An Incrementally Maintainable Index for Approximate Lookups in Hierarchical Data

Nikolaus Augsten  
Free University of Bozen-Bolzano  
Dominikanerplatz 3  
Bozen, Italy  
augsten@inf.unibz.it

Michael Böhlen  
Free University of Bozen-Bolzano  
Dominikanerplatz 3  
Bozen, Italy  
boehlen@inf.unibz.it

Johann Gamper  
Free University of Bozen-Bolzano  
Dominikanerplatz 3  
Bozen, Italy  
gamper@inf.unibz.it

ABSTRACT

Several recent papers argue for approximate lookups in hierarchical data and propose index structures that support approximate searches in large sets of hierarchical data. We propose approximate searches in large sets of hierarchical data. The index is based on small tree patterns, called pq-grams. It supports efficient updates in response to structure and value changes in hierarchical data and is based on the log of tree edit operations. We prove the correctness of the incremental maintenance for sequences of edit operations. Our algorithms identify a small set of pq-grams that must be updated to maintain the index. The experimental results with synthetic and real data confirm the scalability of our approach.

1. INTRODUCTION

Index structures are widely deployed and are being used to index vast amounts of documents with a hierarchical structure on the web. An important property of index structures is how to incrementally update them in response to structure and value changes in the source documents. We propose a persistent and incrementally maintainable index for approximate lookups in hierarchical data. The index is based on small tree patterns, called pq-grams. It supports efficient updates in response to structure and value changes in hierarchical data and is based on the log of tree edit operations. We prove the correctness of the incremental maintenance for sequences of edit operations. Our algorithms identify a small set of pq-grams that must be updated to maintain the index. The experimental results with synthetic and real data confirm the scalability of our approach.

As an application scenario consider Figure 1. \( T_0 \) is a document with a hierarchical structure (e.g., the DBLP file, 211MB). \( \mathcal{I}_0 \) is the index for \( T_0 \). \( T_0 \) is modified by a sequence of edit operations resulting in \( T_n \). Our goal is to update the index structure based on: (1) the old index \( \mathcal{I}_0 \), (2) the resulting document \( T_n \), and (3) the log of inverse edit operations that describes how \( T_n \) can be transformed to \( T_0 \). Note that we do not require that the original document be still available, and we assume that it is not feasible to recompute the index from scratch.

![Figure 1: Application Scenario.](image)

Our key contribution is the proof that we do not need to reconstruct intermediate versions of the document. All inverse edit operations can be applied to the resulting document \( T_n \) to compute the changes to the old index. Note that it is not obvious that this is possible, since the edit operations may depend on each other and have been defined on intermediate trees that can be very different from the resulting tree.

The paper makes the following contributions:

- We define the pq-gram index, which supports approximate lookups in data with a hierarchical structure. The pq-gram index is based on pq-grams [2], which generalize q-grams [17]. Intuitively, the pq-grams of a tree are all its subtrees of a specific shape.
- We prove that the pq-gram index can be updated incrementally given the old index, the log of edit operations, and the resulting document. The index update does not require the reconstruction of intermediate versions of the document.
- We show experimentally that our method efficiently handles logs of several thousand edit operations.

The paper proceeds as follows: Section 2 discusses related work, Section 3 defines the pq-gram index, and Section 4 gives an outline on our approach. Section 5 develops the incremental maintenance for a single edit operation, Section 6 generalizes to a sequences of edit operations and proves the correctness. In Section 7 we discuss the computation of the index maintenance functions. Section 8 discusses the implementation. Section 9 gives experimental results. Section 10 summarizes and points to future research directions.
2. RELATED WORK

Guha et al. [7] propose a framework for indexing approximate XML joins. Each XML document is represented by an XML Document Distance vector (XDD) that stores the distances between the document and all documents in a reference set. The use of XDDs reduces the number of distance computations in a join. Guha et al. [8] investigate the use of R-trees to efficiently access the XDDs that are relevant for pruning. The update of XDDs is not addressed. Building the XDD from scratch means recomputing the distance of the same document are given and the difference is computed [4, 6, 19]. None of these works addresses index maintenance.

Structural joins [1, 9] compute structural relationships (e.g., ancestor-descendant) between XML element sets. Structural joins are part of the XML query evaluation and are not used to approximately match XML documents.

XML queries typically specify path expressions or twig patterns that combine content and structural information. Some papers investigate exact answers [3, 5, 11, 13], while others allow approximate answers [14, 15]. Schenkel et al. [16] introduce a ranking of documents that satisfy the XML query. Typically the twig patterns are much smaller than the same document and the goal is to find parts of the document that match the pattern. The indexes proposed for XML queries have been specialized for this setup and do not support the matching of pairs of large documents.

A number of works propose index-like structures to compute an approximate distance between hierarchical data [2, 6, 19]. None of these works addresses index maintenance.

Our index is based on the pq-gram distance [2], an approximation of the tree edit distance. Augsten et al. [2] give an algorithm to compute the pq-gram distance in $O(n \log n)$ in the number of nodes. For the distance computation they represented the tree as a set of pq-grams. Updates of pq-grams are not addressed: If the data changes, the entire set of pg-grams has to be re-computed. We show that the computation of the pq-grams is by far the most expensive part of the distance computation. We propose the pq-gram index, a persistent and incrementally maintainable index for computing the pq-gram distance. We prove that the pq-gram index can be updated given the old index, the log of edit operations, and the resulting document. It is not necessary to reconstruct intermediate document versions. Our experiments compare the incremental index update with the approach of Augsten et al. and show major performance gains.

3. THE pq-GRAM INDEX

3.1 Preliminaries

A tree $T$ is a directed, acyclic, connected, non-empty graph with nodes $N(T)$ and edges $E(T)$. A node, $n \in N(T)$, is an (identifier, label)-pair. The identifier, $\text{id}(n)$, is unique within the tree. The label, $\lambda(n)$, is a symbol $\sigma \in \Sigma$, where $\Sigma$ is a finite alphabet. A node $\ast$ with the special label $\lambda(\ast) = \ast$ is a null node. We represent nodes by their id or the (id, label)-pair. An edge is an ordered pair $(v, c)$, where $v, c \in N(T)$ are nodes, and $v$ is the parent of $c$. Nodes with the same parent are siblings. Siblings are ordered. Contiguous siblings $s_1 < s_2$ have no sibling $x$ such that $s_1 < x < s_2$.

Node $c_i$ is the $i$-th child of $v$ if $v$ is the parent of $c_i$ and $i = |\{x \in N(T) : (v, x) \in E(T), x \leq c_i\}|$. The number of $v$'s children is its fanout $f_v$. The node with no parent is the root node, $r = \text{root}(T)$, and a node without children is a leaf. A subtree $S \subseteq T$ is a tree with $N(S) \subseteq N(T)$ and $E(S) \subseteq E(T)$ that retains the node order. A forest, $F$, is a set of trees.

An ancestor of $n$ is a node $a$ in the path from the root node to $n$, $a \neq n$. If there is a path of length $k > 0$ from $a$ to $n$, then $a$ is the ancestor of $n$ at distance $k$, and we write $\text{dist}(a, n) = k$. We define $\text{dist}(n, n) = 0$. The parent of a node is its ancestor at distance $1$. $d$ is a descendant of $n$ if $n$ is an ancestor of $d$.

An edit operation $e_j$ transforms a tree $T_i$ into a tree $T_j$, denoted as $T_j = e_j(T_i)$. The inverse edit operation, $e_j^\ast$, undoes $e_j$, i.e., $T_i = e_j^\ast(T_j)$. If a tree $T_0$ is transformed by a sequence of edit operations $(e_1, \ldots, e_n)$ into $T_n$, the log $L = (e_1, \ldots, e_n)$ is the sequence of inverse edit operations that (if applied in inverse order) transform $T_n$ back to $T_0$.

We use the following standard tree edit operations [20] that transform $T_i$ to $T_j$:

- $\text{INS}(n, v, k, m)$: Insert a new node $n$ as a child of node $v$ at position $k$ by substituting the children $c_k, c_{k+1}, \ldots, c_m$ of $v$ with $n$, and inserting them as children of $n$ (preserving the order). The inverse edit operation is $e_j = \text{DEL}(n)$.
- $\text{DEL}(n)$: Delete node $n$ by substituting $n$ with its children, i.e., remove $n$ and connect $n$'s children directly to $n$'s parent node (preserving the order). The inverse operation is $\tilde{e_j} = \text{INS}(n, v, k : (k + f_n - 1))$, where $n$ is the $k$-th child of $v$ in $T_i$, and $f_n$ is the fanout of $n$.
- $\text{REN}(n, l')$: Rename a node $n$ by changing its label $l$ to $l' \in \Sigma$, $l \neq l'$. Inverse operation: $\tilde{e_j} = \text{REN}(n, l)$. 

Figure 2: Sequence of Edit Operations that Transforms Tree $T_0$ into $T_3$. 

\[
\begin{align*}
T_0 & = n_1.a \\
T_1 & = n_1.a \\
T_2 & = n_1.a \\
T_3 & = n_1.a
\end{align*}
\]
Throughout the paper we assume that the root node is not an ancestor of the anchor node $N_0$, of $T_0$.

Figure 2 shows an example tree $T_0$ that is transformed to $T_3$ by a sequence of 3 edit operations.

Below we list standard set algebra rules that we use in our proofs. For sets $A$, $B$, and $C$ the following holds:

1. $(A \cap B) \cup (A \setminus B) = A$
2. $A \setminus (A \cap B) = A \cap B$
3. $(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$
4. $(A \setminus B) \cup B = A \cup B$

If we operate on bags, we use the symbols $\cap$, $\setminus$ to denote bag intersection, difference, and union, respectively.

3.2 The pq-Gram Index

The $pq$-gram index is used to efficiently compute approximate matches in hierarchical data. Intuitively, the $pq$-grams of a tree are all subtrees of a specific shape. Trees that share a high percentage of $pq$-grams are considered similar to trees that share a low percentage.

**Definition 1. pq-Gram.** Let $T$ be a tree, $a$ be a node in $N(T)$, $p > 0$, $q > 0$, and let $T'$ be $T$ extended with null nodes as follows: $p - 1$ ancestors to the root node, $q - 1$ children before the first and after the last child of each non-leaf node, and $q$ children to each leaf.

A $pq$-gram, $g$, of $T$ with anchor node $a$ is a subtree of $T'$ that is composed of the following nodes: $p$ nodes $a_{p-1}, \ldots, a_1, a$, denoted as $p$-part of $g$, where $a_1$ is the ancestor of $a$ at distance $i$; $q$ contiguous children $c_i, \ldots, c_{i+q-1}$ of $a$, denoted as $q$-part of $g$.

We use a linear encoding and represent a $pq$-gram $g$ with anchor node $a$ as a tuple $(a_{p-1}, \ldots, a_1, a, c_i, \ldots, c_{i+q-1})$.

**Example 1.** Consider tree $T_0$ in Figure 2. Figure 3 shows part of the extended tree $T'_0$ ($p = q = 3$) together with two $pq$-grams of $T_0$, namely $g_1 = (a_1, a_2, a, a_3, a_4, a_5)$ with anchor node $n_1$ and $g_2 = (a_1, a_2, a_3, a_4, a_5, a)$ with anchor node $n_5$. The total number of $pq$-grams of $T_0$ is 13.

**Definition 2. pq-Gram Profile.** Let $T$ be a tree, $p > 0$, $q > 0$. The $pq$-gram profile, $P_T$, of tree $T$ is defined as the set of all $pq$-grams of $T$.

![Figure 3: Part of $T'_0$ and Two 3,3-Grams of Tree $T_0$.](image)

**Example 2.** The $pq$-gram profiles of $T_0$ and $T_2$ in Figure 2 are given as follows:

$P_0 = \{ (\ast, \ast, a_1, a_2, a_n), (\ast, \ast, a_1, a_2, a_n, a_3), (\ast, \ast, a_1, a_2, a_n, a_3, a_4), (\ast, \ast, a_1, a_2, a_n, a_3, a_4, a_5), (\ast, \ast, a_1, a_2, a_n, a_3, a_4, a_5, a_6), (\ast, \ast, a_1, a_2, a_n, a_3, a_4, a_5, a_6, a_7) \}$

$P_2 = \{ (\ast, \ast, a_1, a_2, a_n), (\ast, \ast, a_1, a_2, a_n, a_3), (\ast, \ast, a_1, a_2, a_n, a_3, a_4), (\ast, \ast, a_1, a_2, a_n, a_3, a_4, a_5), (\ast, \ast, a_1, a_2, a_n, a_3, a_4, a_5, a_6), (\ast, \ast, a_1, a_2, a_n, a_3, a_4, a_5, a_6, a_7) \}$

With $\lambda(g) = (\lambda(a_1), \ldots, \lambda(a_{p+q}))$ we denote the tuple of the $pq$-gram’s node labels, called its label-tuple. While a $pq$-gram is unique within a tree, different $pq$-grams may yield identical label-tuples.

**Definition 3. pq-Gram Index.** Let $T$ be a tree with profile $P_T$, $p > 0$, $q > 0$. The $pq$-gram index, $I$, of tree $T$ is the bag of all label-tuples of $T$,

$I(T) = \bigcup_{g \in P_T} \lambda(g).$ (5)

We store the $pq$-gram index of a forest $F = \{ T_1, \ldots, T_N \}$ in a relation with tuples $(k, x, n)$, where $k$ is the ID of $T_k$, $x$ is a label-tuple, and $n$ is the number of occurrences of $x$. To deal with node labels of different length, such as labels in XML documents, we use a fingerprint hash function (e.g., the Karp-Rabin fingerprint function [10]) that maps a label $l$ to a hash value $h(l)$ of fixed length that is unique with a high probability. Instead of storing the label-tuples of $pq$-grams, we store the concatenation of the hashed labels (see Figure 4). Note that the only operation we need to perform on labels is to check equality.

**Example 3.** Figure 4 shows part of the $pq$-gram index for tree $T_0$, $p = q = 3$. The label-tuple with the hash values $012000$ occurs twice in $T_0$, in the $pq$-grams $(\ast, a_1, a_2, a_3, a_4, a_5)$ and $(\ast, a_1, a_2, a_3, a_4, a_5)$. All other label-tuples are unique.

An approximate lookup of a search tree $X$ in a forest $F$ returns all trees of the forest that are similar to the search tree, i.e., the set $\{ T \in F | TDist(X, T) < \tau \}$, where $TDist$ is a distance measure between trees and $\tau$ is a threshold value. We use the $pq$-gram distance [2] as a measure for the similarity of two trees. The $pq$-gram distance is based on the number of $pq$-grams that the indexes of the compared trees have in common. For two trees, $T$ and $T'$, the $pq$-gram distance is defined as $dist^{Pq}(T, T') = 1 - \frac{\|I(T)||I(T')\|}{\|I(T)||I(T')\|}$.

![Figure 4: (a) Hash Function, (b) pq-Gram Index.](image)
4. OUTLINE

In the following we give an outline of our approach to incrementally update the index. Figure 5 shows the application scenario and summarizes the solution:

**Input:** The old index, \( I_0 \), the log of inverse edit operations, \((e_1, \ldots, e_n)\), and the resulting tree, \( T_n \) (shaded in Figure 5).

**Output:** The new index, \( I_n \), for tree \( T_n \).

**Solution:** The solution consists of three steps:

\[
\Delta^+_n = \delta(T_n, e_1) \cup \cdots \cup \delta(T_n, e_n)
\]

\[
\Delta^-_n = U(\cdots U(U(\Delta^+_n, e_n), e_{n-1}), \ldots, e_1)
\]

\[
I_n = I_0 \setminus \lambda(\Delta^+_n) \uplus \lambda(\Delta^-_n)
\]

First, we compute \( \Delta^+_n \), the new \( pq \)-grams in the profile of \( T_n \) that were not present in the profile of \( T_0 \). Second, we compute the set \( \Delta^-_n \), the old \( pq \)-grams in the profile of \( T_0 \) that are not present in the profile of \( T_n \). \( \delta(T_n, e_j) \) operates on tree \( T_n \) and uses the reverse edit operation \( \bar{e}_j \) to compute the new \( pq \)-grams. \( U(\delta(T_n, e_j), \bar{e}_j) \) operates on the new \( pq \)-grams and transforms them into the old \( pq \)-grams. Finally, we map the \( pq \)-grams in \( \Delta^+_n \) and \( \Delta^-_n \) to label-tuples and update the index \( I_0 \).

Note the difference between the profile and the index of a tree. The profile, \( P_i \), is a set of \( pq \)-grams, the index, \( I = \lambda(P) \), the respective bag of label-tuples. While the index can be computed from the profile, the reverse is not possible. As we need to distinguish between different nodes with the same label, we compute the deltas on the profiles.

5. SINGLE EDIT STEP

In this section we discuss the effect of a single edit operation on the profile of a tree. Figure 6 graphically illustrates this for two trees \( T_i \) and \( T_j \) with profiles \( P_i \) and \( P_j \), respectively, and an edit operation, \( e_j \), such that \( T_j = e_j(T_i) \). An edit operation changes a small part of the profile by substituting some old \( pq \)-grams (A) by new \( pq \)-grams (B). A substantial part of the profiles overlaps (C). The old \( pq \)-grams exist only in \( P_i \), the new \( pq \)-grams only in \( P_j \).

We give declarative definitions for functions that return the old and the new \( pq \)-grams. Algorithms for these functions will be given in Section 7 and 8.

**Figure 6:** Profile Update for an Edit Operation \( e_j \).

### 5.1 The Delta Function

Assume \( T_s, T_j, e_j \) such that \( T_j = e_j(T_s) \). The delta function, \( \delta(T_j, e_j) \), operates on \( T_j \) and computes the new \( pq \)-grams that have been added by the edit operation \( e_j \).

**Definition 4.** Delta Function. Let \( T_j \) be a tree with profile \( P_j \). Let \( e_j \) be an edit operation and \( \bar{e}_j \) its reverse operation. The delta function is defined as

\[
\delta(T_j, e_j) = \begin{cases} P_j \setminus P_i & \text{iff } \exists T_i : T_i = \bar{e}_j(T_j) \\ \emptyset & \text{otherwise} \end{cases}
\]

\( P_i \) is the profile of \( T_i \).

This definition allows us to compute the delta function even if the edit operation is not defined for the tree (e.g., deletion of a node that is not in the tree). This is crucial in our application, where only the resulting tree, \( T_n \), is given. We will compute the delta function on \( T_n \) for all reverse edit operations in the log. The reverse edit operations in the log are defined on intermediate trees that are different from the resulting tree. They are not guaranteed to be defined on \( T_n \). We further discuss this issue in Section 6.

For the rename (delete) operation the delta function returns all \( pq \)-grams that contain the renamed (deleted) node, for the insert operation the \( pq \)-grams that contain the parent and at least one of the children of the inserted node.

**Lemma 1.** Let \( T_i, T_j \) be trees such that \( T_i = \bar{e}_j(T_j) \), and let \( g \in P_j \) be a \( pq \)-gram with the nodes \( N(g) \). If \( \bar{e}_j = \text{ins}(n,v,k,m) \), \( C = \{e_k, \ldots, e_m\} \), where \( e_i \) is the \( i \)-th child of \( v \), then

\[
g \in \delta(T_j, e_j) \Leftrightarrow v \in N(g) \land \exists c \in C : c \in N(g).\]

If \( \bar{e}_j = \text{del}(n) \) or \( \bar{e}_j = \text{ren}(n,l) \), then

\[
g \in \delta(T_j, e_j) \Leftrightarrow n \in N(g).\]

**Proof.** Each \( pq \)-gram \( g \in P_j \) is a subtree of \( T_j \). If and only if this subtree is affected by the edit operation \( \bar{e}_j \), the \( pq \)-gram is new, i.e., \( g \in \delta(T_j, e_j) \).

**Insert.** \( g \in \delta(T_j, e_j) \Leftrightarrow v \in N(g) \land \exists c \in C : c \in N(g) \) is equivalent to \( v \notin N(g) \lor \forall c \in C : c \notin N(g) \Rightarrow g \notin \delta(T_j, e_j) \): If \( v \notin N(g) \), either (a) \( n \) or (b) all nodes of \( g \) are in the subtree rooted in \( v \). If (a), \( g \) is outside the affected subtree. If (b), a descendant of \( v \) is the root of \( g \), and the inserted node is above its reach. \( g \in \delta(T_j, e_j) \Leftrightarrow v \in N(g) \land \exists c \in C : c \in N(g) \). As \( n \) is inserted between \( v \) and \( c \), all \( pq \)-grams that contain both of them are affected.

**Delete.** \( g \in \delta(T_j, e_j) \Leftrightarrow n \in N(g) \) is equivalent to \( n \notin N(g) \Rightarrow g \notin \delta(T_j, e_j) \): If \( n \) is not in \( g \), no node of \( g \) is affected. \( g \in \delta(T_j, e_j) \Leftrightarrow n \in N(g) \): \( n \) does not exist in \( T_i \). If \( n \) is in \( g \), \( g \) is only in \( P_j \).
If $n \notin N(g)$ then $g \notin \delta(T_j, e_j)$: If $n$ is not in $g$, no node of $g$ is affected. $g \in \delta(T_j, e_j)$ if $n \in N(g)$, then $n \in N(g)$: $\lambda(n) = l$ in $T_1$, but $\lambda(n) \neq l$ in $T_2$. As $g \in P_2$, $\lambda(n) \neq l$ in $g$. Thus, if $n$ is in $g$, $g$ is only in $P_2$.

5.2 The Profile Update Function
There is a symmetry between an edit operation and its reverse: The new pq-grams of the edit operation correspond to the old pq-grams of the reverse edit operations and vice versa. If $T_j = e_j(T_i)$, then $\delta(T_j, e_j)$ denotes the pq-grams that are added by $e_j$, and $\delta(T_i, e_j)$ denotes the pq-grams that are deleted by $e_j$ (Figure 6). Since $T_1$ is not available after the update we define the profile update function, which transforms the new pq-grams into the old pq-grams. As an input we allow a superset of the new pq-grams. This will be relevant for the extension to a sequence of edit operations.

In the output the new pq-grams are replaced by the old pq-grams, all other pq-grams are not affected.

Definition 5. Profile Update Function. Let $T_i, T_j$ be trees with profiles $P_i, P_j$, respectively, and $e_j$ be an edit operation and $e_j$ its reverse operation such that $T_j = e_j(T_i)$, and let $\delta(T_j, e_j) \subseteq P_j \subseteq P_j$. The profile update function, $U: 2^{R_j} \rightarrow 2^{Q_j}$, is defined as follows:

$$U(p_j, e_j) = p_j \setminus \delta(T_j, e_j) \cup \delta(T_i, e_j)$$

(9)

If $p_j = \delta(T_j, e_j)$, the profile update function computes the old pq-grams from the new pq-grams, i.e., $\delta(T_j, e_j) = U(\delta(T_j, e_j), e_j)$. If $p_j = P_j$, the original profile $P_j$ is computed from $P_i$. Due to the symmetry of the scenario also the opposite direction holds:

$$P_i = U(P_j, e_j) \quad P_j = U(P_i, e_j)$$

(10)

6. EDIT SEQUENCE
In this section we extend the results of the previous section to a sequence of edit operations. We begin with basic definitions and an intuitive illustration of the overall update process, followed by formal proofs.

6.1 Incremental Index Update
Consider a sequence of edit operations as shown in Figure 5. $\Delta_n$ denotes the new pq-grams in $P_n$ that were not present in $P_0$ and have been introduced by one of the edit operations. $\Delta_n$ denotes the old pq-grams in $P_0$ that have been removed by one of the edit operations and, hence, are not present in $P_n$.

Definition 6. Let $T_0, \ldots, T_n$ be trees with profiles $P_0, \ldots, P_n$, respectively, where $T_0$ has been transformed into $T_n$ by a sequence of edit operations $(e_1, \ldots, e_n)$, i.e., $T_k = e_k(T_{k-1})$ for $1 \leq k \leq n$. We define the following sets of pq-grams:

Invariants pq-grams: $C_n = P_0 \cap \cdots \cap P_n$ (11)

Old pq-grams: $\Delta_n = P_0 \setminus C_n$

New pq-grams: $\Delta_n^+ = P_n \setminus C_n$ (12)

Figure 7 illustrates these sets for a scenario with $n = 2$. The two shaded regions in Figure 7(a) together form the set $\Delta_n^+$, i.e., the new pq-grams in $P_2$ that were not present in $P_0$. Note that there might exist new pq-grams that have been added by an edit operation but are not contained in the final profile $P_2$, since they have been removed by a subsequent edit operation. Hence, $\Delta_n^+$ is in general a subset of all new pq-grams that have been introduced by edit operations. $C_2$ is the set of pq-grams that are shared by all trees.

Having determined the set $\Delta_n^+$, we recursively apply the profile update function for each reverse edit operation in the log-file: first for $e_n$, then for $e_{n-1}$, etc. This process transforms $\Delta_n^+$ into the set $\Delta_{n-1}^+$ of old pq-grams that have been dropped from $P_0$ by one of the edit operations. Figure 7(b-c) show this transformation of $\Delta_n^+$ into $\Delta_{n-1}^+$. The first call of the update function considers the edit operation $e_2$ and substitutes the new pq-grams in $\Delta_n^+$ that have been introduced by $e_2$. The resulting set of pq-grams is illustrated in Figure 7(b) and is passed to the next call of the profile update function. Figure 7(c) shows the final set $\Delta_1^+$ of old pq-grams that have been removed from $P_0$.

The last step is to map the old and new pq-grams to the corresponding label-tuples and update the index.

Lemma 2. Let $T_0$ be a tree with index $I_0 = \lambda(P_0)$ that is transformed to $T_n$ with index $I_n = \lambda(P_n)$ by a sequence of pq operations. The new index, $I_n$, can be computed from the old index, $I_0$, as follows:

$$I_n = I_0 \setminus \lambda(\Delta_1^+) \cup \lambda(\Delta_n^+)$$

(13)

Proof. First we show that replacing the old by the new pq-grams in $P_0$ results in $P_n$: $P_0 \setminus \Delta_0^+ \supseteq P_0 \setminus \Delta_1^+ \supseteq \cdots \supseteq P_0 \setminus \Delta_n^+$. Then $P_0 \setminus \Delta_n^+ = P_0 \setminus \Delta_n^+ \supseteq \bigcup_{i=0}^{n} \lambda(\Delta_i^+) = \lambda(P_n) \setminus \lambda(\Delta_n^+)$. This shows that $\lambda(\Delta_0^+) \subseteq \lambda(\Delta_1^+) \subseteq \cdots \subseteq \lambda(\Delta_n^+)$. Then $\lambda(\Delta_n^+) \subseteq \lambda(\Delta_1^+)$ is subtracted from $\lambda(P_0)$. As $\Delta_n^+ \subseteq \lambda(\Delta_1^+)$, also $\lambda(\Delta_n^+) \subseteq \lambda(\Delta_1^+)$.

6.2 Deltas of Intermediate Tree Versions
For the computation of $\Delta_n^+$ and $\Delta_n^+$ we have to analyze how the pq-grams have evolved in the individual edit steps. With the functions defined in the previous section we can compute the old and new pq-grams for the last edit operation. This step cannot be repeated for earlier edit operations, as we have no access to the intermediate tree versions.
The delta functions evaluated on the intermediate tree versions give us the $pq$-grams that have been introduced during the edit process. We consider the tree $T_i$ that is transformed to $T_j$ by the edit operation $e_j$, and an edit operation of the log, $\bar{e}_x$. $\bar{e}_x$ reverses an earlier operation in the process that produced $T_x$ (see Figure 8). The delta function for $\bar{e}_x$ is defined on $T_j$ as well as on $T_x$, but the results on $T_x$ and $T_j$ are different, as the trees differ in structure and labels. $\delta(T_j, \bar{e}_x)$ computes the new $pq$-grams for the edit operation $e_x$ that transforms $\bar{e}_x(T_j)$ into $T_j$. $\bar{e}_x(T_j)$ is not a tree in our scenario.

We compute the delta function for the earlier edit operation on both, $T_i$ and $T_j$. We analyze, how $e_j$ affects the results of the delta function. The following lemma shows that the result is the same, except for the $pq$-grams that are replaced by $e_j$. This has an important implication on our application: The delta computed on $T_n$ for an earlier edit operation, $\bar{e}_x$, contains all $pq$-grams of the delta on $T_x$ that where not affected by a later edit operation.

**Lemma 3.** Let $e_j$ be an edit operation that transforms $T_i$ into $T_j$ (see Figure 8). For an edit operation $\bar{e}_x$ that transforms $T_i$ to $\bar{e}_x(T_i)$ and $T_j$ to $\bar{e}_x(T_j)$,

$$
\delta(T_i, \bar{e}_x) \backslash \delta(T_i, e_j) = \delta(T_j, \bar{e}_x) \backslash \delta(T_j, e_j).
$$

Note that $\delta(T_i, e_j) = \mathcal{U}(\delta(T_i, \bar{e}_x), e_j)$ are the old, $\delta(T_j, e_j)$ the new $pq$-grams of $e_j$.

**Proof.** (14) is equivalent to

$$
g \in \delta(T_i, \bar{e}_x) \land g \notin \delta(T_i, e_j) \Leftrightarrow g \in \delta(T_j, \bar{e}_x) \land g \notin \delta(T_j, e_j).
$$

We first show (15) from left to right and denote the left side with $L$. From $L$ follows $g \in P_i \cap P_j$, i.e., the $pq$-grams in $\delta(T_i, \bar{e}_x)$ that are not replaced by $e_j$ are also in $P_j$: $g \in \delta(T_i, \bar{e}_x) \Rightarrow g \in P_i \Rightarrow g \in \delta(T_i, \bar{e}_x) \subseteq P_i$ (6); $g \notin \delta(T_i, e_j) \Rightarrow g \notin \delta(T_i, e_j) \Rightarrow g \notin P_i \cap P_j$, as $T_i = \delta(T_i, e_j); g \notin \delta(T_i, e_j); g \notin \delta(T_i, \bar{e}_x)$ (6); from $g \in P_i$ and $P_j$ follow $g \in P_i \cap P_j$. We distinguish for $\bar{e}_x$:

**Rename.** We first show $L \Rightarrow g \notin \delta(T_j, \bar{e}_x): g \in P_i \cap P_j$ implies $g \notin \delta(T_j, \bar{e}_x)$, as $\delta(T_j, \bar{e}_x) = P_j \backslash P_i$ (6). Now we show $L \Rightarrow g \in \delta(T_j, \bar{e}_x): L$ implies that the renamed node $n$ is a node of $g (g \in \delta(T_i, \bar{e}_x) \Rightarrow n \in N(g))$. As $g$ is in $P_i \cap P_j$ and $P_j$ contains the node renamed by $\bar{e}_x$, it is a new $pq$-gram of $P_i$ with respect to $e_x$: $n \in N(g) \land g \in P_i \Rightarrow g \in \delta(T_i, \bar{e}_x)$ (8).

**Delete.** Same rationale as for rename.

**Insert.** Similar rationale as for rename. Let $v$ be the parent of the inserted node $n$, then its children $C = \{c_k, \ldots, c_m\}$ move under $n$. We show $L \Rightarrow g \notin \delta(T_j, \bar{e}_x): L \Rightarrow g \in P_i \cap P_j$ implies $g \notin \delta(T_j, \bar{e}_x)$. We show $L \Rightarrow g \notin \delta(T_j, \bar{e}_x): L$ implies that (a) the parent of the inserted node and at least one of its children are in $g (g \in \delta(T_i, \bar{e}_x) \Rightarrow v \in N(g) \land \exists c \in C: c \in N(g))$, and (b) that $g \in P_i$, $L \Rightarrow g \in P_i \cap P_j$. With (a), (b): $v \in N(g) \land \exists c \in C: c \in N(g) \land \exists e \in P_j \Rightarrow g \in \delta(T_j, \bar{e}_x).

(15) from right to left follows from the symmetry of $e_j$ and $e_j$, by substituting $e_j$ with $\bar{e}_x$ and vice versa. □

### 6.3 Computing $\Delta^n_+$

In this section we show that the new $pq$-grams, $\Delta^n_+$, can be computed on the tree $T_n$, by evaluating the delta function for each edit operation in the log on the tree $T_n$ and by taking the union of the results, i.e., $\Delta^n_+ = \bigcup_{k=1}^n \delta(T_k, e_k)$. $\Delta^n_+$ does not necessarily contain all new $pq$-grams that have been introduced by an edit operation. Some new $pq$-grams of one edit operation may be removed by a later operation. $\Delta^n_+$ is the set of new $pq$-grams that are present in $P_n$. It is equal to or a subset of all new $pq$-grams, as illustrated in Figure 7 and formalized in the following theorem. We break the proof down into three parts and formulate each part in an individual lemma. The proof of the theorem references the lemmas and connects the parts.

**Lemma 4.** Let $L = (e_1, \ldots, e_n)$ be a sequence of edit operations that transforms $T_0$ into $T_n$, $T_i = e_i(T_{i-1})$, $1 \leq i \leq n$.

$$
P_i = P_0 \backslash \bigcup_{k=1}^i \delta(T_{k-1}, e_k) \cup \bigcup_{k=1}^i \delta(T_k, e_k)
$$

**Proof.** (i) True for $P_1$. (ii) With $A_i = \bigcup_{k=1}^i \delta(T_{k-1}, e_k)$ and $B_i = \bigcup_{k=1}^i \delta(T_k, e_k)$ the induction hypothesis is

$$
P_i = P_0 \backslash A_i \cup B_i \Rightarrow P_{i+1} = P_0 \backslash A_{i+1} \cup B_{i+1}.
$$

**Lemma 5.** Let $L = (e_1, \ldots, e_n)$ be a sequence of edit operations that transforms $T_0$ into $T_n$, $T_i = e_i(T_{i-1})$, $1 \leq i \leq n$. Let $A_n = \bigcup_{k=1}^n \delta(T_{k-1}, e_k)$. Then

$$
A_n = P_0 \backslash A_n.
$$

**Proof.** (a) $P_0 \backslash A_n \supseteq C_n$: $C_n = P_0 \cap \bigcap_{k=1}^n P_k = P_0 \cap \bigcap_{k=1}^n \left(\bigcup_{e_k \in P_k} \delta(T_k, e_k) \cup \delta(T_k, e_k)\right)$. As $\delta(T_k, e_k) \cap \delta(T_k, e_k) = \emptyset$, $C_n = P_0 \cap \bigcap_{k=1}^n \left(\delta(T_k, e_k) \cup \delta(T_k, e_k)\right) = C_n \cap A_n = \emptyset$.

(b) $P_0 \backslash A_n \subseteq C_n$: The opposite, $g \in P_0 \backslash A_n$, and $g \notin C_n$, leads to a contradiction: $g \notin C_n \Rightarrow g \not\in P_0 \lor g \in \bigcup_{k=1}^n P_k \Rightarrow g \not\in \bigcup_{k=1}^n \left(\delta(T_{k-1}, e_k) \cup \delta(T_k, e_k)\right)$. However, by induction we show that $\forall g, g \in P_0 \Rightarrow g \in P_0$ is true. $g \in P_0 \Rightarrow g \in P_{i+1}, 0 \leq i \leq n - 1$: $P_{i+1} = P_0 \backslash \delta(T_i, e_i) \cup \delta(T_i, e_i)$: $g \in P_0 \backslash A_n \Rightarrow g \notin A_n \Rightarrow g \in P_{i+1}$. □
Lemma 6. Let \( L = (e_1, \ldots, e_n) \) be a sequence of edit operations that transforms \( T_0 \) into \( T_n \). Let \( B_1 = \bigcup_{k=1}^{n} \delta(T_i, e_k) \). Then

\[
B_n \cap C_n = \emptyset.
\]

Proof. Proof by induction. (i) True for \( i = 1 \): \( B_1 = \delta(T_1, e_1) \Rightarrow B_1 \cap P_0 = \emptyset \).

(ii) Induction hypothesis:

\[
B_i \cap C_i = \emptyset \Rightarrow B_{i+1} \cap C_i = \emptyset.
\]

We show \( B_{i+1} \cap C_i \subseteq \delta(T_{i+1}, e_{i+1}) \cap C_i \). \( B_{i+1} \cap C_i = [B_i \cap \delta(T_{i+1}, e_{i+1})] \cap C_i \subseteq [B_i \cap \delta(T_{i+1}, e_{i+1})] \cap C_i \). Then it follows with \( \delta(T_{i+1}, e_{i+1}) \cap C_i = \emptyset \) that \( B_{i+1} \cap C_i = \emptyset \). \( \Box \)

Theorem 1. Let \( L = (e_1, \ldots, e_n) \) be a sequence of edit operations that transforms \( T_0 \) into \( T_n \). Let \( T_i = e_i(T_{i-1}) \), \( 1 \leq i \leq n \). The set of new \( pq \)-grams, \( \Delta_n \), can be computed as

\[
\Delta_n^+ = \bigcup_{k=1}^{n} \delta(T_i, e_k).
\]

Proof. With Lemma 4, \( P_n \) can be expressed as

\[
P_n = P_0 \setminus A_n \cup B_n.
\]

If we look at the scenario in the reverse direction (\( T_n \) is transformed to \( T_0 \) by a sequence of edit operations, \( (e_n, \ldots, e_1) \)), then \( \Delta_n^- \) in the reverse scenario corresponds to \( \Delta_n \) in the original scenario. Thus in the original scenario \( \Delta_n^- = \bigcup_{k=1}^{n} \delta(T_n, e_k) \). As \( T_0 \) is not given, we cannot use this approach to compute \( \Delta_n^- \).

6.4 Computing \( \Delta_n^- \)

For two trees, \( T_i = e_j(T_{i-1}) \), the profile update function computes \( P_i \) from \( P_{i-1}, P_i = U(P_{i-1}, e_j) \) (10). Thus, we can compute \( P_0 \) from \( P_n \) by applying the update function recursively, \( P_0 = U(\ldots U(U(P_{i-1}, e_n, e_{n-1}), e_{n-2}), \ldots, e_1) \). Recall that \( \Delta_n^- = P_0 \setminus C_0 \) is a subset of \( P_0 \) and \( \Delta_n^+ = P_n \setminus C_n \) is a subset of \( P_n \). (12) In this section we show that, similar to \( P_0 \) and \( P_n \), we can compute \( \Delta_n \) from \( \Delta_n^+ \) by applying the update function recursively to \( \Delta_n^+ \),

\[
\Delta_n^- = U(\ldots U(U(U(\Delta_n^+, e_n), e_{n-1}), \ldots, e_1))
\]

We will use the following Lemma 7 to rewrite the recursive updates in an un-nested form.

Lemma 7. Let \( \Delta_i^+ \) be the result of iteratively applying the profile update function to \( \Delta_i^+ \) \( i \) times, \( 1 \leq i \leq n \),

\[
\Delta_i^+ = U(\ldots U(U(U(\Delta_i^+, e_n), e_{n-1}), \ldots, e_{i+1})).
\]

Then \( \Delta_n^+ \) can be written in un-nested form as

\[
\Delta_n^+ = \bigcup_{k=1}^{n-i} \delta(T_{n-i}, e_k) \cup \bigcup_{k=n-i+1}^{n} \delta(T_{n-i}, e_k).
\]

Proof. We define \( A_i^* = \bigcup_{k=1}^{n-i} \delta(T_{n-i}, e_k) \) and \( B_i^* = \bigcup_{k=n-i+1}^{n} \delta(T_{n-i}, e_k) \), and show (24) by induction:

(i) \( \Delta_1^+ \) computed with (23) and (24) matches: \( \Delta_1^+ \equiv \bigcup_{k=1}^{n-1} \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \). \( \Delta_1^+ \equiv U(A_1^* \cup B_1^*) \equiv U(A_1^* \cup B_1^* \cup e_{n-i}) = \bigcup_{k=1}^{n-i-1} \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \).

(ii) Induction hypothesis:

\[
\Delta_i^+ = A_i^* \cup B_i^* \Rightarrow \Delta_i^{i+1} = A_i^{i+1} \cup B_i^{i+1}
\]

Proof. We define \( A_i^* = \bigcup_{k=1}^{n-i} \delta(T_{n-i}, e_k) \) and \( B_i^* = \bigcup_{k=n-i+1}^{n} \delta(T_{n-i}, e_k) \), and show (24) by induction:

(i) \( \Delta_1^+ \) computed with (23) and (24) matches: \( \Delta_1^+ \equiv \bigcup_{k=1}^{n-1} \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \). \( \Delta_1^+ \equiv U(A_1^* \cup B_1^*) \equiv U(A_1^* \cup B_1^* \cup e_{n-i}) = \bigcup_{k=1}^{n-i-1} \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \).

(ii) Induction hypothesis:

\[
\Delta_i^+ = A_i^* \cup B_i^* \Rightarrow \Delta_i^{i+1} = A_i^{i+1} \cup B_i^{i+1}
\]

Proof. We define \( A_i^* = \bigcup_{k=1}^{n-i} \delta(T_{n-i}, e_k) \) and \( B_i^* = \bigcup_{k=n-i+1}^{n} \delta(T_{n-i}, e_k) \), and show (24) by induction:

(i) \( \Delta_1^+ \) computed with (23) and (24) matches: \( \Delta_1^+ \equiv \bigcup_{k=1}^{n-1} \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \). \( \Delta_1^+ \equiv U(A_1^* \cup B_1^*) \equiv U(A_1^* \cup B_1^* \cup e_{n-i}) = \bigcup_{k=1}^{n-i-1} \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \).

(ii) Induction hypothesis:

\[
\Delta_i^+ = A_i^* \cup B_i^* \Rightarrow \Delta_i^{i+1} = A_i^{i+1} \cup B_i^{i+1}
\]

Proof. We define \( A_i^* = \bigcup_{k=1}^{n-i} \delta(T_{n-i}, e_k) \) and \( B_i^* = \bigcup_{k=n-i+1}^{n} \delta(T_{n-i}, e_k) \), and show (24) by induction:

(i) \( \Delta_1^+ \) computed with (23) and (24) matches: \( \Delta_1^+ \equiv \bigcup_{k=1}^{n-1} \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \). \( \Delta_1^+ \equiv U(A_1^* \cup B_1^*) \equiv U(A_1^* \cup B_1^* \cup e_{n-i}) = \bigcup_{k=1}^{n-i-1} \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \cup \delta(T_{n-i}, e_k) \).

(ii) Induction hypothesis:

\[
\Delta_i^+ = A_i^* \cup B_i^* \Rightarrow \Delta_i^{i+1} = A_i^{i+1} \cup B_i^{i+1}
\]
\( C_n = C_n \) as both of them are the intersection of the same profiles. With \( P'_n = P_0 \) we get \( \Delta^{+\ast}_n = P_0 \setminus C_n = \Delta_n \). \( \square \)

7. Computing Profile Updates

In this section we introduce a matrix representation of \( pq \)-grams that better reflects our implementation, and we describe the computation of the delta and the profile update function in terms of matrix operations.

7.1 Matrix Representation of \( pq \)-Grams

For a non-leaf anchor node with \( f \) children, \( f + q - 1 \) \( pq \)-grams exist. They all have the same \( p \)-part, but different \( q \)-parts. For a leaf only one \( pq \)-gram exists, where the \( q \)-part consists of \( q \) null nodes.

Definition 7. \( p \)-Matrix and \( q \)-Matrix. Let \( T \) be a tree, \( p > 0, q > 0 \), and let \( a \in N(T) \) be a node with children \( c_1, \ldots, c_f \). The matrix, \( P(a) \), of node \( a \) is the \( 1 \times p \)-matrix that represents the \( p \)-part of \( pq \)-grams anchored in \( a \):

\[
P(a) = (a_{p-1}, \ldots, a_1, a_0)
\]

If \( a \) is a non-leaf node, i.e., \( f > 0 \), the \( q \)-matrix, \( Q(a) \), is defined as an \((f + q - 1) \times q\)-matrix that represents the \( q \)-parts of all \( pq \)-grams anchored in \( a \):

\[
Q(a) = \begin{pmatrix}
\bullet & \bullet & \cdots & \bullet & n_2 \\
\bullet & n_2 & \cdots & n_3 & n_4 \\
\cdots & \cdots & \bullet & \cdots & \cdots \\
\end{pmatrix}
\]

If \( a \) is a leaf node, i.e., \( f = 0 \), the \( q \)-matrix is defined as a \( 1 \times q \)-matrix that contains only null nodes.

The \( pq \)-grams of a node \( a \) can be computed by the concatenation of its \( p \)- and \( q \)-matrix, \( P(a) \circ Q(a) \), which concatenates the \( p \)-part in \( P \) with each \( q \)-part in \( Q \).

Example 4. We consider tree \( T_0 \) in Figure 2, assume \( p = q = 3 \), and compute all \( pq \)-grams with anchor node \( n_1 \) using the \( p \)- and \( q \)-matrices.

\[
P(n_1) \circ Q(n_1) = (\bullet, \bullet, n_1) \circ \begin{pmatrix}
\bullet & \bullet & n_2 \\
\bullet & n_2 & n_3 & n_4 \\
\end{pmatrix}
= \{(\bullet, \bullet, n_1, \bullet, n_2, n_3), (\bullet, n_1, n_2, n_3), (\bullet, n_1, n_2, n_3, \bullet), (\bullet, n_1, n_2, n_3, \bullet)\}
\]

7.2 Effective Computation of \( \delta \) and \( \mathcal{U} \)

For each edit operation we express the new \( pq \)-grams, \( \delta(T_j, \varepsilon) \), in terms of \( p \)- and \( q \)-matrices, and show, how the old \( pq \)-grams, \( \mathcal{U}(\delta(T_j, \varepsilon)) \), are computed from the new ones.

To facilitate the discussion about the computation of the profile update function, we introduce the following notation: \( desc_d(n) \) is the set of \( n \) and its descendants within distance \( d \), i.e., \( desc_d(n) = \{x | x \leq n \text{ and } \text{dist}(n, x) \leq d\} \). We use \( desc_d(n_1, \ldots, n_m) \) as an abbreviation for \( \{x | x \leq n \text{ and } n \in \{n_1, \ldots, n_m\}\} \), i.e., all descendants within distance \( d \) of a node set.

Given a \( p \)-matrix \( P(a) \), the operation \( P^{+\ast n_1}(a) \) inserts node \( n_1 \) at position \( i \), \( P^{-\ast n_1}(a) \) deletes node \( n_1 \) from \( P(a) \), and \( P^{\ast}(a) \) replaces \( a_i \) by \( m \). The other nodes in \( P(a) \) are shifted as shown in Figure 9, where \( a_i \) is \( a \)'s ancestor at distance \( i \).

The operations on \( q \)-matrices are illustrated in Figure 10. \( Q(a) \) is the \( q \)-matrix for anchor node \( a \). The (inverse) diagonals are formed by the children \( c_1, \ldots, c_f \) of \( a \), and the corners are filled with null nodes. With \( Q^{k, m}(a) \) we denote the sub-matrix of \( Q(a) \) that is formed by the rows \( k \) to \( m + q - 1 \). It contains all \( q \)-parts of the children \( c_k, \ldots, c_m \). We introduce the operator \( A/B \) that replaces all diagonals of \( A \) with the diagonals of \( B \). \( D(n) \) initializes a new \( q \)-matrix of size \( q \times q \), with the only diagonal formed by node \( n \).

For insertions and deletions of leaf nodes we define the following special cases: For the \( q \)-matrix of a leaf node \( a \) we define \( Q^{k, m}(a) = (\bullet, \ldots, \bullet) \) and \( (\bullet, \ldots, \bullet)/A = A \). If all non-diagonal elements of a matrix \( A \) are null nodes, then \( A/(\bullet, \ldots, \bullet) = (\bullet, \ldots, \bullet) \), else \( A/(\bullet, \ldots, \bullet) \) deletes all diagonals of \( A \). If a leaf node is inserted under a node \( v \), then \( m = k - 1 \) (see \( c_1 \) in Figure 2), and \( Q^{k, m}(v) \) has no diagonals. We define \( Q^{k, m}(v)/A \) to insert all diagonals of \( A \) as new diagonals in \( Q^{k, m}(v) \), and we define \( A/Q^{k, m}(v) = (\bullet, \ldots, \bullet) \).

Table 1 shows for each edit operation the \( pq \)-gram set that forms \( \delta(T_j, \varepsilon) \) and how this set is modified by the profile update function. We use the notation introduced above.

7.3 Example

Example 5. Consider the first two edit operations in Figure 1 that transform \( T_0 \) into \( T_2 \). The reverse edit operations are \( \varepsilon_1 = \text{DEL}(n_7) \) and \( \varepsilon_2 = \text{INS}(n_1, b, n_1, 2, 3) \). We determine the new \( pq \)-grams, \( \Delta^{+\ast}_1, p = q = 3 \), by evaluating...
Insert node n as the k-th child of node v: INS(n, v, k, m)

$$\delta(T_j, \bar{e}) = P(v) \circ Q^{k-m}(v) \cup P(x) \circ Q(x)$$

$$U(\delta(T_j, \bar{e}), \bar{e}) = P(v) \circ [Q^{k-m}(v) \circ D(n)] \cup P+n0(v) \circ [D(\bar{e}) \circ Q^{k-m}(v)] \cup P+n.d(x) \circ Q(x)$$

$$\forall x \in \text{desc}_{p-2}(c_k, \ldots, c_m)$$

Delete node n, DEL(n):

$$\delta(T_j, \bar{e}) = P(v) \circ Q^{k-k}(v) \cup P(x) \circ Q(x)$$

$$U(\delta(T_j, \bar{e}), \bar{e}) = P(v) \circ [Q^{k-k}(v) \circ Q(n)] \cup P-n(x) \circ Q(x)$$

$$\forall x \in \text{desc}_{p-1}(n)$$

$$v : n$$ is the k-th child of v

Rename node n to l': REN(n, l')

$$\delta(T_j, \bar{e}) = P(v) \circ Q^{k-k}(v) \cup P(x) \circ Q(x)$$

$$U(\delta(T_j, \bar{e}), \bar{e}) = P(v) \circ [Q^{k-k}(v) \circ D(m)] \cup P/m(x) \circ Q(x)$$

$$\forall x \in \text{desc}_{p-1}(n)$$

$$m = (\text{id}(n), l')$$

$$v : n$$ is the k-th child of v

---

Table 1: Computation of the Delta Function and the Profile Update Function.

<table>
<thead>
<tr>
<th>$Q(n_1)$</th>
<th>$Q^{2-3}(n_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Q(n_1)" /></td>
<td><img src="image" alt="Q^{2-3}(n_1)" /></td>
</tr>
</tbody>
</table>

Figure 11: $q$-Matrices for Node Insertion (Example).

---

8. IMPLEMENTATION

8.1 Temporary Storage of the Deltas

We process logs with thousands of edit operations. Each edit operation of the log adds pq-grams to $\Delta_n^+$ (see Algorithm 2). We store the p-parts and q-parts of these pq-grams in a pair (P, Q) of temporary tables. Since p-parts that appear in many pq-grams are stored only once, we gain performance when we have to update them. The update function (see Algorithm 3) is applied to (P, Q) for each edit operation in the log and, step by step, transforms it to $\Delta_n^+$. We prevent duplicates from being inserted into P and Q, and we join them to reconstruct the pq-grams. An index on the anchor IDs proved to give a substantial performance advantage.

Let $P(n)$ be the p-part of the pq-grams with anchor node n, where n is the k-th child of its parent v. We store $P(n)$ as a tuple (n, k, v, h(P(n))) in P, where h() is the hash function introduced in Section 3. Let $Q(n)$ be the q-matrix of anchor node n. We store the i-th row of $Q(n)$, $r_i$, as a tuple (n, i, h(r_i)) in Q. For the pq-grams stored in the table pair (P, Q), we compute the respective label-tuples as

$$\lambda(P, Q) = \pi_{\text{pappart}}(P \otimes Q).$$

(31)

Subsequently, given pairs of tables we use the notation...
the children $k - q + 1$ to $m + q - 1$ of $n$. $Q(n)$ by accessing all children of $n$. We use the functions $P_r(n)$, $Q_{k,m}^-(n)$ and $Q_{r}(n)$ that operate on $T$ and return the respective matrices as tuples for the temporary tables $P$ and $Q$, as shown in Section 8.1.

Algorithm 2: $\delta(T, e)$

1. If $(e = \text{REN}(n, l')) \lor (e = \text{DEL}(n))$ then
2. \quad $v \leftarrow \text{parent of } n$;
3. \quad $k \leftarrow \text{sibling position of } n \text{ (is the } k\text{-th child of } v)$;
4.\quad $(P, Q) \leftarrow (P_r(n), Q_{k,m}^-(n))$;
5.\quad $\text{foreach } x \in \text{desc}_{c} - 1(n) \text{ do}$
6.\quad\quad $(P, Q) \leftarrow (P, Q) \cup (P_r(x), Q_r(x))$
7. end
8. else if $e = \text{INS}(n, v, k, m)$ then
9.\quad $(P, Q) \leftarrow (P_r(n), Q_{k,m}^+(n))$;
10.\quad $\text{foreach } c \in \{c_k, \ldots, c_m\} \text{ of } v \text{ do}$
11.\quad\quad $x \in \text{desc}_{c - 1}(c) \text{ do}$
12.\quad\quad\quad $(P, Q) \leftarrow (P, Q) \cup (P_r(x), Q_r(x))$
13. end
14. end
15. end
16. return $(P, Q)$

8.4 Implementation of the Update Function

The profile update function for $\bar{e}$ replaces $\delta(T, \bar{e})$ of $e$ in a set of $pp$-grams by $U(\delta(T, \bar{e}), e)$. The $pp$-grams are stored in the temporary tables $P$ and $Q$. The first step is to read the $p$-parts and $q$-parts of $\delta(T, \bar{e})$ from these tables. As shown in Table 1, the $q$-parts of $\delta(T, \bar{e})$ are expressed by $Q(n)$ and $Q_{k,m}^-(n)$. We implement these functions as follows:

- $Q(n) \leftarrow \sigma_{\text{anchId}=n}(Q)$
- $Q_{k,m}^-(n) \leftarrow \sigma_{\text{anchId}=n, k \leq \text{row} \leq m + q - 1}(Q)$

$Q_{k,m}^-(n)$ and $Q(n)$ return tuples $(n, i, \text{qpart})$, where $\text{qpart}$ is the $i$-th row of $Q(n)$. Different from $Q_{k,m}^+(n)$ and $Q_r(n)$ in the previous section, they operate on profiles, not on trees.

In the second step we modify $\delta(T, \bar{e})$ to get $U(\delta(T, \bar{e}), e)$. We implement the operator $A \bowtie B$ so it operates on $q$-matrices represented as $(\text{anchId}, \text{row}, \text{qpart})$ tuples and returns the result in this form. The anchor node and the first row number of the result are both determined by the first argument, $A$. The matrix operation itself is straightforward. $\mathcal{D}_n(n)$ initializes a new $q$-matrix with anchor node $a$ and a single diagonal formed by $n$.

For the update of the $p$-parts we use the function $\text{changePParts}(P, n, s, d)$ (see Algorithm 4). It implements the operators on $P(a)$ ($P_{k,m}^+, P_{k,m}^-, P_{k,m}^0$) as concatenations of strings. For each edit operation we construct a string $s$. The last $p - i$ characters of $s$ correspond to the changing part of $P(a)$ (shaded in Figure 9). We concatenate it to the invariant part of length $i$ (line 5). The $p$-parts are retrieved level by level (line 6). $P_{\text{new}}$ returns all $p$-parts of $P$ whose anchor node is $n$ or a descendant of $n$ within distance $d$. $P_{\text{new}}$ is the same set of tuples with the updated values for $p$part.

If rows are deleted from/inserted into the $q$-matrix, the row numbers, $\text{row}$, of the subsequent rows need to be updated. If $p$-parts are deleted or inserted, the sibling numbers, $\text{sibPos}$, in the $p$-parts of the subsequent siblings have to be updated. In both cases the scope of the update query...
We do a lookup in three different collections of XML documents. They have a similar overall number of nodes (approx. $50 \times 10^6$). The number of documents in the collections varies from 31 to 1999. The trees within a collection are of similar size. We measure the wall clock time for the approximate lookup of an XML document.

Figure 13 (left) shows the results for the different data sets. The lookup time with precomputed index is independent of the number of trees in the forest. If the index has to be created on the fly, the lookup time grows for larger tree numbers. Without precomputed index, the index creation is clearly the most expensive operation in the lookup process.

![Figure 13: Lookup and Update Time.](image)

### 9.2 Updating the Index

Each edit operation affects a subset of the $pq$-grams in the index. We expect that updating only the affected $pq$-grams is more efficient than building the whole index from scratch.

The computation time for index rebuilding is expected to grow with the tree size, while the one for updates depends mainly on the number of edit operations.

Figure 13 (right) compares the computation times for building the $pq$-gram index from scratch with updating it based on a log of edit operations. While the index creation time is linear in the tree size (note the log scale of the y axis), the index update time is nearly independent of the tree size. The figure shows the results for trees with up to $27 \times 10^6$ nodes.

### 9.3 Index Size

The index does not store the labels, but only their hash values. Further a $pq$-gram that appears many times in the index is stored only once. In Figure 14 (left) we compare the size of the index with the tree size. The index for both, 1, 2- and 3, 3-grams, is significantly smaller than the tree.

The tree size is linear in the number of nodes, while the index size is less than linear. We explain this with the higher probability of having duplicate $pq$-grams with larger trees.

![Figure 14: Size and Update Time of Index.](image)
9.4 Experiments with Real World Data

We compute the index and perform updates on the DBLP dataset (211MB file size, 11M nodes). From Figure 14 (right) we see that the update time is linear in the number of edit operations. Table 2 shows, for selected numbers of edit operations, the share of the various index update steps in the overall computation time. The conversion of the profile to the index ($\lambda$) is negligible. The computation times for $\Delta_+^n$ and $\Delta^-_n$ are approximately linear. The update of $I_0$ with $\lambda(\Delta_+^n)$ and $\lambda(\Delta^-_n)$ is sublinear in the number of edit operations.

<table>
<thead>
<tr>
<th>Action</th>
<th>Number of edit operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta_+^n$</td>
<td>0.642s 3.903s 37.533s 391.513s</td>
</tr>
<tr>
<td>$\Gamma^+ = \lambda(\Delta_+^n)$</td>
<td>0.184s 0.199s 0.287s 0.443s</td>
</tr>
<tr>
<td>$\Delta^-_n$</td>
<td>0.196s 2.836s 27.967s 295.104s</td>
</tr>
<tr>
<td>$\Gamma^- = \lambda(\Delta^-_n)$</td>
<td>0.177s 0.191s 0.185s 0.383s</td>
</tr>
<tr>
<td>$I_0 \setminus \Gamma^+ \cup \Gamma^-$</td>
<td>2.206s 2.770s 6.475s 19.780s</td>
</tr>
<tr>
<td>total</td>
<td>3.405s 9.900s 17.448s 707.224s</td>
</tr>
</tbody>
</table>

Table 2: Breakdown of the Index Update Time.

10. CONCLUSION

We propose an incrementally maintainable index for data with a hierarchical structure. The index uses pq-grams and we prove that the index can be updated based on the resulting document and the log of edit operations. The experimental results validate the approach for the DBLP dataset and logs with several thousand edit operations.

We process the log sequentially. Later edit operations in the log might undo earlier ones. In future we will investigate how the log can be preprocessed in order to eliminate redundant edit operations. Further the deltas that we compute span several nodes and can overlap. A preprocessing step could merge overlapping regions to optimize the computation of the deltas.

We have addressed the node edit operations rename, delete, and insert. Operations on subtrees, e.g., subtree move, insertion or deletion, are simulated by a sequence of node edit operations. Future work will investigate index updates for subtree operations.

Acknowledgements

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

11. REFERENCES


