p(v)$ or $x$ is an ancestor of $p(v)$; $x$ is a descendant of $v$ iff $v$ is an ancestor of $x$. The trees are rooted (the node with no incoming edge is the root node) and ordered (the sibling order matters).

Each node $v$ has a label, $lbl(v)$, that carries the data of the node. Labels are not necessarily unique. The postorder id of node $v$, $post(v)$, is its postorder position in the tree (zero-based numbering). $v$ is to the left of $w$ if $post(v) < post(w)$ and $v$ is not a descendant of $w$; $v$ is to the right of $w$ if $post(v) > post(w)$ and $v$ is not an ancestor of $w$.

In our examples, we represent a node $v$ by its label and its postorder id: $lbl(v), post(v)$. Consider, for example, tree $T_5$ in Figure 1: nodes $a_5$ (root node) and $a_6$ have identical labels; $a_6$ is to the left of $c_2$, $d_4$ is to the right of $c_2$.

Tree Edit Distance (TED). The edit distance, $\delta(T, T')$, between two trees, $T$ and $T'$, is the minimum number of node edit operations that transform $T$ to $T'$ [1], [17]. Allowable node edit operations are: delete node $v$ and connect all its children to the parent of $v$ maintaining the sibling order; insert a new node $w$ between an existing node $v$ and a consecutive subsequence of $v$’s children, rename the label of node $v$.

Edit Mapping. An editing mapping, $M$, between the nodes of two trees $T$ and $T'$, represents the edit operations that transform $T$ into $T'$. Node pairs that are mapped represent rename operations, unmapped nodes in $T$ are deleted, unmapped nodes in $T'$ are inserted. Nodes with identical label are renamed at zero cost, all other operations have cost 1. The cost of an edit mapping, $cost(M)$, is the cost sum of all its operations. For example, $cost(M_{T_5,T_6}) = 3$ for edit mapping $M_{T_5,T_6} = \{(a_0,a_0), (d_1,d_1), (b_5,c_2), (d_4,d_4), (a_5,a_5)\}$ between $T_5$ and $T_6$ in Figure 1: rename $b_5$ to $c_2$, delete $c_2$, insert $c_3$.

Definition 1. Edit mapping. A mapping $M \subseteq N(T) \times N(T')$ is an edit mapping from $T$ to $T'$ if for any two node pairs $(v, w), (v', w') \in M$ the following conditions hold:

- $v = v'$ iff $w = w'$ (one-to-one),
- $v$ is an ancestor of $v'$ iff $w$ is an ancestor of $w'$ (ancestor),
- $v$ is to the left of $v'$ iff $w$ is to the left of $w'$ (order).

The cost of the optimal (minimum-cost) edit mapping between $T$ and $T'$ is their edit distance [17], e.g., $M_{T_5,T_6}$ is optimal, thus $\delta(T_5, T_6) = cost(M_{T_5,T_6}) = 3$ (Fig. 1).

Definition 2. TED Join. Given a collection of trees $\mathcal{T}$ and a distance threshold $\tau$. The TED join is defined as the set of all distinct tree pairs in $\mathcal{T}$ that are within edit distance $\tau$: $\mathcal{T} \bowtie \mathcal{T} = \{(T, T') \in \mathcal{T} \times \mathcal{T} : T \neq T' \land \delta(T, T') \leq \tau\}$.

Problem definition. The goal of this work is an efficient solution for the TED join problem that scales to both large trees and large collections of trees.

III. Solution Overview

Our TED join algorithm (Algorithm 1) is based on three new techniques that we introduce in this paper: the candidate index, the LGM upper bound filter, and efficient BSM verification.

Algorithm 1: TJoin($\mathcal{T}, \tau$)

```
input : tree collection $\mathcal{T}$, distance threshold $\tau$
output: join result $R = \mathcal{T} \bowtie \mathcal{T}$
1 $R \leftarrow \emptyset$
2 $I \leftarrow \emptyset$
3 foreach $T \in \mathcal{T}$ (in non-decreasing size order of $T$) do
   4   foreach $T' \in I$ (LookUp($T, \tau$)) do
      5      if $LGM(T, T', \tau) \leq \tau$ then $R \leftarrow R \cup \{(T, T')\}$
      6      else if $BSM(T, T', \tau) \leq \tau$ then $R \leftarrow R \cup \{(T, T')\}$
   7   add $T$ to candidate index $I$
8 return $R$
```

We traverse the trees of the input collection $\mathcal{T}$ in non-decreasing size order and perform an indexed nested loop join. A lookup of tree $T$ in the candidate index $I$ (line 4) returns a set of candidates for $T$. The candidates include all trees in the index, $T' \in I$, such that $\delta(T, T') \leq \tau$; in addition, the candidates may include trees $T''$ with $\delta(T, T'') > \tau$, called false positives. Initially, the index is empty. After the lookup, $T$ is added to the index (line 7). The index increases only by $O(\tau)$ when a new tree is inserted, thus the overall index size does not depend on the size of the indexed trees. Since we traverse and index the trees in increasing size order, $|T'| \leq |T|$ for all candidates $T'$ of $T$.

The candidates may include false positives, thus all candidates must be verified. We verify candidates in two steps.

First, we do an inexpensive check ($O(n \tau)$ time for trees of size $n$) using our LGM upper bound (line 5): If the upper bound distance between $T$ and a candidate $T'$ is within $\tau$, then it is safe to add the pair $(T, T')$ to the join result and no further verification is required. In our experiments we show that our LGM upper bound is highly effective and, for many datasets, can identify a large portion of the result pairs.

Second, we verify the remaining candidates (line 6). All previous solutions verify a tree pair by computing the edit distance, which requires $O(n^3)$ time and $O(n^2)$ space. We show that this not necessary. Our efficient verification is based on the bounded structural modifications (BSM) distance [16], which efficiently computes the edit distance if an upper bound on the number of insertions and deletions in the optimal edit mapping is known. We show how to leverage BSM to verify a candidate pair in $O(n \tau^2)$ time and $O(n \tau)$ space.

Due to space constraints, we omit the proofs for lemmas and theorems in the following sections.

IV. Candidate Index

We present our candidate index, which is based on (1) the structural filter, a novel lower bound that considers both node labels and tree structure, and (2) tree prefixes that allow us
Lemma 1. The size difference between trees $T$ and $T'$ is a lower bound for the edit distance: $|T| - |T'| \leq \delta(T, T')$. [18]

The size difference is a loose bound because it disregards both node labels and tree structure. The tighter label lower bound [18] takes into account the number of identical labels between two trees. Let $L(T)$ be the bag of all node labels in tree $T$, and $L(T) \cap L(T')$ the intersection of two node label bags. Intuitively, the labels in the intersection represent node pairs mapped at cost 0, while labels of the larger tree that are not in the intersection are either renamed or deleted at cost 1.

Lemma 2. The label lower bound between trees $T$ and $T'$ with labels $L(T)$, $L(T')$ is defined as: $\max(|T|,|T'|) - |L(T) \cap L(T')| \leq \delta(T, T')$. [18]

Example 1. Consider the trees $T_2$, $T_3$, and $T_5$ in Figure 1 with label bags $L(T_2) = \{c, b, d, a\}$, $L(T_3) = \{c, b, d, a, a\}$, and $L(T_5) = \{c, b, d, a, a\}$. For $T_2$ and $T_3$, the size difference is 2, while the label lower bound is 3, which matches the edit distance $\delta(T_2, T_3) = 3$. For $T_3$ and $T_5$, both the size difference and the label lower bound are 0, while $\delta(T_3, T_5) = 5$.

The label lower bound maps node pairs with identical labels, regardless of their position in a tree. We next discuss how the edit mapping conditions restrict the node pairs that can exist in a mapping $M$ with a cost$(M) \leq \tau$. A node $v$ partitions a tree into four mapping regions.

Definition 3. Mapping regions. Given a node $v$ in tree $T$, we define the following node sets:

- $a(T, v)$: set of $v$’s ancestors in tree $T$;
- $d(T, v)$: set of $v$’s descendents in tree $T$;
- $l(T, v)$: set of all nodes in tree $T$ to the left of node $v$;
- $r(T, v)$: set of all nodes in tree $T$ to the right of node $v$.

Figure 2 illustrates the mapping regions w.r.t. to the node pair $(v, w)$ in a mapping $M$. According to the edit mapping conditions (cf. Definition 1), all other nodes in $T$ can only be mapped to the nodes in $T'$ that are in the corresponding mapping regions.

Lemma 3. Given a node pair $(v, w)$ in an edit mapping $M$, any other pair in $M$ is in exactly one of the following products of mapping regions: $a(T, v) \times a(T', w)$, $d(T, v) \times d(T', w)$, $l(T, v) \times l(T', w) - r(T, v) \times r(T', w)$.

Mapping a node pair imposes restrictions on the unmapped nodes. Thanks to Lemma 3, we can estimate a lower bound on the mapping cost when a specific node pair is mapped. If the lower bound is within $\tau$, the node pair is called $\tau$-valid.

Definition 4. A node pair $(v, w) \in N(T) \times N(T')$ is $\tau$-valid if

$$\|a(T, v)\| - |a(T', w)| + \|d(T, v)\| - |d(T', w)| + \|l(T, v)\| - |l(T', w)| + \|r(T, v)\| - |r(T', w)| \leq \tau.$$ 

A $\tau$-valid node pair $(v, w)$ with $\text{lbl}(v) = \text{lbl}(w)$ is a $\tau$-valid label match.

We are only interested in edit mappings with a cost that does not exceed the distance threshold $\tau$. Such mappings can only be composed of $\tau$-valid node pairs.

Lemma 4. If $(v, w) \in M$ is not $\tau$-valid, then cost$(M) > \tau$.

We introduce the $\tau$-valid label intersection, which allows only $\tau$-valid label matches.

Definition 5. Given trees $T$, $T'$, and a distance threshold $\tau$. The $\tau$-valid label intersection is defined as $|N(T) \cap N(T')| = \max(|P|)$, where $P$ is a one-to-one mapping between nodes $N(T)$ and $N(T')$ such that $\forall (v, w) \in P : (v, w)$ is a $\tau$-valid label match.

We define the structural filter by substituting the label bag intersection in Lemma 2 with the $\tau$-valid label intersection.
Theorem 1. The structural filter between trees \( T \) and \( T' \) is a lower bound for the edit distance:
\[
\max(|T|,|T'|) - |N(T) \cap N(T')| \leq \delta(T,T')
\]

**Example 2.** Consider trees \( T_3, T_4 \) in Figure 1, \( N(T_3) = \{c_0, b_4, d_3, d_5, a_1, a_2\}, N(T_4) = \{c_2, b_3, d_1, d_4, a_0, a_5\}, \tau = 2. \)
The label lower bound is 0, since it does not consider the tree structure. The structural filter achieves a tighter bound of 4 ((\( b_4, b_3 \)) and (\( a_1, a_0 \)) are mapped), since it only considers \( \tau \)-valid label matches. The edit distance is \( \delta(T_3, T_4) = 5 \).

**B. Tree Prefix**

We introduce the concept of tree prefixes, which are used in our candidate index to reduce the number of tree pairs that must be considered. The frequency ordering orders the labels by increasing frequency of their appearance in the tree collection; ties are broken in an arbitrary but consistent way to get a strict ordering (e.g., \( c < c < b < d < a \) in Figure 1). Let \( v_1, v_2, \ldots, v^{\tau} \) be the nodes of tree \( T \) lexicographically sorted by labels and increasing postorder ids of the nodes.

The \( \tau \)-prefix of \( T \), \( \text{prefix}(T, \tau) \), consists of the nodes in the shortest sequence \( v_1, v_2, \ldots, v^\tau \) such that \( \tau \geq \tau +1 \) and \( \text{lbl}(v^i) < \text{lbl}(v^{i+1}) \) \( (i = n \text{ if } \text{lbl}(v^{\tau+1}) = \text{lbl}(v^n)) \), i.e., the prefix includes all nodes of some label \( l \), or none.

**Theorem 2.** Given trees \( T, T' \), and a distance threshold \( \tau \). If \( \text{prefix}(T, \tau) \subseteq \text{prefix}(T', \tau) \), then \( \delta(T,T') > \tau \).

**C. Building the Index**

The candidate index \( I \) maintains a set of inverted lists \( \mathcal{L} \). A list maps a label \( l \) to all indexed trees that have that label in the prefix. For each tree, in turn, we store a postorder-sorted list of all the prefix nodes with label \( l \). Figure 1 shows the inverted lists for the example trees \( T_1 \) to \( T_5 \) and \( \tau = 2 \) after all nodes in the prefixes are inserted. A new tree is added by appending its prefix nodes to the respective lists. Since we process the trees in size order, the lists are sorted by tree size.

Thanks to Theorem 2, we only need to index the prefix of a tree \( T \). The prefix may be larger than \( \tau +1 \) if the last label in the prefix appears multiple times in \( T \), since the prefix is extended to include all nodes with that label. This is necessary because we consider only \( \tau \)-valid node pairs: Let \( v^i \in T \) and \( w^j \in T' \) be nodes with label \( l \) such that \( i, j \leq \tau +1 < k \). Assume that \( (v^i, w^j) \) is \( \tau \)-valid, whereas \( (v^i, w^j) \) is not. By limiting the prefix length to \( \tau +1 \) we miss the \( \tau \)-valid node pair \( (v^i, w^j) \).

To avoid indexing more than \( \tau +1 \) nodes of a tree \( T \) when \( \text{prefix}(T, \tau) > \tau +1 \), we can change the index as follows. For the last label \( l \) in the prefix we index only one node (instead of all nodes with that label). Together with this node we store the postorder range of the smallest and the largest node in \( T \) with label \( l \). The nodes of two trees match if their ranges overlap or the gap between the ranges is smaller than \( \tau \) (since in this case a \( \tau \)-valid label match cannot be excluded). Thus, the index size is \( O(|T| \tau) \) and does not depend on the tree size. To simplify the discussion, in the remainder of this paper we assume that we index the full prefix.

**Algorithm 2: **LookUp\((T, \tau)\)

**input:** \( T \) probing tree, \( \tau \) distance threshold
**output:** \( C \) candidate trees for tree \( T \)
1. \( ov \) : number of \( \tau \)-valid label matches per pre-candidate
2. \( C \) : empty set of candidates
3. if \( |T| \leq \tau \) then foreach \( T' \) in index \( I \) do \( ov[T'] \) ++
4. foreach distinct label \( l \in \text{prefix}(T, \tau) \) do
5. foreach \( (T'', \text{nodes}) \in \mathcal{L}[l] \) do
6. if \( |T''| < |T| - \tau \) then remove tree \( T'' \) from \( \mathcal{L}[l] \)
7. else \( ov[T'] \leftarrow ov[T'] + \)
8. StructuralMapping\((\text{nodes}(T, l), \text{nodes}', \tau)\)
9. if \( ov[T'] > 0 \) and StructuralFilter\((T, T', ov[T'], \tau)\)
10. \( \mathcal{C} \leftarrow \mathcal{C} \cup \{T'\} \)
11. return \( \mathcal{C} \)

**Algorithm 3: **StructuralMapping\((\text{nodes}, \text{nodes}', \tau)\)

**input:** \( \text{nodes}, \text{nodes}' \) duplicate sets, \( \tau \) dist. threshold
**output:** \( \mathcal{C} \) : empty set of candidates
1. \( c \leftarrow 0 \)
2. \( d \leftarrow \text{smaller/larger set of nodes/nodes}' \)
3. foreach \( v \in d \) do
4. foreach \( w \in d_L : \tau \)-valid window do
5. if \( (v, w) \) is \( \tau \)-valid then \( c \leftarrow c + 1 \)
6. return \( c \)

**D. Index Lookup**

The index lookup proceeds in two steps. In the first step, we retrieve pre-candidates: tree pairs that have at least one \( \tau \)-valid label match in the prefix. In the second step, we prune pre-candidates using the structural filter; pre-candidates that pass the structural filter are candidates.

**Running example.** We do a lookup of example tree \( T_3 \) (cf. Figure 1), \( \tau = 3 \). \( T_1 \) and \( T_2 \) are already in the index, the inverted lists are \( \mathcal{L} = \{a \rightarrow (T_1[a_0], T_2[a_1]), b \rightarrow (T_1[b_1], T_2[b_3]), d \rightarrow (T_2[d_2]), e \rightarrow (T_2[e_0])\} \). The join result so far is \( R = \{(T_1, T_2)\} \), since \( \delta(T_1, T_2) = 2 < \tau \).

**Pre-candidate generation.** Algorithm 2 retrieves pre-candidates for a probing tree \( T \) and threshold \( \tau \). A tree \( T' \) in the index is a pre-candidate if the prefixes of \( T \) and \( T' \) have at least one \( \tau \)-valid label match and the size lower bound holds.

We maintain an overlap counter, \( ov[T'] \), for candidate tree \( T' \) to count the number of \( \tau \)-valid label matches with probing tree \( T \). This is useful to (partially) skip the prefix part of the trees during the evaluation of the structural filter.

Algorithm 2 proceeds as follows. (a) Deal with small probing trees (line 3). (b) For each label \( l \) in the prefix of \( T \), retrieve and process the inverted list of \( l \) (lines 4-5). (c) Remove candidates from the inverted lists that are too small (size difference, line 6). (d) Compute the \( \tau \)-valid label intersection (structural mapping, Algorithm 3) between the
nodes in the prefix of \( T \) and candidates \( T' \) from the inverted list and update the overlap counter \( ov[T'] \) (line 7). (e) Apply the structural filter to each pre-candidate pair (lines 8-10).

a) **Handling small trees** (Algorithm 2, line 3). If the size of the probing tree is \( |T| < \tau \), a tree \( |T'| \leq |T| \) may satisfy the threshold without having a \( \tau \)-valid label match. Since the trees are processed and indexed in increasing size order, all trees in the index are pre-candidates for \( T \). In our example, no pre-candidates are added, since \( |T_3| > \tau \).

b) **Lookup inverted lists** \( L \) (Algorithm 2, lines 4-5). In order to ensure that no \( \tau \)-valid label matches are missed, all nodes with identical labels must be handled together. With nodes \((T, l)\) we denote the set of all nodes of tree \( T \) with label \( l \). For each label \( l \), we look up the inverted list \( L[l] \) to retrieve the list of all trees that have at least one node with label \( l \) in the prefix. For each tree \( T' \) we get a list nodes' of nodes with label \( l \), sorted by postorder id. The nodes in nodes\((T, l)\) are sorted by postorder id as well. In our example, we lookup labels \( c \) (no entry), \( b \) (returns \( T_1[b_1], T_2[b_3] \)), and \( d \) (returns \( T_2[d_2] \)).

c) **Apply size difference** (Algorithm 2, line 6). Since we process the trees in non-decreasing size order and append new trees to the end of the inverted list, the trees in \( L \) are sorted by size. Due to the size difference lower bound, we can discard all trees \( T' \) with size \( |T'| < |T| - \tau \) at the beginning of the inverted list by incrementing a start pointer. For example, after the index lookup of label \( b \), \( T_1[b_1] \) is no longer relevant for \( T_3 \) nor for any probing tree after \( T_3 \), since \( |T_1| < |T_3| - \tau \).

d) **Structural mapping.** Algorithm 3 computes the \( \tau \)-valid label intersection, \( \{\text{nodes} \cap \text{nodes}'\} \), for two sets of nodes with identical labels, nodes \((\text{from tree } T)\) and nodes' \((\text{from tree } T')\). Since finding an optimal mapping is computationally expensive, we disregard the one-to-one mapping constraint from Definition 5, thus a node may be mapped to multiple \( \tau \)-valid nodes in the other set. Note, that weakening the constraints will make the lower bound looser but not incorrect.

Let \( d_2 \) be the smaller and \( d_1 \) be the larger set of nodes and nodes'. The maximum number of mappings is \( |d_2| \), which is achieved by mapping each node in the smaller set \( d_2 \). For each node in \( d_2 \), we scan at most \( 2|\tau + 1 \) nodes, called \( \tau \)-valid window, in \( d_1 \) to avoid a nested loop over \( d_1 \). The \( \tau \)-valid window is based on a necessary condition of \( \tau \)-validity: If \((v, w)\) is \( \tau \)-valid, then \(|\text{post}(v) - \text{post}(w)| \leq \tau \). Given node \( v \in T \), we know that \( \tau \)-valid pairs are in the set defined by a window of size \( 2\tau + 1 \) on the postorder ids of the nodes, that is, \( \{(v, w) : w \in T' \land \text{post}(v) - \tau \leq \text{post}(w) \leq \text{post}(v) + \tau \} \).

In our example, label \( d \) occurs once in \( T_2 \) (nodes' = \( (d_2) \)) and twice in \( T_3 \) (nodes = \( (d_3, d_5) \)). Since \( |\text{nodes}'| < |\text{nodes}| \), we verify whether \( d_2 \) is \( \tau \)-valid with \( d_3 \) or \( d_5 \). In fact, \((d_2, d_3)\) is \( \tau \)-valid. \( T_3 \) is removed from the index and \((T_2, T_3)\) is identified as pre-candidate.

e) **Structural filter** (Algorithm 2, lines 8-10). For a probing tree \( T \), we get a set of pre-candidates and the number of \( \tau \)-valid label matches in the prefix \( ov[T'] \) for each of the pre-candidates \( T' \). In a merge-like approach, we verify whether a pre-candidate passes the structural filter in Theorem 1. We compare the label of the last node in the prefix of the probing tree \( T \) and the pre-candidate \( T' \). For the tree with the larger label, we start the scan after the prefix; for the other tree we start at position \( ov[T'] \). We process the remaining nodes of the trees by computing the structural mapping (Algorithm 3) for nodes with identical labels. We scan until either the threshold is satisfied or cannot be reached.

In our example, the last label in the prefix of \( T_2 \) is \( a \), which is greater than \( d \) in \( T_3 \). Therefore, we start at node \( a_1 \) in \( T_2 \) and node \( b_4 \) in \( T_3 \). Further, \((a_1, a_1)\) is the third \( \tau \)-valid label match and, therefore, \((T_2, T_3)\) is returned as a candidate.

V. **Linear Upper Bound for TED**

The candidate pairs resulting from the candidate index must be verified. Most of the previous TED join algorithms compute the expensive tree edit distance for each candidate pair. Thus, even if the candidate filter is highly effective and returns no false positives, edit distance must be computed for each pair in the join result.

In this section, we show how to avoid expensive verifications using a novel upper bound for the edit distance. If the upper bound for a candidate pair is within the distance threshold, we can add the pair to the join result and need no further verification. Among the previous TED join algorithms, only the solution by Guha et al. [2] uses an upper bound for reducing the number of candidate pairs. They use the quadratic-time constrained tree edit distance [19] which we show to be inefficient for the join scenario.

We introduce the **label guided mapping** (LGM), the first upper bound that is both effective (close to the edit distance) and efficient (linear runtime). The key observation is that any edit mapping between two trees constitutes an upper bound for the tree edit distance. The closer the mapping is to the optimal one, the tighter is the bound. Consider how two nodes \( v \in T \) and \( w \in T' \) contribute to the cost of the mapping. If \( v \) and \( w \) are not mapped, \( v \) must be deleted and \( w \) inserted, increasing the mapping cost by two. If the nodes are mapped, they only add one (different labels) or zero (identical labels) to the cost. Thus, the goal is to map as many nodes as possible, ideally nodes with identical labels.

Finding a good edit mapping is challenging. Mapping a pair of nodes imposes restrictions on the remaining, unmapped nodes. Poor choices in the beginning may prevent good matches later in the process. Consider a simple approach that greedily maps nodes in one tree to nodes with the same label in the other tree. Then, node \( c_0 \in T_3 \) is mapped to \( c_4 \in T_4 \) (cf. Fig. 1), leading to a loose upper bound of 9: the pair \((c_0, c_4)\) allows only one additional pair, \((d_5, a_5)\), being mapped.

Our LGM algorithm carefully prevents poor choices by considering the tree structure in addition to the labels. We proceed in three steps. **Step (1)** We map nodes with identical labels and only consider \( \tau \)-valid node pairs to prevent poor matches. The result is a one-to-one mapping, but the order and ancestor conditions of the edit mapping may be violated. **Step (2)** We clean the mapping resulting from Step 1 by removing the
node pairs that violate an edit mapping condition. Step (3) All node pairs in the mapping resulting from Step 2 have identical labels. The nodes that could not be mapped based on their labels must be deleted or inserted, which increases the mapping cost. In this step, we check all unmapped nodes and try to map \( \tau \)-valid node pairs that preserve the edit mapping conditions and decrease the overall cost.

A. Verifying Edit Mapping Conditions in Constant Time

The LGM algorithm incrementally builds a mapping \( M \) by inserting new node pairs. Thus, LGM requires an efficient technique to verify the edit mapping conditions for every new pair. This is challenging because the conditions in Definition 1 must hold for any two node pairs in the mapping. Thus, adding a new pair to an existing mapping \( M \) requires \(|M|\) checks, leading to an overall quadratic algorithm. In the remaining section, we introduce new rules and techniques to verify a new pair in the mapping in constant time.

First, we derive new edit mapping conditions that are evaluated on a single node pair (rather than two pairs). We use the constraints imposed by the mapping regions (cf. Definition 3) of a new node pair \((v, w)\) that is inserted into \( M \), e.g., all descendants of \( v \) must be mapped to descendants of \( w \) (and vice versa). The constraints are expressed by restrictions of \( M \) to some subset of nodes, \( S \), defined as \( M|_S = \{(v, w) \in M : v \in S \land w \in S\} \). For example, \( M|_{d(T, w)} \) restricts \( M \) to those node pairs that map a descendant of \( v \).

**Lemma 5.** Given edit mapping \( M \) between \( T \) and \( T' \), a node pair \((v, w)\), \( v \in T, w \in T' \), such that \( M|_{\{v\}} = M|_{\{w\}} = \emptyset \), i.e., neither \( v \) nor \( w \) is mapped. \( M \cup \{(v, w)\} \) is an edit mapping iff \( M|_{a(T, w)} = M|_{a(T', w)} \), \( M|_{d(T, v)} = M|_{d(T', v)} \), \( M|_{l(T, v)} = M|_{l(T', v)} \), and \( M|_{r(T, w)} = M|_{r(T', w)} \).

Next, we state that edit mappings preserve the postorder sequence of the mapped nodes.

**Lemma 6.** For any two node pairs \((v_s, w_s), (v_e, w_e) \in M\): if \( v_s \) precedes \( v_e \) in postorder then \( w_s \) precedes \( w_e \) in postorder, i.e., \( \text{post}(v_s) < \text{post}(v_e) \Rightarrow \text{post}(w_s) < \text{post}(w_e) \).

A new node pair \((v, w)\) is always inserted into a so-called POSTORDER gap. The postorder gap is formed by two pairs \((v_s, w_s), (v_e, w_e) \in M\) (called start resp. end pair), such that \( v_s \) is the closest predecessor and \( v_e \) the closest successor of \( v \) in postorder, i.e., \( \text{post}(v_s) < \text{post}(v) < \text{post}(v_e) \). Due to Lemma 6, \( w \) can only be mapped between \( w_s \) and \( w_e \). Observe that there are no other pairs in the postorder gap.

We use the mapping regions defined by start pair \((v_s, w_s)\) and end pair \((v_e, w_e)\) to derive editing mapping checks for a newly inserted node pair \((v, w)\). We distinguish the two cases illustrated in Figure 3. Case I: \( v_e \) is an ancestor of \( v_s \). Case II: \( v_e \) is to the right of \( v_s \). Since \( \text{post}(v_s) < \text{post}(v_e) \), no other cases exist; the relative position of \( w_s/w_e \) is the same as for \( v_s/v_e \) due to the edit mapping conditions.

Node \( v \) of a newly inserted pair \((v, w)\) falls into one of the colored regions (Case I: \( A, R \), Case II: \( A, RL, D \)). Node \( w \) must always fall into the respective region in the other tree. Consider, for example, Case I: If \( v \) is in region \( A \) (i.e., \( v \) is an ancestor of \( v_s \)), \( w \) must be mapped to a node in region \( A \) in the other tree. However, mapping \( v \) and \( w \) to the same region is not always enough. Although there are no mapped nodes in the postorder gap that could violate the conditions in Lemma 5, \( v \) and \( w \) may have mapped descendants or ancestors outside the postorder gap. Thus, the conditions \( M|_{d(T, v)} = M|_{d(T', w)} \) or \( M|_{a(T, v)} = M|_{a(T', w)} \) could be violated.

In the following theorem we propose a new set of checks that cover all cases. The checks are based on ancestor and descendant count and are essential for our upper bound algorithm (Steps 2 and 3). We denote a subtree of tree \( T \) rooted at node \( v \) as \( T_v \); \( T_v \) consists of node \( v \), all its descendants, and all edges between these nodes.

**Theorem 3.** Given edit mapping \( M \) between \( T \) and \( T' \), a POSTORDER gap \((v_s, w_s), (v_e, w_e) \in M\), and a node pair \((v, w)\), \( v \in T, w \in T' \), \( \text{post}(v_s) < \text{post}(v) < \text{post}(v_e) \) such that \( M|_{\{v\}} = M|_{\{w\}} = \emptyset \), i.e., neither \( v \) nor \( w \) is mapped. \( M \cup \{(v, w)\} \) is an edit mapping if one of the conditions (i) to (iv) holds.

**Case I.** \( v \in a(T, v_s) \) and \( w_e \in a(T', w_e) \) (Figure 3a):

1. \( v \in a(T, v_s) \land w \in a(T, w_s) \land |M|_{d(T, v)} = |M|_{d(T', w)} \) (red region A).
2. \( v \in r(T, v_e) \land w \in r(T, w_e) \) (blue region R).

**Case II.** \( v \in r(T, v_s) \) and \( w \in r(T', w_e) \) (Figure 3b):

3. \( v \in a(T, v_s) \land w \in a(T', w_e) \land |M|_{d(T, v)} = |M|_{d(T', w)} \) (red region A).
4. \( v \in r(T, v_s) \land l(T, v_e) \land w \in r(T, w_s) \land l(T', w_e) \land l(T, w_e) \land |M|_{a(T, v)} = |M|_{a(T', w)} \) (yellow region RL).
5. \( v \in d(T, v) \land w \in d(T', w) \) (green region D).

B. Step 1: Greedy Label Mapping

In the first step of LGM, we map promising node pairs between the two input trees that have identical labels (Algorithm 4). A node pair is promising if it is \( \tau \)-valid. Mapping a node pair that is not \( \tau \)-valid would lead to an upper bound that is larger than \( \tau \), and thus is useless for our purpose (cf. Lemma 4). The output is a one-to-one mapping \( M_{out} \).
which may contain some node pairs that violate the edit mapping conditions.

The difficulty lies in the complexity of finding the right node pairs to map. It is prohibitive to scan the nodes of the trees in a nested loop fashion. To this end, we build an inverted list index $il$ that efficiently retrieves $\tau$-valid node pairs with identical labels. We use the index in an index nested loop way, in a nested loop fashion. To this end, we build an inverted list index $il$ contains all nodes in a tree that carry label $l$ (cf. Figure 4). The lists are obtained in a single postorder traversal of a tree. The nodes in each list are sorted by their postorder ids. Each node is stored exactly once in $il$, thus, the size of $il$ is linear in the tree size.

Traversing the entire inverted list $il[lb(l)]$ for each label lookup is expensive and scans a quadratic number of elements in the lists. We iterate over the nodes $v$ in $T$ in increasing postorder (line 3). For each node $v$, we search for $v$’s first partner (with the smallest postorder id) in $il[lb(l)]$, such that $(v, w)$ is $\tau$-valid. We limit the search to the $\tau$-valid window $[post(v) - \tau, post(v) + \tau]$. We update the pointer to the first element of the window while performing the lookups. If node $w$ is found, a new pair, $(v, w)$, is added to the result mapping $M_{out}$ (line 6). $w$ is removed from the inverted list (line 7) to guarantee a one-to-one mapping. Thanks to a $\tau$-valid window that we move while iterating over the nodes, for each node $v$ in $T$ at most $2\tau + 1$ nodes in $T'$ need to be verified. This guarantees $O(|T|\tau)$ runtime complexity.

Since we did not check the ancestor and order conditions during the mapping process, the result mapping $M_{out}$ may violate the edit mapping conditions. We fix that in the next step by removing such pairs from $M_{out}$.

![Algorithm 4: GreedyLabelMapping](image)

**Algorithm 4: GreedyLabelMapping**

**input**: trees $T$ and $T'$, threshold $\tau$

**output**: one-to-one mapping $M_{out}$ between nodes with identical labels

1. $M_{out}$: empty one-to-one mapping;
2. $il$: inverted list on nodes of $T'$;
3. for $v \in T$ in increasing postorder do
4.   $w \leftarrow$ first node in $il[lb(l)]$: $(v, w)$ is $\tau$-valid;
5.   if $w$ exists then
6.     $M_{out} \leftarrow M_{out} \cup \{(v, w)\};$
7.     remove $w$ from $il[lb(l)];$
8. return $M_{out}$;

![Algorithm 5: CleanMapping](image)

**Algorithm 5: CleanMapping**

**input**: one-to-one mapping $M_{in}$ between $T$ and $T'$

**output**: edit mapping $M_{out}$ \subseteq $M_{in}$

1. $M_{out}$: empty mapping; $v' \leftarrow c$; $w' \leftarrow c$; $w_M \leftarrow c$;
2. $d[]$: array of size $|T| + |T'|$; $pr[]$: array of size $|T'|$;
3. for $(v, w) \in M_{in}$ in increasing postorder of $v$ do
4.   if $post(w) > post(w_M)$ then
5.     while $post(v) \leq post(w)$ do
6.       $d[p(v')] \leftarrow d[p(v')] + d[v'];$
7.       $v' \leftarrow$ next node in postorder after $v'$ in $T';$
8.     while $post(w') \leq post(w)$ do
9.       $d[p(w')] \leftarrow d[p(w')] + pr[w'];$
10.      $pr[w'] \leftarrow 0;$ /* reset counter */
11.     $w' \leftarrow$ next node in postorder after $w'$ in $T';$
12.   if $d[v] = d[w]$ then
13.     $M_{out} \leftarrow M_{out} \cup \{(v, w)\};$
14.     $d[p(v)]++;$ $d[p(w)]++;$ $pr[p(w)]++;$
15.     $w_M \leftarrow w; w' \leftarrow w;$
16. return $M_{out};$

Fig. 4: Example for the Label Guided Mapping (LGM).

**Example 3.** Consider trees $T_4$ and $T_5$ in Figure 4, $\tau = 4$. The initial inverted list index $il$ is shown on the right. In line 7 the crossed-out nodes are removed. Algorithm 4 results in the mapping depicted by single lines, which is not an edit mapping. For example, $(d_2, d_1)$ violates the ancestor condition with the node pair $(d_3, d_4)$.

**C. Step 2: Clean the Mapping**

In the second step, we remove all pairs from a one-to-one mapping that violate the edit mapping conditions (Algorithm 5). We traverse the node pairs of the input mapping $M_{in}$ in increasing postorder of nodes in $T$ (line 3) and try to add it to the output mapping $M_{out}$, which initially is empty. In line 4, we check if the postorder id of the nodes in $T'$ increases (Lemma 6). We remember the mapped node $w_M \in T'$ that has the largest postorder id, and compare it to that of the current node $w$. We use Theorem 3 to verify each pair that we try to add. We observe that the nodes of each consecutive pair have larger postorder ids than all other nodes in the output mapping $M_{out}$. Thus, only cases (i) and (i) of Theorem 3 occur. If $v$ and $w$ do not have any mapped descendants, case (i) occurs. Otherwise, some of their descendants have been mapped, and case (i) takes place. We verify the number of mapped descendants in line 13. Only if this condition is satisfied, a node pair is added to the output mapping (line 14).

We maintain the number of mapped descendants incrementally in a bottom-up fashion that guarantees linear time complexity of this step. $d[v]$ stores the number of descendants of node $v$ that are in $M_{out}$. When $v$ is processed, we update the number of mapped descendants of $v$’s parent. While traversing the node pairs in line 3, to correctly propagate the counts, we process also the unmapped nodes $v'$ and $w'$, such that $post(v') < post(v)$ and $post(w') < post(w)$ (lines 5 to 12).
Each node that is mapped increments the number of mapped descendants of its parent by one (line 15). The postorder ids of the nodes from $T'$ may decrease although we traverse $M_{in}$ in increasing postorder of the nodes in $T$. First, we ensure that we do not propagate the number of mapped descendants multiple times. This is important because we may encounter a node multiple times. Therefore, we use a propagation array $pr$, which stores the number of $w'$'s mapped descendants to propagate it to $w$'s parent ($pr[w]$). Once the propagation at $w$ is done, we reset $pr[w]$, such that the mapped descendants are not propagated again (line 11). Second, we show in Lemma 7 that the maximum number of times we encounter a particular node is bounded by $2\tau$. This is due to the fact that the nodes in every node pair are within a $\tau$-valid window.

**Lemma 7.** Given two consecutive node pairs in a one-to-one mapping $M$ resulting from Algorithm 4, $(v, w)$ and $(v', w')$, such that $post(v) < post(v')$ and $post(w) > post(w')$, the following holds: $post(w) - post(w') < 2\tau$.

With a constant number of operations for each node in the input trees and Lemma 7, the runtime complexity of this step is $O(|T|\tau)$.

**Example 4.** Consider trees $T_3$ and $T_5$ in Figure 4, and the threshold $\tau = 4$. GreedyLabelMapping (Algorithm 4) results in the mapping depicted by single lines (cf. Example 3). We try to add the node pairs from $M_{in}$ to $M_{out}$ one by one and maintain the number of mapped descendants. Pair $(d_2, d_1)$ is added. Next, $(d_3, d_4)$ is discarded, since $|M_{out}|[d(T_3, d_3)] = 1$ and $|M_{out}|[d(T_3, d_4)] = 0$. The next pair $(c_4, c_2)$ is added to $M_{out}$. At the last pair, $(a_5, a_5)$, we process the nodes $b_3$ and $d_4$ again for updating the numbers of mapped descendants. The counter $pr[b_3]$ was reset, which ensures that we do not update $d[a_5]$ twice with the mapped descendants of $b_3$.

**D. Step 3: Filling the Gaps**

In the last step, we add additional node pairs into the final mapping in order to decrease its overall cost (Algorithm 6). We substitute deletions and insertions of cost 2 with renames of cost 1. We fill every postorder gap in the input edit mapping $M_{in}$ with additional node pairs, such that the result mapping $M_{out}$ is an edit mapping. The cost of $M_{out}$ is less than the cost of $M_{in}$ by the number of node pairs we add in this step.

We consider any two consecutive node pairs, $(v, w)$ and $(v', w')$, in the input mapping $M_{in}$ (line 3). If there is a postorder gap $(v, w)$ or $(v', w')$ between these node pairs (line 4), there are nodes that can potentially be mapped. We use dummy node pairs $(e_{-1}, e_{-1})$ and $(e_T, e_T)$ to support the gaps before and after all node pairs in $M_{in}$. We look for the first $\tau$-valid node pair $(v, w)$ in the gap (nodes with the smallest postorder ids) such that one of the conditions of Theorem 3 is satisfied (line 5). To keep the search complexity low, we do not consider all pairs of nodes in a gap. Instead, we employ a merge-like approach which advances at each step either $v$ or $w$, wherever the postorder gap is larger. Each time a new node pair $(v, w)$ is found, the new pair marks the beginning of a new gap (line 5), and the pair is added to the result mapping (line 6).

We verify the conditions of Theorem 3 in constant time as follows. To check if nodes occur in a specific region, we compare their postorder and preorder ids\(^2\) as in Table I. The number of mapped descendants is maintained as in Algorithm 5. The number of mapped ancestors is precomputed in linear time at the beginning of Algorithm 6 by traversing a tree top-down and propagating the values to the children of each node. As opposed to the number of descendants, the number of mapped ancestors will never change for a node in the course of Algorithm 6 since the node pairs are added with increasing postorder ids.

Thanks to a merge-like approach for finding additional node pairs, this step can be computed in $O(|T|)$ time.

**Example 5.** Consider trees $T_4$ and $T_5$ in Figure 4, $\tau = 4$, and the results of CleanMapping as in Example 4, that is, $M_{in} = \{(d_2, d_1), (c_4, c_2), (a_5, a_5)\}$. Algorithm 6 finds one gap between the node pairs $(v, w) = (e_{-1}, e_{-1})$ and $(v', w') = (d_2, d_1)$. The node pair $(v, w) = (e_0, a_0) \ (\text{double line})$ is added to the output mapping as of case (i) in Theorem 3, that is, $e_0 \in a(T_4, e_{-1}) \land a_0 \in a(T_5, e_{-1}) \land |M_{out}[d(T_4, e_0)]| = |M_{out}[d(T_5, a_0)]| = 1$. The algorithm returns the final mapping $M_{out} = \{(e_0, a_0), (d_2, d_1), (c_4, c_2), (a_5, a_5)\}$, which has cost 5 (rename $e_0$ to $a_0$, delete $b_1$, $d_3$, insert $b_3$, $d_4$). Since $5 > \tau = 4$, $(T_4, T_5)$ cannot be immediately sent to the join result, and it has to be verified.

**E. Linear Time Upper Bound LGM**

Our LGM algorithm executes GreedyLabelMapping, CleanMapping, and FillGaps sequentially. Summarizing, the time

```
Algorithm 6: FillGaps

input: edit mapping $M_{in}$, threshold $\tau$
output: edit mapping $M_{out}$, $M_{out} \supseteq M_{in}$
1 $M_{out} = M_{in}$;
2 $(v_s, w_s) \leftarrow (e_{-1}, e_{-1})$;
3 for $(v_e, w_e) \in M_{in} \cup \{(e_T, e_T)\}$ in incr. post. of $v_e$ do
4   while $(v_e, w_e)$ has a $\tau$-valid pair do
5     $(v_s, w_s) \leftarrow$ first $\tau$-valid pair in the gap;
6     $M_{out} \leftarrow M_{out} \cup \{(v_s, w_s)\}$;
7     $(v_s, w_s) \leftarrow (v_e, w_e);$ 
8   return $M_{out}$;
9
Case I. $(v_e, w_e) \in a(T, v_e) \Rightarrow pre(v_e) < pre(v_s) \land post(v_s) > post(v_e)$
(i) $v \in a(T, v_e) \Rightarrow pre(v) > pre(v_e) \land pre(v) < pre(v_s)$
(ii) $v \in r(T, v_e) \Rightarrow pre(v) < pre(v_s)$
Case II. $(v_e, w_e) \in r(T, v_e) \Rightarrow pre(v_e) > pre(v_s) \land post(v_s) > post(v_e)$
(i) $v \in a(T, v_e) \Rightarrow pre(v) < pre(v_e)$
(ii) $v \in r(T, v_e) \Rightarrow pre(v) > pre(v_e) \land pre(v) < pre(v_s)$
(iii) $v \in d(T, v_e) \Rightarrow pre(v) > pre(v_s)$

TABLE I: Verifying the mapping regions.
```

\(^2\)The preorder traversal visits the root node first and recursively traverses the subtrees rooted in the children of a visited node in increasing sibling order.
and space complexities of LGM between two trees, $T$ and $T'$, are $O(n\tau)$ and $O(n)$ ($n = \max\{|T|,|T'|\}$), respectively.

VI. VERIFYING IN LINEAR TIME

The candidate pairs $(T, T')$ that pass the upper bound filter must undergo verification. All previous works compute the tree edit distance $D = \delta(T, T')$ and check $D \leq \tau$ to identify result pairs. This is expensive, since the best algorithms for TED require $O(n^3)$ time and $O(n^2)$ space in the number of tree nodes. In this section, we show how we verify in linear time and space. We leverage the fact that we do not need to know the exact distance value $D$, but only need to verify if $D \leq \tau$. Our solution is based on the bounded structural modification distance.

**Definition 6.** The bounded structural modification distance, $\text{BSM}(T, T', k)$, is the cost of the optimal edit mapping with at most $k$ unmapped nodes, deletions in $T$ or insertions in $T'$. If no such edit mapping exists, then $\text{BSM}(T, T', k) = \infty$.

BSM restricts the search space of allowable edit mappings w.r.t. TED and is easier to compute. In fact, Touzet [16] proposes an algorithm that solves BSM in $O(nk^3)$ time and $O(nk)$ space. We show that we can substitute TED by BSM during verification and still get the exact result. Thus, we reduce the runtime of the verification step from cubic to linear.

**Theorem 4.** For a given distance threshold $\tau$, the following holds: $\text{BSM}(T, T', \tau) \leq \tau \iff \delta(T, T') \leq \tau$.

The intuition of this result is as follows. Assume the minimum-cost edit mapping between two candidate trees $T$, $T'$, requires $d$ deletions, $i$ insertions, and $r$ renames, i.e., $\delta(T, T') = d + i + r$. For a given distance threshold $\tau$, we distinguish the following cases. (1) If $\tau \geq d + i$, then we allow enough structural modifications to find the optimal edit mapping, thus $\text{BSM}(T, T', \tau) = \delta(T, T')$. (2) If $\tau < d + i$, then BSM produces a sub-optimal edit mapping if any edit mapping with $\tau$ structural modifications exists, or returns infinity otherwise. In both cases, $\text{BSM}(T, T', \tau) > \tau$ and we will discard the candidate pair $(T, T')$. This is correct, since $\tau < d + i \leq d + i + r = \delta(T, T')$.

VII. RELATED WORK

**Tree Similarity Joins.** Kailing et al. [13] and Yang et al. [15] introduce lower bounds to reduce the number of expensive tree edit distance computations. The edit distance is only evaluated if the lower bound is below the threshold. Kailing et al. [13] introduce lower bounds based on the $L_1$ distance between histograms over node properties (label, node degree, distance to leaf). Yang et al. [15] split the trees into so-called binary branches, which are used to compute a linear time lower bound. Augsten et al. [20] follow a similar approach and split the trees into pq-gram tokens. However, pq-grams provide a lower bound only for a variant of the edit distance that weights nodes based on the number of their children [21].

Guha et al. [2] leverage the metric properties of the tree edit distance to produce candidates. Before verification, a string-based lower bound and the constrained edit distance upper bound [19] are applied, both requiring quadratic time. We develop linear time lower and upper bounds for the tree edit distance and show their effectiveness in a join scenario.

The recent algorithm by Tang et al. [14] performs an index nested loop join. The index leverages a lower bound based on the pigeonhole principle to retrieve join candidates. Trees are systematically partitioned into subgraphs such that only trees that share a common subgraph qualify as candidates.

We empirically compare to the previous tree similarity join algorithms [2], [13]–[15] and show the superior performance of our technique. Our index produces a smaller number of candidates. In addition, we can avoid many of the verifications using our novel, linear time upper bound. Overall, our algorithm scales to larger datasets and clearly outperforms all other approaches.

**Tree edit distance.** APTED [12], the state-of-the-art tree edit distance algorithm runs in optimal $O(n^3)$-time [22] and $O(n^2)$-space. In a theoretical work, Touzet [16] computes the bounded structural modification (BSM) distance, which is the minimum edit distance for a given maximum number $k$ of allowable structural modifications including deletions and insertions, in $O(nk^3)$ time and $O(nk)$ space. We leverage BSM to verify join candidates in linear time. All previous tree similarity join algorithms compute the expensive tree edit distance, which prevents them from scaling to large trees.

Li et al. [23] survey lower bounds for tree similarity joins. They compare string-based [2], histogram-based [13], and set-based bounds [15]. An efficient join framework, which was originally developed for pq-grams [21], is used to avoid a nested loop join between trees. We empirically compare to all of these techniques and also use the efficient join framework.

Tree edit distance variants that disregard the sibling order [24] or allow subtree move [25] have shown to be NP-complete. Heuristic join algorithms for this setting include, for example, windowed pq-grams [26], tree embeddings [27], and the histogram-based approach by Kailing et al. [13].

**Other.** Similarity joins have also been proposed for sets [28], strings [29], and graphs [30]. An effective technique for sets is the prefix filter [31], which is leveraged in the AllPairs join [32] to produce candidates. We use a similar technique to generate join candidates in our index. Compared to set prefix filters, (a) in addition to node label sets we must also consider the structure of a tree and develop filter guarantees w.r.t. the tree edit distance, (b) our prefix is independent of the tree size, whereas prefixes for set similarity are linear in the set size $s$ (e.g., $s - \lceil \tau s \rceil + 1$ for Jaccard thresholds $0 < \tau < 1$); this allows us to build a lightweight index for arbitrarily large trees.

VIII. EXPERIMENTS

We compare our solution, TJoin, to four competitors: Tang [14], Binary [15], Histo [23], and Guha [2]. We implemented all algorithms in C++. The source code for the algorithms and the experimental setup (including data preprocessing scripts) are available online [33]. We execute the experiments single-threaded on an Intel Xeon E5-2630 v3 2.40GHz server with 8 cores and 96GB of RAM, running
Debian 8.11. We use a timeout of 10 hours on the overall join runtime.

**Algorithms.** To avoid a nested loop join, Binary [15] and Histio [23] use the efficient join framework by Augsten et al. [21]. Guha [2] computes so-called reference vectors to leverage the metric properties of the edit distance. The vectors can be computed using either the exact tree edit distance (called RSB [2]) or a lower and an upper bound (RSC [2]). We tested both RSB and RSC with varying reference vector sizes (2^i, 1 ≤ i ≤ 6), and we show the winning configuration in our plots (RSB, vector size 8). To compute the vectors, a set of reference trees must be picked. We do not count the runtime for selecting the reference trees, thus all runtime numbers for Guha are lower bounds.

**Tang for small trees.** Tang’s candidate index partitions each tree into δ = 2τ + 1 non-overlapping subgraphs. Tree pairs that share at least one subgraph are candidates. However, a small tree with less than δ nodes cannot be partitioned into δ subgraphs and thus cannot be indexed. Tang et al. [14] do not discuss this issue. To get correct join results, we implement the following fix: In addition to the trees in the index, the lookup of a tree T returns all trees T’, |T| − τ ≤ |T’| < δ, as candidates. The runtime difference in our experiments is small and does not change the overall picture.

**Datasets.** We run our experiments on four real-world datasets from various domains with different tree characteristics (cf. Table II). DBLP: bibliographic XML data (snapshot Nov. 1, 2017); a tree is a scientific publication; we include all entries except “www” elements, which are small subtrees that all match for τ ≥ 3, blowing up the join result by billions of pairs. Python: Abstract syntax trees (AST) of Python source code in JSON. Sentiment: Semantic trees of movie reviews in the PennTreeBank format. Swissprot: protein sequence data in XML. All datasets are publicly available [33].

### A. Join Runtime

We measure the overall join runtime in Figure 5. On the smallest dataset, Sentiment, which does not contain any large trees, all algorithms complete within the 10 hours timeout. TJoin is about five times faster than the best competitor, Histio. In Figure 5b, we sample three subsets of DBLP with increasing sizes (10k, 50k, 100k trees), and use τ = 6. TJoin scales clearly better than all other algorithms with increasing dataset size, resulting in a runtime difference of more than one order of magnitude. In fact, the total join time of TJoin is below the candidate generation time (cf. Figure 7b) of any of the competitors in this experiment. TJoin is the only algorithm that terminates within the timeout on all datasets including Swissprot, DBLP, and Python (cf. Figures 5c-5e). Histio and Binary complete for small thresholds of Swissprot, but TJoin is two orders magnitude faster.

### B. Candidate Generation

We analyze the performance of the candidate generation in Figures 6 and 7. Tang, Binary, and Histio compute the tree edit distance on all candidates; Guha applies the string edit distance lower bound and the constrained edit distance upper bound (CTED) to all candidates before verification; TJoin filters candidates using the LGM upper bound (LGM). We only present numbers for experiments with a total join runtime below the 10 hours timeout.

**Effectiveness.** In Figure 6, we count the number of candidates and compare it to the size of the final join result, a subset of the candidate sets. TJoin generates the smallest candidate set for all thresholds and datasets. Histio performs well for XML-based datasets (cf. Figure 6c and 6d), but also on these datasets TJoin generates fewer candidates by a factor of 1.5.

**Efficiency.** In terms of candidate generation time, TJoin outperforms its competitor in almost all settings and is up to two orders of magnitude faster (cf. Figure 7). The candidate index of TJoin only indexes and looks up τ + 1 nodes of a tree, which makes it highly efficient, especially for large trees and large datasets. For small datasets and high thresholds (cf. Figure 7a), Histio and Binary produce candidates faster than TJoin, but their higher number of candidates leads to a slower join time.

**Discussion.** Overall, our candidate index produces fewer candidates than any competitor and is faster in almost all settings. The runtime difference is up to two orders of magnitude.

Tang and Guha perform expensive steps to produce candidates. Tang must partition the input trees at runtime to build the index. Guha computes so-called reference vectors which are further processed in a quadratic-size nested loop to produce candidates. In [34] an R-tree is proposed to avoid the nested loop. However, the resulting candidates are the same. In our experiments, Guha’s candidate generation is only a small fraction of the overall join runtime. Thus, we do not expect the R-tree to give a significant improvement. Tang, Histio, and Binary consider each tree pair that either shares a so-called top twig in some postorder range (Tang), a label (Histio), or a binary branch (Binary). The resulting pre-candidates are further filtered to produce the candidates. The pre-candidate set may grow large and limits the applicability to large datasets.

We encode global structure information in the index and consider only τ-valid label matches to reduce the number of pre-candidates. Further, we index only O(τ) nodes with infrequent labels of each tree, which is fast (in terms of creating the index) and reduces the probability of a false positive match. Finally, we apply the structural filter on all pre-candidates to achieve a small candidate set.

### C. LGM Upper Bound and BSM Verification

Before we verify a candidate pair, we apply the LGM upper bound filter. Only pairs that pass the filter need further verification with BSM.

<table>
<thead>
<tr>
<th>Name</th>
<th>Tree Size</th>
<th>Label</th>
<th>Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>26</td>
<td>2.987</td>
<td>9</td>
</tr>
<tr>
<td>Python</td>
<td>946</td>
<td>46.481</td>
<td>1</td>
</tr>
<tr>
<td>Sentiment</td>
<td>37</td>
<td>103</td>
<td>3</td>
</tr>
<tr>
<td>Swissprot</td>
<td>802</td>
<td>48,286</td>
<td>101</td>
</tr>
</tbody>
</table>

**Table II:** Statistics and details of the analyzed datasets.

<table>
<thead>
<tr>
<th>Name</th>
<th>Tree Size</th>
<th>Label</th>
<th>Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>26</td>
<td>2.987</td>
<td>9</td>
</tr>
<tr>
<td>Python</td>
<td>946</td>
<td>46.481</td>
<td>1</td>
</tr>
<tr>
<td>Sentiment</td>
<td>37</td>
<td>103</td>
<td>3</td>
</tr>
<tr>
<td>Swissprot</td>
<td>802</td>
<td>48,286</td>
<td>101</td>
</tr>
</tbody>
</table>
Effectiveness of Upper Bound. Figures 6c-6e show the number of candidates that are sent to the join result thanks to our effective LGM upper bound filter. Even in the worst case, LGM identifies 22% of the join result (Swissprot, $\tau = 55$). For all other datasets the ratio is higher, reaching 100% for DBLP, i.e., only false positive pairs need to be verified.

Candidate trees are typically similar since they pass the structural filter. We evaluate the tightness of the upper bound also on dissimilar trees. To this end, we sort the trees in our datasets by size (breaking ties arbitrarily) to get a sequence of trees, $T_1, T_2, \ldots, T_n$. Then, for a given threshold $\tau$, we generate a new dataset of tree pairs $(T_i, T_{i+1})$ for each $i < n$; we exclude tree pairs for which the size difference exceeds the threshold. We call the new datasets DBLPPairs and SentimentPairs.

The results for $\tau = 10$ are shown in Figures 8a-8b. The size (x-axis) of a tree pair $(T_i, T_j)$ is $(|T_i| + |T_j|)/2$, the relative error ($(u - \delta)/\delta$: $u$ – upper bound, $\delta$ – edit distance) is averaged over all tree pairs of the same size (y-axis). In addition to the relative error of the LGM upper bound, we also show the dissimilarity of the tree pairs in the dataset (measured as their edit distance normalized by their average size). The relative error of the LGM upper bound is small although many of the tree pairs show a high degree of dissimilarity. For SentimentPairs, the average error is 0.26 with standard deviation $\sigma = 0.13$; for DBLPPairs, the average error is 0.1, $\sigma = 0.13$. The other datasets show similar behavior.

We also plot the relative error of the constrained tree edit distance (CTED) upper bound used by Guha. CTED is very effective with low average error (SentimentPairs: 0.07, $\sigma = 0.04$; DBLPPairs: 0.04, $\sigma = 0.06$); unfortunately, CTED requires quadratic time in the tree size and is not fast enough to improve the overall join time, as discussed below.

Runtime. We compare the runtime of the LGM upper bound, BSM verification, CTED upper bound, and the state-of-the-art edit distance algorithm, APTED [12]. Figures 8c-8d show the runtime results for DBLPPairs and SentimentPairs, $\tau = 10$; we get similar figures for the other datasets. As expected, both LGM and BSM are much faster than the edit distance computation with APTED. The difference is up to two orders of magnitude for LGM vs. APTED on DBLPPairs. The LGM upper bound is faster than BSM by up to a factor of 4.7 for DBLPPairs and 8.5 for SentimentPairs for verifying an individual tree pair. The runtime difference between LGM and BSM increases with the threshold, e.g., when we verify all trees in SentimentPairs we get runtime factors 1.4, 4.4, 7.9, and 13.5 for thresholds 5, 10, 15, and 20, respectively. This is expected since BSM requires $O(n^3)$ time while LGM runs in $O(n\tau)$ time. Interestingly, in our experiments the runtime of LGM is not affected by $\tau$, i.e., the worst case (when we repeatedly must scan $\tau$ duplicate labels to find a $\tau$-valid node pair) does not occur in any of the real-world datasets.
The performance of the CTED upper bound is similar to that of APTED on DBLPairs, and it ranges between BSM and APTED on SentimentPairs, i.e., CTED is slower than BSM verification. For an upper bound to be beneficial in a join scenario, however, it must be faster than the verification.

IX. CONCLUSION

In this paper, we proposed the first tree similarity join algorithm that scales to both large trees and datasets with millions of trees. We developed a new candidate index that efficiently produces a small set of join candidates. We introduced LGM, a novel upper bound filter that we successfully applied to verify candidates. LGM is the first non-trivial linear time upper bound for the tree edit distance. We are also the first to propose a linear time verification technique and thus avoid executing the cubic time tree edit distance algorithm on the join candidates. In an empirical evaluation, we executed tree similarity joins on large real-world datasets with millions of trees and could show that our algorithm outperforms all previous solutions, often by orders of magnitude.

Acknowledgments. We wish to thank Sebastian Arming and Daniel Kocher for their valuable suggestions, and Manuel Widmoser for his help with the implementation.

REFERENCES