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. These index structures must be updated if the underlying data changes. Since the performance of a full index reconstruction is prohibitive, the index must be updated incrementally.

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.

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 that supports approximate lookups in hierarchical data. The approximate lookup of a search document in a document collection returns all documents of the collection that are similar to the search document.

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.

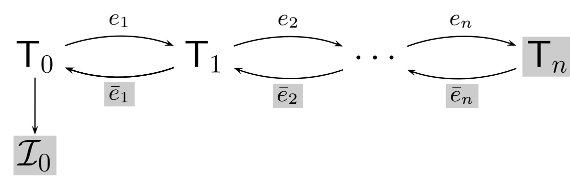

Figure 1: Application Scenario.

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.

VLDB '06, September 12-15, 2006, Seoul, Korea.

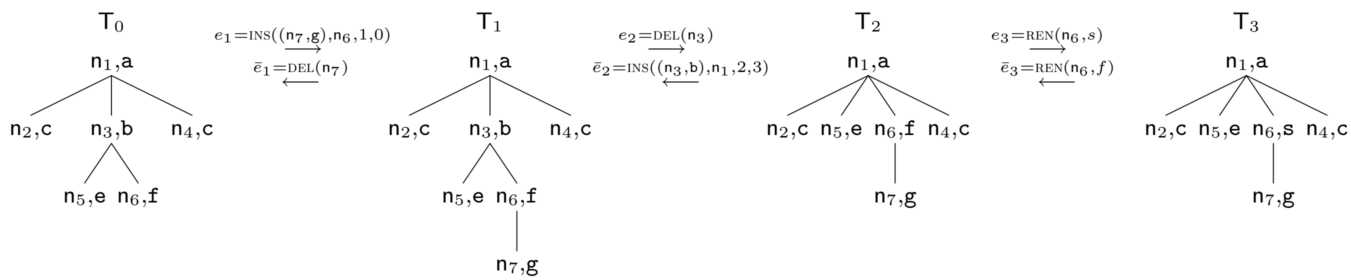

Figure 2: Sequence of Edit Operations that Transforms Tree T_0 into T_3.

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 distances 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 tree to all trees in the reference set. This step is expensive and depends on the size of the trees. We update our index locally and are nearly independent of the tree size.

The comparison of hierarchical documents has been addressed in the context of duplicate and change detection. Weis and Naumann [18] propose a framework for detecting duplicates. In change detection scenarios two versions of the same document are given and the difference is computed [4, 12]. Index use and maintenance is not addressed.

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 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 $N(\mathsf{T})$ and edges $E(\mathsf{T})$. A *node*, $\mathsf{n} \in N(\mathsf{T})$, is an (identifier, label)-pair. The identifier, $\mathrm{id}(\mathsf{n})$, is unique within the tree. The *label*, $\lambda(\mathsf{n})$, is a symbol $\sigma \in \mathbf{\Sigma}$, where $\mathbf{\Sigma}$ is a finite alphabet. A node $\bullet$ with the special label $\lambda(\bullet) = *$ is a *null node*. We represent nodes by their id or the (id, label)-pair. An *edge* is an ordered pair (v, c), where $\mathsf{v}, \mathsf{c} \in N(\mathsf{T})$ are nodes, and v is the *parent* of c. Nodes with the same parent are *siblings*. Siblings are ordered. *Contiguous* siblings $\mathsf{s}_1 < \mathsf{s}_2$ have no sibling x such that $\mathsf{s}_1 < \mathsf{x} < \mathsf{s}_2$. Node c_i is the i-th *child* of v if v is the parent of c_i and $i = |\{\mathsf{x} \in N(\mathsf{T}) : (\mathsf{v}, \mathsf{x}) \in E(\mathsf{T}), \mathsf{x} \leq \mathsf{c}_i\}|$. The number of v's children is its *fanout* f_v. The node with no parent is the *root* node, $\mathsf{r} = \mathrm{root}(\mathsf{T})$, and a node without children is a *leaf*. A *subtree* $\mathsf{S} \subseteq \mathsf{T}$ is a tree with $N(\mathsf{S}) \subseteq N(\mathsf{T})$ and $E(\mathsf{S}) \subseteq E(\mathsf{T})$ 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, $\mathsf{a} \neq \mathsf{n}$. 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 $\mathrm{dist}(\mathsf{a}, \mathsf{n}) = k$. We define $\mathrm{dist}(\mathsf{n}, \mathsf{n}) = 0$. 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 $\mathsf{T}_j = e_j(\mathsf{T}_i)$. The *inverse edit operation*, $\bar{e}_j$, undoes e_j, i.e., $\mathsf{T}_i = \bar{e}_j(\mathsf{T}_j)$. If a tree T_0 is transformed by a sequence of edit operations $(e_1, \ldots, e_n)$ into T_n, the *log* $L = (\bar{e}_1, \ldots, \bar{e}_n)$ 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 into T_j:

- $\mathrm{INS}(\mathsf{n}, \mathsf{v}, k, m)$: *Insert* a new node n as a child of node v at position k by substituting the children $\mathsf{c}_k, \mathsf{c}_{k+1}, \ldots, \mathsf{c}_m$ of v with n, and inserting them as children of n (preserving the order). The inverse edit operation is $\bar{e}_j = \mathrm{DEL}(\mathsf{n})$.

- $\mathrm{DEL}(\mathsf{n})$: *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 $\bar{e}_j = \mathrm{INS}(\mathsf{n}, \mathsf{v}, k, (k + f_\mathsf{n} - 1))$, where n is the k-th child of v in T_i, and f_n is the fanout of n.

- $\mathrm{REN}(\mathsf{n}, l')$: *Rename* a node n by changing its label l to $l' \in \mathbf{\Sigma}$, $l \neq l'$. Inverse operation: $\bar{e}_j = \mathrm{REN}(\mathsf{n}, l)$.

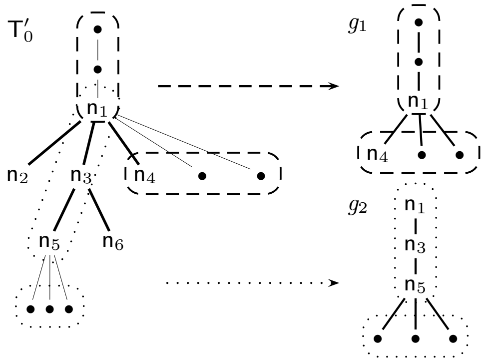

Figure 3: Part of T'_0 and Two 3,3-Grams of Tree T_0.

Throughout the paper we assume that the root node is not changed. Two nodes of different trees, T_i and T_j, are equal iff identifier and label match.

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:

$$(A \cap B) \cup (A \setminus B) = A \quad (1)$$

$$A \setminus (A \setminus B) = A \cap B \quad (2)$$

$$(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C) \quad (3)$$

$$(A \setminus B) \cup B = A \cup B \quad (4)$$

If we operate on bags, we use the symbols $\cap$, $\setminus$ and $\uplus$ 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 more similar than trees that share a low percentage.

Definition 1. *pq*-Gram. *Let T be a tree, a be a node in $N(\mathsf{T})$, $p > 0$, $q > 0$, 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 $\mathsf{a}_{p-1}, \dots, \mathsf{a}_1, \mathsf{a}$, denoted as* p-part *of g, where a_i is the ancestor of a at distance i; q contiguous children $\mathsf{c}_i, \dots, \mathsf{c}_{i+q-1}$ 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 $(\mathsf{a}_{p-1}, \dots, \mathsf{a}_1, \mathsf{a}, \mathsf{c}_i, \dots, \mathsf{c}_{i+q-1})$.

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 $g_1 = (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_4, \bullet, \bullet)$ with anchor node n_1 and $g_2 = (\mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_5, \bullet, \bullet, \bullet)$ 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, $\mathbf{P}$, *of tree T is defined as the set of all pq-grams of T.*

l	h(l)
*	0
a	1
b	3
c	2
d	6

l	h(l)
e	8
f	4
g	7
h	5
s	9

(a)

treeId	*pqg*	*cnt*
T_0	001002	1
T_0	001023	1
T_0	001232	1
T_0	001320	1
T_0	001200	1
T_0	012000	2
...	...	...

(b)

Figure 4: (a) Hash Function, (b) *pq*-Gram Index.

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

$$\begin{aligned}\mathbf{P}_0 = \{&(\bullet,\bullet,\mathsf{n}_1,\bullet,\bullet,\mathsf{n}_2), (\bullet,\bullet,\mathsf{n}_1,\bullet,\mathsf{n}_2,\mathsf{n}_3), (\bullet,\bullet,\mathsf{n}_1,\mathsf{n}_2,\mathsf{n}_3,\mathsf{n}_4),\\ &(\bullet,\bullet,\mathsf{n}_1,\mathsf{n}_3,\mathsf{n}_4,\bullet), (\bullet,\bullet,\mathsf{n}_1,\mathsf{n}_4,\bullet,\bullet), (\bullet,\mathsf{n}_1,\mathsf{n}_2,\bullet,\bullet,\bullet),\\ &(\bullet,\mathsf{n}_1,\mathsf{n}_3,\bullet,\bullet,\mathsf{n}_5), (\bullet,\mathsf{n}_1,\mathsf{n}_3,\bullet,\mathsf{n}_5,\mathsf{n}_6), (\bullet,\mathsf{n}_1,\mathsf{n}_3,\mathsf{n}_5,\mathsf{n}_6,\bullet),\\ &(\bullet,\mathsf{n}_1,\mathsf{n}_3,\mathsf{n}_6,\bullet,\bullet), (\mathsf{n}_1,\mathsf{n}_3,\mathsf{n}_5,\bullet,\bullet,\bullet), (\mathsf{n}_1,\mathsf{n}_3,\mathsf{n}_6,\bullet,\bullet,\bullet),\\ &(\bullet,\mathsf{n}_1,\mathsf{n}_4,\bullet,\bullet,\bullet)\}\\ \mathbf{P}_2 = \{&(\bullet,\bullet,\mathsf{n}_1,\bullet,\bullet,\mathsf{n}_2), (\bullet,\bullet,\mathsf{n}_1,\bullet,\mathsf{n}_2,\mathsf{n}_5), (\bullet,\bullet,\mathsf{n}_1,\mathsf{n}_2,\mathsf{n}_5,\mathsf{n}_6),\\ &(\bullet,\bullet,\mathsf{n}_1,\mathsf{n}_5,\mathsf{n}_6,\mathsf{n}_4), (\bullet,\bullet,\mathsf{n}_1,\mathsf{n}_6,\mathsf{n}_4,\bullet), (\bullet,\bullet,\mathsf{n}_1,\mathsf{n}_4,\bullet,\bullet),\\ &(\bullet,\mathsf{n}_1,\mathsf{n}_2,\bullet,\bullet,\bullet), (\bullet,\mathsf{n}_1,\mathsf{n}_5,\bullet,\bullet,\bullet), (\bullet,\mathsf{n}_1,\mathsf{n}_6,\bullet,\bullet,\mathsf{n}_7),\\ &(\bullet,\mathsf{n}_1,\mathsf{n}_6,\bullet,\mathsf{n}_7,\bullet), (\bullet,\mathsf{n}_1,\mathsf{n}_6,\mathsf{n}_7,\bullet,\bullet), (\mathsf{n}_1,\mathsf{n}_6,\mathsf{n}_7,\bullet,\bullet,\bullet),\\ &(\bullet,\mathsf{n}_1,\mathsf{n}_4,\bullet,\bullet,\bullet)\}\end{aligned}$$

With $\lambda(g) = (\lambda(\mathsf{n}_1), \dots, \lambda(\mathsf{n}_{p+q}))$ 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 $\mathbf{P}_\mathsf{T}$, $p > 0$, $q > 0$. The pq-gram index, $\mathcal{I}$, of tree T is the bag of all label-tuples of T,*

$$\mathcal{I}(\mathsf{T}) = \biguplus_{g \in \mathbf{P}_\mathsf{T}} \lambda(g). \quad (5)$$

We store the *pq*-gram index of a forest $\mathsf{F} = \{\mathsf{T}_1, \dots, \mathsf{T}_N\}$ 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 $(\bullet, \mathsf{n}_1, \mathsf{n}_2, \bullet, \bullet, \bullet)$ and $(\bullet, \mathsf{n}_1, \mathsf{n}_4, \bullet, \bullet, \bullet)$. 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 $\{\mathsf{T} \in \mathsf{F} \mid \mathrm{TDist}(\mathsf{X}, \mathsf{T}) < \tau\}$, where TDist is a distance measure between trees and τ 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 $\mathrm{dist}^{p,q}(\mathsf{T}, \mathsf{T}') = 1 - 2\frac{|\mathcal{I}(\mathsf{T}) \cap \mathcal{I}(\mathsf{T}')|}{|\mathcal{I}(\mathsf{T}) \uplus \mathcal{I}(\mathsf{T}')|}$.

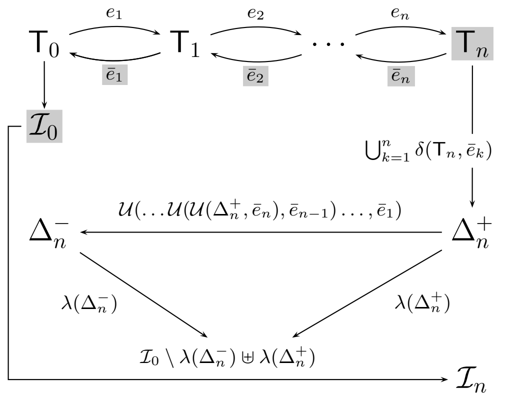

Figure 5: Application Scenario and Solution.

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, $\mathcal{I}_0$, the log of inverse edit operations, $(\bar{e}_1, \dots, \bar{e}_n)$, and the resulting tree, T_n (shaded in Figure 5).

Output: The new index, $\mathcal{I}_n$, for tree T_n.

Solution: The solution consists of three steps:

$$\begin{aligned}\Delta_n^+ &= \delta(\mathsf{T}_n, \bar{e}_1) \cup \dots \cup \delta(\mathsf{T}_n, \bar{e}_n)\\ \Delta_n^- &= \mathcal{U}(\dots\mathcal{U}(\mathcal{U}(\Delta_n^+, \bar{e}_n), \bar{e}_{n-1})\dots, \bar{e}_1)\\ \mathcal{I}_n &= \mathcal{I}_0 \setminus \lambda(\Delta_n^-) \uplus \lambda(\Delta_n^+)\end{aligned}$$

First, we compute Δ_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 Δ_n^-, the old *pq*-grams in the profile of T_0 that are not present in the profile of T_n. $\delta(\mathsf{T}_n, \bar{e}_j)$ operates on tree T_n and uses the reverse edit operation $\bar{e}_j$ to compute the new *pq*-grams. $\mathcal{U}(\delta(\mathsf{T}_n, \bar{e}_j), \bar{e}_j)$ operates on the new *pq*-grams and transforms them into the old *pq*-grams. Finally, we map the *pq*-grams in Δ_n^+ and Δ_n^- to label-tuples and update the index $\mathcal{I}_0$.

Note the difference between the profile and the index of a tree. The profile, $\mathbf{P}$, is a set of *pq*-grams, the index, $\mathcal{I} = \lambda(\mathbf{P})$, 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 $\mathbf{P}_i$ and $\mathbf{P}_j$, respectively, and an edit operation, e_j, such that $\mathsf{T}_j = e_j(\mathsf{T}_i)$. 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 $\mathbf{P}_i$, the new *pq*-grams only in $\mathbf{P}_j$.

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.

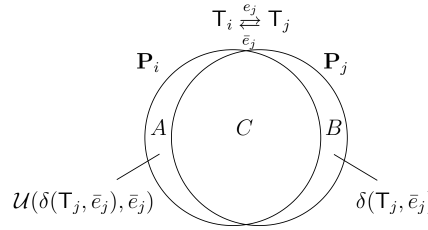

Figure 6: Profile Update for an Edit Operation $\bar{e}_j$.

5.1 The Delta Function

Assume $\mathsf{T}_i, \mathsf{T}_j, e_j$ such that $\mathsf{T}_j = e_j(\mathsf{T}_i)$. The *delta function*, $\delta(\mathsf{T}_j, \bar{e}_j)$, 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 $\mathbf{P}_j$. Let e_j be an edit operation and $\bar{e}_j$ its reverse operation. The* delta function *is defined as*

$$\delta(\mathsf{T}_j, \bar{e}_j) \quad = \quad \begin{cases}\mathbf{P}_j \setminus \mathbf{P}_i & \textit{iff } \exists \mathsf{T}_i : \mathsf{T}_i = \bar{e}_j(\mathsf{T}_j)\\ \emptyset & \textit{otherwise}\end{cases} \tag{6}$$

$\mathbf{P}_i$ 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 $\mathsf{T}_i, \mathsf{T}_j$ be trees such that $\mathsf{T}_i = \bar{e}_j(\mathsf{T}_j)$, and let $g \in \mathbf{P}_j$ be a pq-gram with the nodes $N(g)$. If $\bar{e}_j = \text{INS}(\mathsf{n}, \mathsf{v}, k, m)$, $\mathsf{C} = \{\mathsf{c}_k, \dots, \mathsf{c}_m\}$, where c_i is the i-th child of v, then*

$$g \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftrightarrow \mathsf{v} \in N(g) \wedge \exists \mathsf{c} \in \mathsf{C} : \mathsf{c} \in N(g). \tag{7}$$

If $\bar{e}_j = \text{DEL}(\mathsf{n})$ or $\bar{e}_j = \text{REN}(\mathsf{n}, l)$, then

$$g \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftrightarrow \mathsf{n} \in N(g). \tag{8}$$

PROOF. Each *pq*-gram $g \in \mathbf{P}_j$ 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., $g \in \delta(\mathsf{T}_j, \bar{e}_j)$.

Insert. $g \in \delta(\mathsf{T}_j, \bar{e}_j) \Rightarrow \mathsf{v} \in N(g) \wedge \exists \mathsf{c} \in \mathsf{C} : \mathsf{c} \in N(g)$ is equivalent to $\mathsf{v} \notin N(g) \vee \forall \mathsf{c} \in \mathsf{C} : \mathsf{c} \notin N(g) \Rightarrow g \notin \delta(\mathsf{T}_j, \bar{e}_j)$: If $\mathsf{v} \notin N(g)$, either (a) *no* 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. $g \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftarrow \mathsf{v} \in N(g) \wedge \exists \mathsf{c} \in \mathsf{C} : \mathsf{c} \in N(g)$: As n is inserted between v and c, all *pq*-grams that contain both of them are affected.

Delete. $g \in \delta(\mathsf{T}_j, \bar{e}_j) \Rightarrow \mathsf{n} \in N(g)$ is equivalent to $\mathsf{n} \notin N(g) \Rightarrow g \notin \delta(\mathsf{T}_j, \bar{e}_j)$: If n is not in g, no node of g is affected. $g \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftarrow \mathsf{n} \in N(g)$: n does not exist in T_i. If n is in g, g is *only* in $\mathbf{P}_j$.

Rename. $\mathsf{n} \notin N(g) \Rightarrow g \notin \delta(\mathsf{T}_j, \bar{e}_j)$: If n is not in g, no node of g is affected. $g \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftarrow \mathsf{n} \in N(g)$: $\lambda(\mathsf{n}) = l$ in T_i, but $\lambda(\mathsf{n}) \neq l$ in T_j. As $g \in \mathbf{P}_j$, $\lambda(\mathsf{n}) \neq l$ in g. Thus, if n is in g, g is *only* in $\mathbf{P}_j$. $\square$

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 $\mathsf{T}_j = e_j(\mathsf{T}_i)$, then $\delta(\mathsf{T}_j, \bar{e}_j)$ denotes the pq-grams that are added by e_j, and $\delta(\mathsf{T}_i, e_j)$ denotes the pq-grams that are deleted by e_j (Figure 6). Since T_i 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 $\mathsf{T}_i, \mathsf{T}_j$ be trees with profiles $\mathbf{P}_i, \mathbf{P}_j$, respectively, let e_j be an edit operation and $\bar{e}_j$ its reverse operation such that $\mathsf{T}_i = \bar{e}_j(\mathsf{T}_j)$, and let $\delta(\mathsf{T}_j, \bar{e}_j) \subseteq \mathsf{p}_j \subseteq \mathbf{P}_j$. The* profile update function, *$\mathcal{U} : 2^{\mathbf{P}_j} \to 2^{\mathbf{P}_i}$, is defined as follows:*

$$\mathcal{U}(\mathsf{p}_j, \bar{e}_j) = \mathsf{p}_j \setminus \delta(\mathsf{T}_j, \bar{e}_j) \cup \delta(\mathsf{T}_i, e_j) \tag{9}$$

If $\mathsf{p}_j = \delta(\mathsf{T}_j, \bar{e}_j)$, the profile update function computes the old pq-grams from the new pq-grams, i.e., $\delta(\mathsf{T}_i, e_j) = \mathcal{U}(\delta(\mathsf{T}_j, \bar{e}_j), \bar{e}_j)$. If $\mathsf{p}_j = \mathbf{P}_j$, the original profile $\mathbf{P}_i$ is computed from $\mathbf{P}_j$. Due to the symmetry of the scenario also the opposite direction holds:

$$\mathbf{P}_i = \mathcal{U}(\mathbf{P}_j, \bar{e}_j) \qquad \mathbf{P}_j = \mathcal{U}(\mathbf{P}_i, e_j) \tag{10}$$

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. Δ_n^+ denotes the new pq-grams in $\mathbf{P}_n$ that were not present in $\mathbf{P}_0$ and have been introduced by one of the edit operations. Δ_n^- denotes the old pq-grams in $\mathbf{P}_0$ that have been removed by one of the edit operations and, hence, are not present in $\mathbf{P}_n$.

Definition 6. *Let $\mathsf{T}_0, \ldots, \mathsf{T}_n$ be trees with profiles $\mathbf{P}_0, \ldots, \mathbf{P}_n$, respectively, where T_0 has been transformed into T_n by a sequence of edit operations $(e_1, \ldots, e_n)$, i.e., $\mathsf{T}_k = e_k(\mathsf{T}_{k-1})$ for $1 \leq k \leq n$. We define the following sets of pq-grams:*

$$\begin{aligned} &\text{Invariant } pq\text{-grams:} && \mathbf{C}_n = \mathbf{P}_0 \cap \cdots \cap \mathbf{P}_n && (11) \\ &\text{Old } pq\text{-grams:} && \Delta_n^- = \mathbf{P}_0 \setminus \mathbf{C}_n && \\ &\text{New } pq\text{-grams:} && \Delta_n^+ = \mathbf{P}_n \setminus \mathbf{C}_n && (12) \end{aligned}$$

Figure 7 illustrates these sets for a scenario with $n = 2$. The two shaded regions in Figure 7(a) together form the set Δ_2^+, i.e., the new pq-grams in $\mathbf{P}_2$ that were not present in $\mathbf{P}_0$. Note that there might exist new pq-grams that have been added by an edit operation but are not contained in the final profile $\mathbf{P}_2$, since they have been removed by a subsequent edit operation. Hence, Δ_n^+ is in general a subset of all new pq-grams that have been introduced by edit operations. $\mathbf{C}_2$ is the set of pq-grams that are shared by all trees.

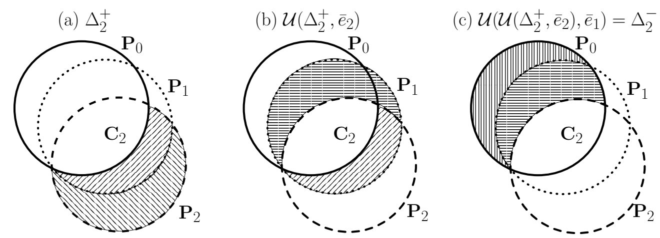

Figure 7: Profiles for Two Edit Operations.

Having determined the set Δ_n^+, we recursively apply the profile update function for each reverse edit operation in the log-file: first for $\bar{e}_n$, then for $\bar{e}_{n-1}$, etc. This process transforms Δ_n^+ into the set Δ_n^- of old pq-grams that have been dropped from $\mathbf{P}_0$ by one of the edit operations. Figure 7(b-c) show this transformation of Δ_2^+ into Δ_2^-. The first call of the update function considers the edit operation $\bar{e}_2$ and substitutes the new pq-grams in Δ_2^+ 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 Δ_2^- of old pq-grams that have been removed from $\mathbf{P}_0$.

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 $\mathcal{I}_o = \lambda(\mathbf{P}_0)$ that is transformed to T_n with index $\mathcal{I}_n = \lambda(\mathbf{P}_n)$ by a sequence of n edit operations. The new index, $\mathcal{I}_n$, can be computed from the old index, $\mathcal{I}_0$, as follows:*

$$\mathcal{I}_n = \mathcal{I}_0 \setminus \lambda(\Delta_n^-) \uplus \lambda(\Delta_n^+). \tag{13}$$

Proof. First we show that replacing the old by the new pq-grams in $\mathbf{P}_0$ results in $\mathbf{P}_n$: $\mathbf{P}_0 \setminus \Delta_n^- \overset{(12)}{=} \mathbf{P}_0 \setminus [\mathbf{P}_0 \setminus \mathbf{C}_n] \overset{(2)}{=} \mathbf{P}_0 \cap \mathbf{C}_n \overset{(11)}{=} \mathbf{C}_n$, thus $\mathbf{P}_0 \setminus \Delta_n^- \cup \Delta_n^+ = \mathbf{C}_n \cup \Delta_n^+ \overset{(12)}{=} \mathbf{C}_n \cup [\mathbf{P}_n \setminus \mathbf{C}_n] \overset{(4)(11)}{=} \mathbf{P}_n$. As $\mathcal{I}_n = \lambda(\mathbf{P}_n)$ it follows that $\mathcal{I}_n = \lambda(\mathbf{P}_0 \setminus \Delta_n^- \cup \Delta_n^+)$. Next we show $\lambda(\mathbf{P}_0 \setminus \Delta_n^- \cup \Delta_n^+) = \lambda(\mathbf{P}_0) \setminus \lambda(\Delta_n^-) \uplus \lambda(\Delta_n^+)$: As $\lambda()$ maps equal pq-grams in different pq-gram sets to equal label-tuples, for each pq-gram $g \in \Delta_n^-$ that is subtracted from $\mathbf{P}_0$ the respective label-tuple $\lambda(g) \in \lambda(\Delta_n^-)$ is subtracted from $\lambda(\mathbf{P}_0)$. As $\Delta_n^- \subseteq \mathbf{P}_0$ (12), also $\lambda(\Delta_n^-) \subseteq \lambda(\mathbf{P}_0)$. Thus for each subtracted label-tuple $\lambda(g) \in \lambda(\Delta_n^-)$ there is a pq-gram, $g \in \Delta_n^-$, that is subtracted from $\mathbf{P}_0$. This shows that $\lambda(\mathbf{P}_0 \setminus \Delta_n^-) = \lambda(\mathbf{P}_0) \setminus \lambda(\Delta_n^-)$. The set union, $\lambda([\mathbf{P}_0 \setminus \Delta_n^-] \cup \Delta_n^+)$ and the bag union, $\lambda(\mathbf{P}_0 \setminus \Delta_n^-) \uplus \lambda(\Delta_n^+)$, are equivalent if $[\mathbf{P}_0 \setminus \Delta_n^-]$ is disjoint from Δ_n^+. Then no pq-grams get lost with the set union. This is the case, as $\mathbf{P}_0 \setminus \Delta_n^- = \mathbf{C}_n$ (see above) and $\Delta_n^+ \overset{(12)}{=} \mathbf{P}_n \setminus \mathbf{C}_n$. $\square$

6.2 Deltas of Intermediate Tree Versions

For the computation of Δ_n^- and Δ_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.

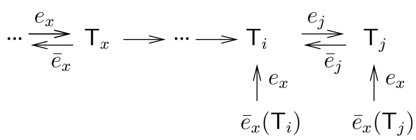

Figure 8: Setting in Lemma 3.

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. $\delta(\mathsf{T}_j, \bar{e}_x)$ computes the new pq-grams for the edit operation e_x that transforms $\bar{e}_x(\mathsf{T}_j)$ into T_j. $\bar{e}_x(\mathsf{T}_j)$ 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 $\bar{e}_x(\mathsf{T}_i)$ and T_j to $\bar{e}_x(\mathsf{T}_j)$,*

$$\delta(\mathsf{T}_i, \bar{e}_x) \setminus \delta(\mathsf{T}_i, e_j) = \delta(\mathsf{T}_j, \bar{e}_x) \setminus \delta(\mathsf{T}_j, \bar{e}_j). \tag{14}$$

Note that $\delta(\mathsf{T}_i, e_j) = \mathcal{U}(\delta(\mathsf{T}_j, \bar{e}_j), \bar{e}_j)$ are the old, $\delta(\mathsf{T}_j, \bar{e}_j)$ the new pq-grams of e_j.

PROOF. (14) is equivalent to

$$g \in \delta(\mathsf{T}_i, \bar{e}_x) \wedge g \notin \delta(\mathsf{T}_i, e_j) \Leftrightarrow \\ g \in \delta(\mathsf{T}_j, \bar{e}_x) \wedge g \notin \delta(\mathsf{T}_j, \bar{e}_j). \tag{15}$$

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

Rename. We first show $L \Rightarrow g \notin \delta(\mathsf{T}_j, \bar{e}_j)$: $g \in \mathbf{P}_i \cap \mathbf{P}_j$ implies $g \notin \delta(\mathsf{T}_j, \bar{e}_j)$, as $\delta(\mathsf{T}_j, \bar{e}_j) = \mathbf{P}_j \setminus \mathbf{P}_i$ (6). Now we show $L \Rightarrow g \in \delta(\mathsf{T}_j, \bar{e}_x)$: L implies that the renamed node n is a node of g ($g \in \delta(\mathsf{T}_i, \bar{e}_x) \Rightarrow \mathsf{n} \in N(g)$ (8)). As g is in $\mathbf{P}_j$ ($L \Rightarrow g \in \mathbf{P}_i \cap \mathbf{P}_j$) and it contains the node renamed by $\bar{e}_x$, it is an new pq-gram of $\mathbf{P}_j$ with respect to e_x: $\mathsf{n} \in N(g) \wedge g \in \mathbf{P}_j \Rightarrow g \in \delta(\mathsf{T}_j, \bar{e}_x)$ (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 $\mathsf{C} = \{\mathsf{c}_k, \ldots, \mathsf{c}_m\}$ move under n. We show $L \Rightarrow g \notin \delta(\mathsf{T}_j, \bar{e}_j)$: $L \Rightarrow g \in \mathbf{P}_i \cap \mathbf{P}_j \Rightarrow g \notin \delta(\mathsf{T}_j, \bar{e}_j)$. We show $L \Rightarrow g \in \delta(\mathsf{T}_j, \bar{e}_x)$: L implies that (a) the parent of the inserted node and at least on of its children are in g ($g \in \delta(\mathsf{T}_i, \bar{e}_x) \Rightarrow \mathsf{v} \in N(g) \wedge \exists \mathsf{c} \in \mathsf{C} : \mathsf{c} \in N(g)$ (7)), and (b) that $g \in \mathbf{P}_j$ ($L \Rightarrow g \in \mathbf{P}_i \cap \mathbf{P}_j$). With (a), (b): $\mathsf{v} \in N(g) \wedge \exists \mathsf{c} \in \mathsf{C} : \mathsf{c} \in N(g) \wedge g \in \mathbf{P}_j \Rightarrow g \in \delta(\mathsf{T}_j, \bar{e}_x)$.

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

6.3 Computing Δ_n^+

In this section we show that the new pq-grams, Δ_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., $\Delta_n^+ = \bigcup_{k=1}^n \delta(\mathsf{T}_n, \bar{e}_k)$. Δ_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. Δ_n^+ is the set of new pq-grams that are present in $\mathbf{P}_n$. 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, $\mathsf{T}_i = e_i(\mathsf{T}_{i-1})$, $1 \leq i \leq n$.*

$$\mathbf{P}_i = \mathbf{P}_0 \setminus \underbrace{\bigcup_{k=1}^{i} \delta(\mathsf{T}_{k-1}, e_k)}_{\mathbf{A}_i} \cup \underbrace{\bigcup_{k=1}^{i} \delta(\mathsf{T}_i, \bar{e}_k)}_{\mathbf{B}_i} \tag{16}$$

PROOF. (i) True for $\mathbf{P}_1$. (ii) With $\mathbf{A}_i = \bigcup_{k=1}^i \delta(\mathsf{T}_{k-1}, e_k)$ and $\mathbf{B}_i = \bigcup_{k=1}^i \delta(\mathsf{T}_i, \bar{e}_k)$ the induction hypothesis is

$$\mathbf{P}_i = \mathbf{P}_0 \setminus \mathbf{A}_i \cup \mathbf{B}_i \Rightarrow \mathbf{P}_{i+1} = \mathbf{P}_0 \setminus \mathbf{A}_{i+1} \cup \mathbf{B}_{i+1}.$$

$$\begin{aligned}
\mathbf{P}_{i+1} &\overset{(10)}{=} \mathcal{U}(\mathbf{P}_i, e_{i+1}) \\
&\overset{(9)}{=} [\mathbf{P}_0 \setminus \mathbf{A}_i \cup \mathbf{B}_i] \setminus \delta(\mathsf{T}_i, e_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \\
&\overset{(3)}{=} \mathbf{P}_0 \setminus [\mathbf{A}_i \cup \delta(\mathsf{T}_i, e_{i+1})] \cup \\
&\qquad [\mathbf{B}_i \setminus \delta(\mathsf{T}_i, e_{i+1})] \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1})
\end{aligned}$$

$$\begin{aligned}
&\mathbf{A}_i \cup \delta(\mathsf{T}_i, e_{i+1}) = \bigcup_{k=1}^{i+1} \delta(\mathsf{T}_{k-1}, e_k) = \mathbf{A}_{i+1} \\
&\mathbf{B}_i \setminus \delta(\mathsf{T}_i, e_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \\
&\quad \overset{(14)}{=} \bigcup_{k=1}^{i} \delta(\mathsf{T}_{i+1}, \bar{e}_k) \setminus \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \\
&\quad \overset{(4)}{=} \bigcup_{k=1}^{i} \delta(\mathsf{T}_{i+1}, \bar{e}_k) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) = \mathbf{B}_{i+1}
\end{aligned} \tag{17}$$

Thus, $\mathbf{P}_{i+1} = \mathbf{P}_0 \setminus \mathbf{A}_{i+1} \cup \mathbf{B}_{i+1}$. $\square$

LEMMA 5. *Let $L = (e_1, \ldots, e_n)$ be a sequence of edit operations that transforms T_0 into T_n, $\mathsf{T}_i = e_i(\mathsf{T}_{i-1})$, $1 \leq i \leq n$. Let $\mathbf{A}_n = \bigcup_{k=1}^n \delta(\mathsf{T}_{k-1}, e_k)$. Then*

$$\mathbf{C}_n = \mathbf{P}_0 \setminus \mathbf{A}_n. \tag{18}$$

PROOF. (a) $\mathbf{P}_0 \setminus \mathbf{A}_n \supseteq \mathbf{C}_n$: $\mathbf{C}_n \overset{(11)}{=} \mathbf{P}_0 \cap \bigcap_{k=1}^n \mathbf{P}_k \overset{(10)}{=} \mathbf{P}_0 \cap \bigcap_{k=1}^n [\mathbf{P}_{k-1} \setminus \delta(\mathsf{T}_{k-1}, e_k) \cup \delta(\mathsf{T}_k, \bar{e}_k)]$. As $\delta(\mathsf{T}_{k-1}, e_k) \cap \delta(\mathsf{T}_k, \bar{e}_k) = \emptyset$, $\mathbf{C}_n = \mathbf{P}_0 \cap \bigcap_{k=1}^n [\mathbf{P}_{k-1} \cup \delta(\mathsf{T}_k, \bar{e}_k) \setminus \delta(\mathsf{T}_{k-1}, e_k)] \Rightarrow \mathbf{C}_n \cap \mathbf{A}_n = \emptyset$.

(b) $\mathbf{P}_0 \setminus \mathbf{A}_n \subseteq \mathbf{C}_n$: The opposite, $g \in \mathbf{P}_0 \setminus \mathbf{A}_n$ and $g \notin \mathbf{C}_n$, leads to a contradiction: $g \notin \mathbf{C}_n \overset{(11)}{\Rightarrow} \exists_{\mathbf{P}_i} g \notin \mathbf{P}_i, 0 \leq i \leq n$. However, by induction we show that $\forall_{\mathbf{P}_i} g \in \mathbf{P}_i$: $g \in \mathbf{P}_0$ is true. $g \in \mathbf{P}_i \Rightarrow g \in \mathbf{P}_{i+1}, 0 \leq i \leq n-1$: $\mathbf{P}_{i+1} \overset{(10)}{=} \mathbf{P}_i \setminus \delta(\mathsf{T}_i, e_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1})$; $g \in \mathbf{P}_0 \setminus \mathbf{A}_n \Rightarrow g \notin \mathbf{A}_n \Rightarrow \forall_{i=0..n-1} g \notin \delta(\mathsf{T}_i, e_{i+1}) \Rightarrow g \in \mathbf{P}_{i+1}$. $\square$

LEMMA 6. *Let $L = (e_1, \dots, e_n)$ be a sequence of edit operations that transforms T_0 into T_n, $\mathsf{T}_i = e_i(\mathsf{T}_{i-1})$, $1 \le i \le n$. Let $\mathbf{B}_i = \bigcup_{k=1}^{i} \delta(\mathsf{T}_i, \bar{e}_k)$. Then*

$$\mathbf{B}_n \cap \mathbf{C}_n = \emptyset. \tag{19}$$

PROOF. Proof by induction. (i) True for $i = 1$: $\mathbf{B}_1 = \delta(\mathsf{T}_1, \bar{e}_1) \Rightarrow \mathbf{B}_1 \cap \mathbf{P}_0 = \emptyset \overset{(11)}{\Rightarrow} \mathbf{B}_1 \cap \mathbf{C}_n = \emptyset$.

(ii) Induction hypothesis:

$$\mathbf{B}_i \cap \mathbf{C}_n = \emptyset \Rightarrow \mathbf{B}_{i+1} \cap \mathbf{C}_n = \emptyset. \tag{20}$$

We show $\mathbf{B}_{i+1} \cap \mathbf{C}_n \subseteq \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{C}_n$: $\mathbf{B}_{i+1} \cap \mathbf{C}_n \overset{(17)}{=} [\mathbf{B}_i \setminus \delta(\mathsf{T}_i, e_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1})] \cap \mathbf{C}_n \subseteq [\mathbf{B}_i \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1})] \cap \mathbf{C}_n = [\mathbf{B}_i \cap \mathbf{C}_n] \cup [\delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{C}_n] \overset{(20)}{=} [\delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{C}_n]$. Then it follows with $\delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{P}_i = \emptyset \overset{(11)}{\Rightarrow} \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{C}_n = \emptyset$ that $\mathbf{B}_{i+1} \cap \mathbf{C}_n = \emptyset$. $\square$

THEOREM 1. *Let $L = (e_1, \dots, e_n)$ be a sequence of edit operations that transforms T_0 into T_n, $\mathsf{T}_i = e_i(\mathsf{T}_{i-1})$, $1 \le i \le n$. The set of new pq-grams, Δ_n^+, can be computed as*

$$\Delta_n^+ = \bigcup_{k=1}^{n} \delta(\mathsf{T}_n, \bar{e}_k). \tag{21}$$

PROOF. With Lemma 4, $\mathbf{P}_n$ can be expressed as

$$\mathbf{P}_n = \mathbf{P}_0 \setminus \mathbf{A}_n \cup \mathbf{B}_n, \tag{22}$$

where $\mathbf{A}_n$ are the old *pq*-grams of each individual edit step, and $\mathbf{B}_n$ are the new *pq*-grams for the edit operations in the log computed on T_n: $\mathbf{A}_n = \bigcup_{k=1}^{n} \delta(\mathsf{T}_{k-1}, e_k)$ and $\mathbf{B}_n = \bigcup_{k=1}^{n} \delta(\mathsf{T}_n, \bar{e}_k)$. We show that $\mathbf{B}_n$ is equivalent to Δ_n^+: $\mathbf{P}_n \overset{(22)}{=} \mathbf{P}_0 \setminus \mathbf{A}_n \cup \mathbf{B}_n \overset{(18)}{=} \mathbf{C}_n \cup \mathbf{B}_n$. As $\mathbf{B}_n$ and $\mathbf{C}_n$ are disjoint (Lemma 6), we can rewrite $\mathbf{P}_n = \mathbf{C}_n \cup \mathbf{B}_n$ as $\mathbf{B}_n = \mathbf{P}_n \setminus \mathbf{C}_n \overset{(12)}{=} \Delta_n^+$. $\square$

6.4 Computing Δ_n^-

If we look at the scenario in the reverse direction (T_n is transformed to T_0 by a sequence of edit operations, $(\bar{e}_n, \dots, \bar{e}_1)$), then Δ_n^+ in the reverse scenario corresponds to Δ_n^- in the original scenario. Thus in the original scenario $\Delta_n^- = \bigcup_{k=1}^{n} \delta(\mathsf{T}_0, e_k)$. As T_0 is not given, we can not use this approach to compute Δ_n^-.

For two trees, $\mathsf{T}_j = e_j(\mathsf{T}_i)$, the profile update function computes $\mathbf{P}_i$ from $\mathbf{P}_j$, $\mathbf{P}_i = \mathcal{U}(\mathbf{P}_j, \bar{e}_j)$ (10). Thus, we can compute $\mathbf{P}_0$ from $\mathbf{P}_n$ by applying the profile update function recursively, $\mathbf{P}_0 = \mathcal{U}(\dots\mathcal{U}(\mathcal{U}(\mathbf{P}_n, \bar{e}_n), \bar{e}_{n-1}) \dots, \bar{e}_1)$. Recall that $\Delta_n^- = \mathbf{P}_0 \setminus \mathbf{C}_n$ is a subset of $\mathbf{P}_0$ and $\Delta_n^+ = \mathbf{P}_n \setminus \mathbf{C}_n$ is a subset of $\mathbf{P}_n$ (12). In this section we show that, similar to $\mathbf{P}_0$ and $\mathbf{P}_n$, we can compute Δ_n^- from Δ_n^+ by applying the update function recursively to Δ_n^+,

$$\Delta_n^- = \mathcal{U}(\dots\mathcal{U}(\mathcal{U}(\Delta_n^+, \bar{e}_n), \bar{e}_{n-1}) \dots, \bar{e}_1).$$

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

LEMMA 7. *Let Δ_i^* be the result of iteratively applying the profile update function to Δ_n^+ i times, $1 \le i \le n$,*

$$\Delta_i^* = \mathcal{U}(\dots\mathcal{U}(\mathcal{U}(\Delta_n^+, \bar{e}_n), \bar{e}_{n-1}) \dots, \bar{e}_{n-i+1}). \tag{23}$$

Then Δ_i^* can be written in un-nested form as

$$\Delta_i^* = \underbrace{\bigcup_{k=1}^{n-i} \delta(\mathsf{T}_{n-i}, \bar{e}_k)}_{\mathbf{A}_i^*} \cup \underbrace{\bigcup_{k=n-i+1}^{n} \delta(\mathsf{T}_{n-i}, e_k)}_{\mathbf{B}_i^*}. \tag{24}$$

PROOF. We define $\mathbf{A}_i^* = \bigcup_{k=1}^{n-i} \delta(\mathsf{T}_{n-i}, \bar{e}_k)$ and $\mathbf{B}_i^* = \bigcup_{k=n-i+1}^{n} \delta(\mathsf{T}_{n-i}, e_k)$, and show (24) by induction:

(i) Δ_1^* computed with (23) and (24) matches: $\Delta_1^* \overset{(24)}{=} \bigcup_{k=1}^{n-1} \delta(\mathsf{T}_{n-1}, \bar{e}_k) \cup \delta(\mathsf{T}_{n-1}, e_n)$. $\Delta_1^* \overset{(23)}{=} \mathcal{U}(\Delta_n^+, \bar{e}_n) \overset{(21)}{=} \mathcal{U}(\bigcup_{k=1}^{n} \delta(\mathsf{T}_n, \bar{e}_k), \bar{e}_n) \overset{(9)}{=} \bigcup_{k=1}^{n} \delta(\mathsf{T}_n, \bar{e}_k) \setminus \delta(\mathsf{T}_n, \bar{e}_n) \cup \delta(\mathsf{T}_{n-1}, e_n) = \bigcup_{k=1}^{n-1} \delta(\mathsf{T}_n, \bar{e}_k) \setminus \delta(\mathsf{T}_n, \bar{e}_n) \cup \delta(\mathsf{T}_{n-1}, e_n) \overset{(3)(14)}{=} \bigcup_{k=1}^{n-1} \delta(\mathsf{T}_{n-1}, \bar{e}_k) \setminus \delta(\mathsf{T}_{n-1}, e_n) \cup \delta(\mathsf{T}_{n-1}, e_n) \overset{(4)}{=} \bigcup_{k=1}^{n-1} \delta(\mathsf{T}_{n-1}, \bar{e}_k) \cup \delta(\mathsf{T}_{n-1}, e_n)$.

(ii) Induction hypothesis:

$$\Delta_i^* = \mathbf{A}_i^* \cup \mathbf{B}_i^* \Rightarrow \Delta_{i+1}^* = \mathbf{A}_{i+1}^* \cup \mathbf{B}_{i+1}^*$$

$$\begin{aligned} \Delta_{i+1}^* &\overset{(23)}{=} \mathcal{U}(\Delta_i^*, \bar{e}_{n-i}) = \mathcal{U}(\mathbf{A}_i^* \cup \mathbf{B}_i^*, \bar{e}_{n-i}) \\ &\overset{(9)}{=} [\mathbf{A}_i^* \cup \mathbf{B}_i^*] \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) \cup \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \\ &\overset{(3)}{=} [\mathbf{A}_i^* \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i})] \cup [\mathbf{B}_i^* \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i})] \cup \\ &\quad\ \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \end{aligned} \tag{25}$$

$$\begin{aligned} &\mathbf{A}_i^* \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) \\ &\quad \overset{(3)}{=} \bigcup_{k=1}^{n-i-1} \delta(\mathsf{T}_{n-i}, \bar{e}_k) \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) \\ &\quad \overset{(14)}{=} \bigcup_{k=1}^{n-i-1} \delta(\mathsf{T}_{n-i-1}, \bar{e}_k) \setminus \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \\ &\quad = \mathbf{A}_{i+1}^* \setminus \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \end{aligned} \tag{26}$$

$$\begin{aligned} &\mathbf{B}_i^* \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) \\ &\quad \overset{(14)}{=} \bigcup_{k=n-i+1}^{n} \delta(\mathsf{T}_{n-i-1}, e_k) \setminus \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \end{aligned} \tag{27}$$

$$\begin{aligned} &\mathbf{B}_i^* \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) \cup \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \\ &\quad \overset{(27)(4)}{=} \bigcup_{k=n-i+1}^{n} \delta(\mathsf{T}_{n-i-1}, e_k) \cup \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \\ &\quad = \bigcup_{k=n-i}^{n} \delta(\mathsf{T}_{n-i-1}, e_k) = \mathbf{B}_{i+1}^* \end{aligned} \tag{28}$$

With (25), (26) and (28) we get $\mathbf{P}_{i+1}^* = \mathbf{A}_{i+1}^* \cup \mathbf{B}_{i+1}^*$. $\square$

THEOREM 2. *Let $L = (e_1, \dots, e_n)$ be a sequence of edit operations that transforms T_0 into T_n, $\mathsf{T}_i = e_i(\mathsf{T}_{i-1})$, $1 \le i \le n$. The set of old pq-grams, Δ_n^-, can be computed as*

$$\Delta_n^- = \mathcal{U}(\dots\mathcal{U}(\mathcal{U}(\Delta_n^+, \bar{e}_n), \bar{e}_{n-1}) \dots, \bar{e}_1). \tag{29}$$

PROOF. As $\Delta_n^- = \mathcal{U}(\dots\mathcal{U}(\mathcal{U}(\Delta_n^+, \bar{e}_n), \bar{e}_{n-1}) \dots, \bar{e}_1) \overset{(23)}{=} \Delta_n^*$, with (24) we can rewrite (29) in un-nested form as

$$\Delta_n^- = \bigcup_{k=1}^{n} \delta(\mathsf{T}_0, e_k). \tag{30}$$

For the proof of (30) consider the inverse scenario, i.e., T_n is transformed to T_0 by $(\bar{e}_n, \dots \bar{e}_1)$. With the substitutions $\mathbf{P}'_i = \mathbf{P}_{n-i}$, $\mathsf{T}'_i = \mathsf{T}_{n-i}$, and $e'_i = \bar{e}_{n-i+1}$, the invariant *pq*-grams of the inverse scenario are $\mathbf{C}'_n = \bigcap_{i=0}^{n} \mathbf{P}'_i$, and the new *pq*-grams can be expressed as

$$\Delta_n'^+ \overset{(12)}{=} \mathbf{P}'_n \setminus \mathbf{C}'_n \quad \text{or} \quad \Delta_n'^+ \overset{(21)}{=} \bigcup_{k=1}^{n} \delta(\mathsf{T}_0, e_k).$$

$\mathbf{C}'_n = \mathbf{C}_n$ as both of them are the intersection of the same profiles. With $\mathbf{P}'_n = \mathbf{P}_0$ we get $\Delta'^{+}_n = \mathbf{P}_0 \setminus \mathbf{C}_n \overset{(12)}{=} \Delta^{-}_n$. $\square$

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 consist of q null nodes.

Definition 7. p-Matrix and q-Matrix. *Let T be a tree, $p > 0$, $q > 0$, and let $\mathsf{a} \in N(\mathsf{T})$ be a node with children $\mathsf{c}_1, \ldots, \mathsf{c}_f$. The* p-matrix, $P(\mathsf{a})$, *of node a is the $1 \times p$-matrix that represents the p-part of all pq-grams anchored in a:*

$$P(\mathsf{a}) = (\mathsf{a}_{p-1}, \ldots, \mathsf{a}_i, \ldots, \mathsf{a}_1, \mathsf{a})$$

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

$$Q(\mathsf{a}) = \begin{pmatrix} \bullet & \cdots & \bullet & \mathsf{c}_1 \\ \vdots & & & \vdots \\ \bullet & & & \mathsf{c}_f \\ \mathsf{c}_1 & & & \bullet \\ \vdots & & & \vdots \\ \mathsf{c}_f & \bullet & \cdots & \bullet \end{pmatrix}$$

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, $P(\mathsf{a}) \circ Q(\mathsf{a})$, 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.*

$$\begin{aligned} P(\mathsf{n}_1) \circ Q(\mathsf{n}_1) &= (\bullet, \bullet, \mathsf{n}_1) \circ \begin{pmatrix} \bullet & \bullet & \mathsf{n}_2 \\ \bullet & \mathsf{n}_2 & \mathsf{n}_3 \\ \mathsf{n}_2 & \mathsf{n}_3 & \mathsf{n}_4 \\ \mathsf{n}_3 & \mathsf{n}_4 & \bullet \\ \mathsf{n}_4 & \bullet & \bullet \end{pmatrix} \\ &= \{(\bullet, \bullet, \mathsf{n}_1, \bullet, \bullet, \mathsf{n}_2), (\bullet, \bullet, \mathsf{n}_1, \bullet, \mathsf{n}_2, \mathsf{n}_3), \\ &\quad (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_2, \mathsf{n}_3, \mathsf{n}_4), (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_4, \bullet), \\ &\quad (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_4, \bullet, \bullet)\} \end{aligned}$$

7.2 Effective Computation of δ and $\mathcal{U}$

For each edit operation we express the new *pq*-grams, $\delta(\mathsf{T}_j, \bar{e})$, in terms of p- and q-matrices, and show, how the old *pq*-grams, $\mathcal{U}(\delta(\mathsf{T}_j, \bar{e}), \bar{e})$, are computed from the new ones.

To facilitate the discussion about the computation of the profile update function, we introduce the following notation: $\mathrm{desc}_d(\mathsf{n})$ is the set of n and its descendants within distance d, i.e., $\mathrm{desc}_d(\mathsf{n}) = \{\mathsf{x} \mid \mathsf{x}$ is n or a descendant of n with $\mathrm{dist}(\mathsf{n}, \mathsf{x}) \leq d\}$. We use $\mathrm{desc}_d(\mathsf{n}_k, \ldots, \mathsf{n}_m)$ as an abbreviation for $\{\mathsf{x} \mid \mathsf{x} \in \mathrm{desc}_d(\mathsf{n}) \wedge \mathsf{n} \in \{\mathsf{n}_k, \ldots, \mathsf{n}_m\}\}$, i.e., all descendants within distance d of a node set.

Given a p-matrix $P(\mathsf{a})$, the operation $P^{+\mathsf{n},i}(\mathsf{a})$ inserts node n at position i, $P^{-\mathsf{a}_i}(\mathsf{a})$ deletes node a_i from $P(\mathsf{a})$, and

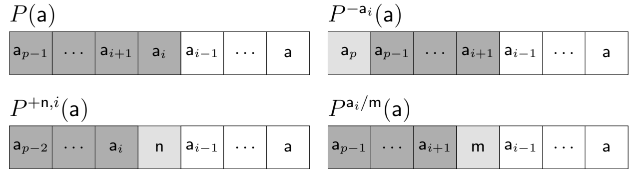

Figure 9: Operators on the p-Matrix.

$P^{\mathsf{a}_i/\mathsf{m}}$ replaces a_i by m. The other nodes in $P(\mathsf{a})$ 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. $Q(\mathsf{a})$ is the q-matrix for anchor node a. The (inverse) diagonals are formed by the children $\mathsf{c}_1, \ldots, \mathsf{c}_f$ of a, and the corners are filled with null nodes. With $Q^{k..m}(\mathsf{a})$ we denote the sub-matrix of $Q(\mathsf{a})$ that is formed by the rows k to $m+q-1$. It contains all q-parts of the children $\mathsf{c}_k, \ldots, \mathsf{c}_m$. We introduce the operator $A /\!\!/ B$ that replaces all diagonals of A with the diagonals of B. $D(\mathsf{n})$ initializes a new q-matrix of size $q \times q$, with the only diagonal formed by node n.

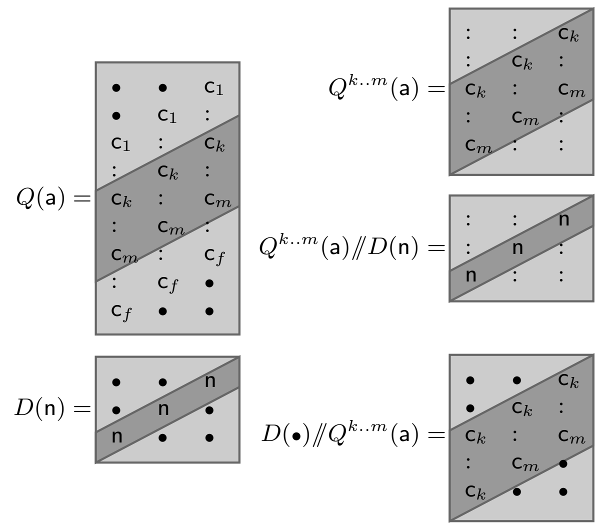

Figure 10: Operators on the q-Matrix.

For insertions and deletions of leaf nodes we define the following special cases: For the q-matrix of a leaf node a we define $Q^{k..m}(\mathsf{a}) = (\bullet \ldots \bullet)$ 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 e_1 in Figure 2), and $Q^{k..m}(\mathsf{v})$ has no diagonals. We define $Q^{k..k-1}(\mathsf{v}) /\!\!/ A$ to insert all diagonals of A as new diagonals in $Q^{k..k-1}(\mathsf{v})$, and we define $A /\!\!/ Q^{k..k-1}(\mathsf{v}) = (\bullet \ldots \bullet)$.

Table 1 shows for each edit operation the *pq*-gram set that forms $\delta(\mathsf{T}_j, \bar{e})$ and how this set is modified by the profile update function. We use the notation introduced above. All information for the computation of the profile update function is in the *pq*-grams of $\delta(\mathsf{T}_j, \bar{e})$ and the edit operation $\bar{e}$. The tree T_j is not accessed.

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 $\bar{e}_1 = \mathrm{DEL}(\mathsf{n}_7)$ and $\bar{e}_2 = \mathrm{INS}((\mathsf{n}_3, \mathsf{b}), \mathsf{n}_1, 2, 3)$. We determine the new pq-grams, Δ_2^+, $p = q = 3$, by evaluating*

Insert node n as the k-th child of node v: $\textsc{ins}(\mathsf{n}, \mathsf{v}, k, m)$

$$\delta(\mathsf{T}_j, \bar{e}) = P(\mathsf{v}) \circ Q^{k..m}(\mathsf{v}) \cup P(\mathsf{x}) \circ Q(\mathsf{x}) \qquad \forall \mathsf{x} \in \mathrm{desc}_{p-2}(\mathsf{c}_k, \ldots, \mathsf{c}_m)$$

$$\mathcal{U}(\delta(\mathsf{T}_j, \bar{e}), \bar{e}) = P(\mathsf{v}) \circ [Q^{k..m}(\mathsf{v}) /\!\!/ D(\mathsf{n})] \cup P^{+\mathsf{n},0}(\mathsf{v}) \circ [D(\bullet) /\!\!/ Q^{k..m}(\mathsf{v})] \cup P^{+\mathsf{n},d}(\mathsf{x}) \circ Q(\mathsf{x}) \qquad \forall \mathsf{x} \in \mathrm{desc}_{p-2}(\mathsf{c}_k, \ldots, \mathsf{c}_m),\ d = \mathrm{dist}(\mathsf{c}_i, \mathsf{x}) + 1$$

c_i : i-th child of v

Delete node n, $\textsc{del}(\mathsf{n})$:

$$\delta(\mathsf{T}_j, \bar{e}) = P(\mathsf{v}) \circ Q^{k..k}(\mathsf{v}) \cup P(\mathsf{x}) \circ Q(\mathsf{x}) \qquad \forall \mathsf{x} \in \mathrm{desc}_{p-1}(\mathsf{n})$$

$$\mathcal{U}(\delta(\mathsf{T}_j, \bar{e}), \bar{e}) = P(\mathsf{v}) \circ [Q^{k..k}(\mathsf{v}) /\!\!/ Q(\mathsf{n})] \cup P^{-\mathsf{n}}(\mathsf{x}) \circ Q(\mathsf{x}) \qquad \forall \mathsf{x} \in \mathrm{desc}_{p-1}(\mathsf{n}) \setminus \{\mathsf{n}\}$$

v : n is the k-th child of v

Rename node n to l': $\textsc{ren}(\mathsf{n}, l')$

$$\delta(\mathsf{T}_j, \bar{e}) = P(\mathsf{v}) \circ Q^{k..k}(\mathsf{v}) \cup P(\mathsf{x}) \circ Q(\mathsf{x}) \qquad \forall \mathsf{x} \in \mathrm{desc}_{p-1}(\mathsf{n})$$

$$\mathcal{U}(\delta(\mathsf{T}_j, \bar{e}), \bar{e}) = P(\mathsf{v}) \circ [Q^{k..k}(\mathsf{v}) /\!\!/ D(\mathsf{m})] \cup P^{\mathsf{n}/\mathsf{m}}(\mathsf{x}) \circ Q(\mathsf{x}) \qquad \forall \mathsf{x} \in \mathrm{desc}_{p-1}(\mathsf{n})$$

$\mathsf{m} = (\mathrm{id}(\mathsf{n}), l')$ v : n is the k-th child of v

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

the delta functions in Table 1 for $\bar{e}_1$ and $\bar{e}_2$, i.e.,

$$\begin{aligned}\Delta_2^+ = \delta(\mathsf{T}_2, \bar{e}_1) \cup \delta(\mathsf{T}_2, \bar{e}_2) = &\{(\bullet, n_1, n_6, \bullet, \bullet, n_7), (\bullet, n_1, n_6, \bullet, n_7, \bullet),\\ &(\bullet, n_1, n_6, n_7, \bullet, \bullet), (n_1, n_6, n_7, \bullet, \bullet, \bullet)\} \cup\\ &\{(\bullet, \bullet, n_1, \bullet, n_2, n_5), (\bullet, \bullet, n_1, n_2, n_5, n_6), (\bullet, \bullet, n_1, n_5, n_6, n_4),\\ &(\bullet, \bullet, n_1, n_6, n_4, \bullet), (\bullet, n_1, n_5, \bullet, \bullet, \bullet), (\bullet, n_1, n_6, \bullet, \bullet, n_7),\\ &(\bullet, n_1, n_6, \bullet, n_7, \bullet), (\bullet, n_1, n_6, n_7, \bullet, \bullet), (n_1, n_6, n_7, \bullet, \bullet, \bullet)\}\\ = &\{(\bullet, \bullet, n_1, \bullet, n_2, n_5), (\bullet, \bullet, n_1, n_2, n_5, n_6), (\bullet, \bullet, n_1, n_5, n_6, n_4),\\ &(\bullet, \bullet, n_1, n_6, n_4, \bullet), (\bullet, n_1, n_5, \bullet, \bullet, \bullet), (\bullet, n_1, n_6, \bullet, \bullet, n_7),\\ &(\bullet, n_1, n_6, \bullet, n_7, \bullet), (\bullet, n_1, n_6, n_7, \bullet, \bullet), (n_1, n_6, n_7, \bullet, \bullet, \bullet)\}.\end{aligned}$$

Next, we compute the old pq-grams Δ_2^- from Δ_2^+, using the profile update function, i.e., $\Delta_2^- = \mathcal{U}(\mathcal{U}(\Delta_2^+, \bar{e}_2), \bar{e}_1)$. Figure 11 shows some of the modified q-matrices that are used in the evaluation of the update function for $\bar{e}_2 = \textsc{ins}((n_3, b), n_1, 2, 3)$. The relevant p-parts in Δ_2^+ are transformed by inserting the new node n_3, e.g.,

$$\begin{aligned}P(n_1) = (\bullet, \bullet, n_1) \quad &\rightarrow \quad P^{+n_3,0}(n_1) = (\bullet, n_1, n_3)\\ P(n_5) = (\bullet, n_1, n_5) \quad &\rightarrow \quad P^{+n_3,1}(n_5) = (n_1, n_3, n_5)\end{aligned}$$

By concatenating the respective p- and q-parts we get

$$\begin{aligned}\mathcal{U}(\Delta_2^+, \bar{e}_2) = &\{(\bullet, \bullet, n_1, \bullet, n_2, n_3), (\bullet, \bullet, n_1, n_2, n_3, n_4), (\bullet, \bullet, n_1, n_3, n_4, \bullet),\\ &(\bullet, n_1, n_3, \bullet, \bullet, n_5), (\bullet, n_1, n_3, \bullet, n_5, n_6), (\bullet, n_1, n_3, n_5, n_6, \bullet),\\ &(\bullet, n_1, n_3, n_6, \bullet, \bullet), (n_1, n_3, n_5, \bullet, \bullet, \bullet), (n_1, n_3, n_6, \bullet, \bullet, n_7),\\ &(n_1, n_3, n_6, \bullet, n_7, \bullet), (n_1, n_3, n_6, n_7, \bullet, \bullet), (n_3, n_6, n_7, \bullet, \bullet, \bullet)\}.\end{aligned}$$

Now the profile update function for $\bar{e}_1$ is applied to the result of $\mathcal{U}(\Delta_2^+, \bar{e}_2)$ which returns the final set of old pq-grams

$$\begin{aligned}\Delta_2^- = &(\bullet, \bullet, n_1, \bullet, n_2, n_3), (\bullet, \bullet, n_1, n_2, n_3, n_4), (\bullet, \bullet, n_1, n_3, n_4, \bullet),\\ &(\bullet, n_1, n_3, \bullet, \bullet, n_5), (\bullet, n_1, n_3, \bullet, n_5, n_6), (\bullet, n_1, n_3, n_5, n_6, \bullet),\\ &(\bullet, n_1, n_3, n_6, \bullet, \bullet), (n_1, n_3, n_5, \bullet, \bullet, \bullet), (n_1, n_3, n_6, \bullet, \bullet, \bullet)\}.\end{aligned}$$

The final step is to update $\mathcal{I}_0$ with $\lambda(\Delta_n^+)$ and $\lambda(\Delta_n^-)$.

$$\begin{aligned}\lambda(\Delta_2^-) = &\{(*, *, a, *, c, b), (*, *, a, c, b, c), (*, *, a, b, c, *),\\ &(*, a, b, *, *, e), (*, a, b, *, e, f), (*, a, b, e, f, *),\\ &(*, a, b, f, *, *), (a, b, e, *, *, *), (a, b, f, *, *, *)\}\\ \lambda(\Delta_2^+) = &\{(*, *, a, *, c, e), (*, *, a, c, e, f), (*, *, a, e, f, c),\\ &(*, *, a, f, c, *), (*, a, e, *, *, *), (*, a, f, *, *, g),\\ &(*, a, f, *, g, *), (*, a, f, g, *, *), (a, f, g, *, *, *)\}\end{aligned}$$

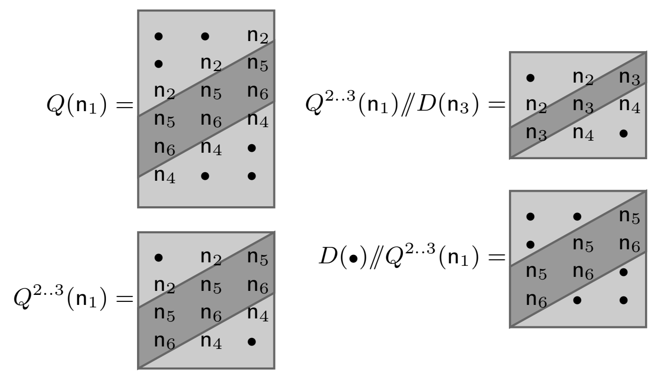

Figure 11: q-Matrices for Node Insertion (Example).

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 Δ_n^+ (see Algorithm 2). We store the p-parts and q-parts of these pq-grams in a pair $(\mathtt{P}, \mathtt{Q})$ 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 $(\mathtt{P}, \mathtt{Q})$ for each edit operation in the log and, step by step, transforms it to Δ_n^-. We prevent duplicates from being inserted into $\mathtt{P}$ and $\mathtt{Q}$, and we join them to reconstruct the pq-grams. An index on the anchor IDs proved to give a substantial performance advantage.

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

$$\lambda(\mathtt{P}, \mathtt{Q}) = \pi_{ppart \circ qpart}[\mathtt{P} \bowtie \mathtt{Q}]. \tag{31}$$

Subsequently, given pairs of tables we use the notation

P

anchId	*sibPos*	*parId*	*ppart*
n_1	-	-	001
n_5	2	n_1	018
n_6	3	n_1	014
n_7	1	n_6	147

Q

anchId	*row*	*qpart*
n_1	2	028
n_1	3	284
n_1	4	842
n_1	5	420
n_5	1	000
n_6	1	007
n_6	2	070
n_6	3	700
n_7	1	000

Figure 12: Δ_2^+ for T_2, Stored in the Table Pair $(\mathtt{P}, \mathtt{Q})$.

$(A, B) \leftarrow (A', B') \cup (A'', B'')$ for $A \leftarrow A' \cup A''$ and $B \leftarrow B' \cup B''$. We use relational algebra expressions in the description of the algorithms. The expression $A = A \setminus B \cup C$ is implemented as an efficient UPDATE statement in SQL.

EXAMPLE 6. *Figure 12 shows $\Delta_2^+ = \bigcup_{i=1}^{2} \delta(\mathsf{T}_2, \bar{e}_i)$ for our example tree in Figure 2. The first rows of* P *and* Q *show the hashed p-part and q-part of the label-tuple* $(*, *, \mathsf{a}, *, \mathsf{c}, \mathsf{e})$.

8.2 Index Update

For the index maintenance we use the old index $\mathcal{I}_0$, the resulting tree T_n, and the log L. The index is updated in three major steps, the computation of Δ_n^+, the computation of Δ_n^- from Δ_n^+, and the update of $\mathcal{I}_0$ with $\lambda(\Delta_n^+)$ and $\lambda(\Delta_n^-)$ (see Algorithm 1). Δ_n^+ is computed by evaluating the delta function for all edit operations in the log on T_n (line 2), Δ_n^- is computed by applying the profile update function recursively to Δ_n^+ (line 4).

Algorithm 1: $\mathtt{updateIndex}(\mathtt{I}_0, \mathsf{T}_n, L)$

1 $(\mathtt{P}, \mathtt{Q}) \leftarrow (\emptyset, \emptyset)$;
2 **foreach** $\bar{e}_i \in L$ **do** $(\mathtt{P}, \mathtt{Q}) \leftarrow (\mathtt{P}, \mathtt{Q}) \cup \delta(\mathsf{T}_n, \bar{e}_i)$;
3 $\mathtt{I}^+ \leftarrow \lambda(\mathtt{P}, \mathtt{Q})$;
4 **for** $i \leftarrow n$ **downto** 1 **do** $\mathcal{U}(\mathtt{P}, \mathtt{Q}, \bar{e}_i)$;
5 $\mathtt{I}^- \leftarrow \lambda(\mathtt{P}, \mathtt{Q})$;
6 $\mathtt{I}_n \leftarrow \mathtt{I}_0 \setminus \mathtt{I}^- \cup \mathtt{I}^+$;
7 **return** $\mathtt{I}_n$;

$\delta(\mathsf{T}, \bar{e}_i)$ computes all *pq*-grams of a subtree of T. The subtree size is independent of the tree size $|T|$, and we consider it a constant. Then the nodes of the subtree are accessed in $O(\log |\mathsf{T}|)$ time, and the delta function returns a constant number of *pq*-grams. $\mathcal{U}(\mathtt{P}, \mathtt{Q}, \bar{e}_i)$ operates on the result of the $|L|$ delta computations, where $|L|$ is the log size. Each *pq*-gram is accessed in $O(\log |L|)$ time and a constant time transformation is applied to it. Both delta and update function are computed $|L|$ times, resulting in an overall complexity of $O(|L|(\log |T| + \log |L|))$. Our experiments confirm the near constant complexity of the delta and the profile update function, and the linear dependence of the overall algorithm from the log size.

8.3 Delta Function

The delta function $\delta(\mathsf{T}, \bar{e})$ is computed by creating the relevant p- and q-matrices from the tree T (see Algorithm 2). The relevant matrices for each edit operation are shown in Table 1. The p-part $P(\mathsf{n})$ is computed by accessing the $p-1$ ancestors of n in the tree. $Q^{k..m}(\mathsf{n})$ is formed by accessing the children $k - q + 1$ to $m + q - 1$ of n, $Q(\mathsf{n})$ by accessing all children of n. We use the functions $\mathtt{P}_\mathsf{T}(\mathsf{n})$, $\mathtt{Q}_\mathsf{T}^{k..m}(\mathsf{n})$ and $\mathtt{Q}_\mathsf{T}(\mathsf{n})$ 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: $\delta(\mathsf{T}, \bar{e})$

if $(\bar{e} = \text{REN}(\mathsf{n}, l')) \vee (\bar{e} = \text{DEL}(\mathsf{n}))$ **then**
    $\mathsf{v} \leftarrow$ parent of n;
    $k \leftarrow$ sibling position of n (n is the k-th child of v);
    $(\mathtt{P}, \mathtt{Q}) \leftarrow (\mathtt{P}_\mathsf{T}(\mathsf{v}), \mathtt{Q}_\mathsf{T}^{k..k}(\mathsf{v}))$;
    foreach $\mathsf{x} \in \text{desc}_{p-1}(\mathsf{n})$ **do**
        $(\mathtt{P}, \mathtt{Q}) \leftarrow (\mathtt{P}, \mathtt{Q}) \cup (\mathtt{P}_\mathsf{T}(\mathsf{x}), \mathtt{Q}_\mathsf{T}(\mathsf{x}))$
    end
else if $\bar{e}$=INS$(\mathsf{n},\mathsf{v},k,m)$ **then**
    $(\mathtt{P}, \mathtt{Q}) \leftarrow (\mathtt{P}_\mathsf{T}(\mathsf{v}), \mathtt{Q}_\mathsf{T}^{k..m}(\mathsf{v}))$;
    foreach *child* $\mathsf{c} \in \{\mathsf{c}_k, \ldots, \mathsf{c}_m\}$ *of* v **do**
        foreach $\mathsf{x} \in \text{desc}_{p-2}(\mathsf{c})$ **do**
            $(\mathtt{P}, \mathtt{Q}) \leftarrow (\mathtt{P}, \mathtt{Q}) \cup (\mathtt{P}_\mathsf{T}(\mathsf{x}), \mathtt{Q}_\mathsf{T}(\mathsf{x}))$
        end
    end
end
return $(\mathtt{P}, \mathtt{Q})$;

8.4 Implementation of the Update Function

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

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

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

In the second step we modify $\delta(\mathsf{T}, \bar{e})$ to get $\mathcal{U}(\delta(\mathsf{T}, \bar{e}), \bar{e})$. We implement the operator $A /\!\!/ B$ so it operates on q-matrices represented as $(anchId, row, qpart)$ 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. $\mathtt{D}_\mathsf{a}(\mathsf{n})$ 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 $\mathtt{changePParts}(\mathtt{P}, \mathsf{n}, s, d)$ (see Algorithm 4). It implements the operators on $P(\mathsf{a})$ ($P^{+\mathsf{n},i}$, $P^{-\mathsf{a}_i}$, $P^{\mathsf{a}_i/\mathsf{m}}$) 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 $P(\mathsf{a})$ (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). $\mathtt{P}_{\mathtt{old}}$ returns all p-parts of P whose anchor node is n or a descendant of n within distance d. $\mathtt{P}_{\mathtt{new}}$ is the same set of tuples with the updated values for *ppart*.

If rows are deleted from/inserted into the q-matrix, the row numbers, *row*, of the subsequent rows need to be updated. If p-parts are deleted or inserted, the sibling numbers, *sibPos*, in the p-parts of the subsequent siblings have to be updated. In both cases the scope of the update query

Algorithm 3: $\mathcal{U}(\mathtt{P}, \mathtt{Q}, \bar{e})$

switch $\bar{e}$ **do**
case REN(n, l')
  $t \leftarrow \sigma_{anchId=\mathsf{n}}(\mathtt{P})$; $\mathsf{v} \leftarrow t[parId]$; $k \leftarrow t[sibPos]$;
  $\mathtt{Q} \leftarrow \mathtt{Q} \setminus \mathtt{Q}^{k..k}(\mathsf{v}) \cup [\mathtt{Q}^{k..k}(\mathsf{v}) /\!\!/ \mathtt{D}_{\mathsf{v}}((\mathrm{id}(\mathsf{n}), l'))]$;
  $s \leftarrow \mathtt{subStr}(t[ppart], 1, p-1) \circ l'$;
  $(\mathtt{P}_{\mathtt{old}}, \mathtt{P}_{\mathtt{new}}) \leftarrow \mathtt{changePParts}(\mathtt{P}, \mathsf{n}, s, p-1)$;
  $\mathtt{P} \leftarrow \mathtt{P} \setminus \mathtt{P}_{\mathtt{old}} \cup \mathtt{P}_{\mathtt{new}}$;
case DEL(n)
  $t \leftarrow \sigma_{anchId=\mathsf{n}}(\mathtt{P})$; $\mathsf{v} \leftarrow t[parId]$; $k \leftarrow t[sibPos]$;
  $\mathtt{Q} \leftarrow \mathtt{Q} \setminus [\mathtt{Q}^{k..k}(\mathsf{v}) \cup \mathtt{Q}(\mathsf{n})] \setminus [\mathtt{Q}^{k..k}(\mathsf{v}) /\!\!/ \mathtt{Q}(\mathsf{n})]$;
  $s \leftarrow \lambda(\bullet) \circ \mathtt{subStr}(t[ppart], 1, p-1)$;
  $(\mathtt{P}_{\mathtt{old}}, \mathtt{P}_{\mathtt{new}}) \leftarrow \mathtt{changePParts}(\mathtt{P}, \mathsf{n}, s, p-1)$;
  $\mathtt{P} \leftarrow \mathtt{P} \setminus \mathtt{P}_{\mathtt{old}} \cup \sigma_{anchId \neq \mathsf{n}}(\mathtt{P}_{\mathtt{new}})$;
case INS$(\mathsf{n}, \mathsf{v}, k, m)$
  $\mathtt{Q} \leftarrow \mathtt{Q} \setminus \mathtt{Q}^{k..m}(\mathsf{v}) \cup [\mathtt{Q}^{k..m}(\mathsf{v}) /\!\!/ \mathtt{D}_{\mathsf{v}}(\mathsf{n})]$
    $\cup [\mathtt{D}_{\mathsf{n}}(\bullet) /\!\!/ \mathtt{Q}^{k..m}(\mathsf{v})]$;
  $s \leftarrow \mathtt{subStr}(\pi_{ppart}\sigma_{anchId=\mathsf{v}}(\mathtt{P}), 2, p) \circ \lambda(\mathsf{n})$;
  $\mathtt{P}_{\mathtt{old}} \leftarrow \emptyset$; $\mathtt{P}_{\mathtt{new}} \leftarrow \emptyset$;
  foreach $\mathsf{c} \in \pi_{anchId}\sigma_{parId=\mathsf{v}, k \leq sibPos \leq m}(\mathtt{P})$ **do**
    $s' \leftarrow \mathtt{subStr}(s, 2, p) \circ \lambda(\mathsf{c})$;
    $(\mathtt{P}_{\mathtt{old}}, \mathtt{P}_{\mathtt{new}}) \leftarrow (\mathtt{P}_{\mathtt{old}}, \mathtt{P}_{\mathtt{new}}) \cup$
      $\mathtt{changePParts}(\mathtt{P}, \mathsf{c}, s', p-2)$;
  end
  $\mathtt{P} \leftarrow \mathtt{P} \setminus \mathtt{P}_{\mathtt{old}} \cup \mathtt{P}_{\mathtt{new}} \cup \{(\mathsf{n}, k, \mathsf{v}, s)\}$;
end

Algorithm 4: $\mathtt{changePParts}(\mathtt{P}, \mathsf{n}, s, d)$

$\mathtt{P}_{\mathtt{old}} \leftarrow \emptyset$; $\mathtt{P}_{\mathtt{new}} \leftarrow \emptyset$;
$\mathtt{Z} \leftarrow \sigma_{anchId=\mathsf{n}}(\mathtt{P})$;
for $i \leftarrow 0$ **to** d **do**
  $\mathtt{P}_{\mathtt{old}} \leftarrow \mathtt{P}_{\mathtt{old}} \cup \mathtt{Z}$;
  $\mathtt{P}_{\mathtt{new}} \leftarrow \mathtt{P}_{\mathtt{new}} \cup \pi[anchId, sibPos, parId,$
    $\mathtt{subStr}(s, i+1, |s|) \circ$
    $\mathtt{subStr}(ppart, p-i+1, p) \rightarrow ppart](\mathtt{Z})$;
  if $i < d$ **then** $\mathtt{Z} \leftarrow \mathtt{P} \bowtie \pi_{anchId \rightarrow parId}\mathtt{Z}$;
end
return $(\mathtt{P}_{\mathtt{old}}, \mathtt{P}_{\mathtt{new}})$;

is limited by the fanout of the anchor node. As typically not all rows of a q-part and not all p-parts of a node's children are in $(\mathtt{P}, \mathtt{Q})$, the effect on structure change is even smaller.

9. EXPERIMENTS

We use XML trees for our experiments. The synthetic trees are generated with **`xmlgen`**, provided by the XML benchmark project XMark[1]. The real world experiments are done on the DBLP dataset[2]. Unless otherwise noted, we use 3, 3-grams for the indexes.

9.1 Lookup Efficiency

If we look up a tree T in a forest F, we have to compute the pq-gram distance between T and each of the trees in F. We compare approximate lookups with and without the use of a *precomputed* index.

We do a lookup in three different collections of XML documents. They have a similar overall number of nodes (approx. 50×10^6). 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.

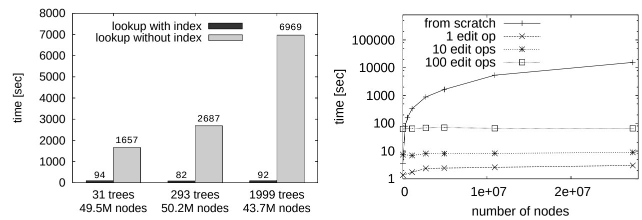

Figure 13: Lookup and Update Time.

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 27×10^6 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.

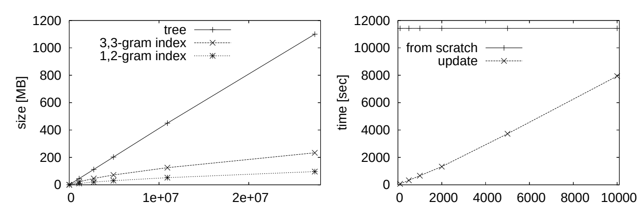

Figure 14: Size and Update Time of Index.

[1] `http://monetdb.cwi.nl/xml/`
[2] `http://www.informatik.uni-trier.de/~ley/db/`

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 Δ_n^+ and Δ_n^- are approximately linear. The update of $\mathtt{I}_0$ with $\lambda(\Delta_n^-)$ and $\lambda(\Delta_n^+)$ is sublinear in the number of edit operations.

Action	Number of edit operations			
	1	10	100	1000
Δ_n^+	0.642s	3.903s	37.533s	391.513s
$\mathtt{I}^+ = \lambda(\Delta_n^+)$	0.184s	0.199s	0.287s	0.443s
Δ_n^-	0.196s	2.836s	27.967s	295.104s
$\mathtt{I}^- = \lambda(\Delta_n^-)$	0.177s	0.191s	0.185s	0.383s
$\mathtt{I}_0 \setminus \mathtt{I}^- \cup \mathtt{I}^+$	2.206s	2.770s	6.475s	19.780s
total	3.405s	9.900s	72.448s	707.224s

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 Loperfido, and Danila Sartori.

11. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of ICDE*, pages 141–152. IEEE Computer Society, 2002.

[2] N. Augsten, M. Böhlen, and J. Gamper. Approximate matching of hierarchical data using *pq*-grams. In *Proc. of VLDB*, pages 301–312. Morgan Kaufmann Publishers Inc., 2005.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. of SIGMOD*, pages 310–321. ACM Press, 2002.

[4] G. Cobéna, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Proc. of ICDE*, pages 41–52. IEEE Computer Society, 2002.

[5] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proc. of VLDB*, pages 341–350. Morgan Kaufmann Publishers Inc., 2001.

[6] M. Garofalakis and A. Kumar. XML stream processing using tree-edit distance embeddings. *ACM Trans. on Database Systems*, 30(1):279–332, 2005.

[7] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In *Proc. of SIGMOD*, pages 287–298. ACM Press, 2002.

[8] S. Guha, N. Koudas, D. Srivastava, and T. Yu. Index-based approximate XML joins. In *Proc. of ICDE*, pages 708–710. IEEE Computer Society, 2003.

[9] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-tree: Indexing XML data for efficient structural joins. In *Proc. of ICDE*, pages 253–263. IEEE Computer Society, 2003.

[10] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[11] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for structure indexes. In *Proc. of VLDB*, pages 239–250. Morgan Kaufmann Publishers Inc., 2002.

[12] K.-H. Lee, Y.-C. Choy, and S.-B. Cho. An efficient algorithm to compute differences between structured documents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(8):965–979, 2004.

[13] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of VLDB*, pages 361–370. Morgan Kaufmann Publishers Inc., 2001.

[14] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate XML query answers. In *Proc. of SIGMOD*, pages 263–274. ACM Press, 2004.

[15] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proc. of SIGMOD*, pages 134–144. ACM Press, 2003.

[16] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In *ICDE*, pages 360–371. IEEE Computer Society, 2005.

[17] E. Ukkonen. Approximate string-matching with *q*-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[18] M. Weis and F. Naumann. DogmatiX tracks down duplicates in XML. In *Proc. of SIGMOD*, pages 431–442. ACM Press, 2005.

[19] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *Proc. of SIGMOD*, pages 754–765. ACM Press, 2005.

[20] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.