Approximate Matching of Hierarchial Data

Augsten, Nikolaus

Publication date:
2008

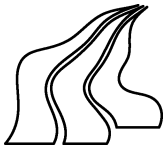
Document Version
Publisher's PDF, also known as Version of record

Nikolaus Augsten

Approximate Matching of Hierarchical Data

Ph.D. Dissertation

Department of Computer Science
Faculty of Engineering and Science
Aalborg University



Approximate Matching of Hierarchical Data

Nikolaus Augsten

Ph.D. Dissertation

A dissertation submitted to the Faculty of Engineering and Science at Aalborg University, Denmark, in partial fulfillment of the requirements for the Ph.D. degree in computer science.

Approximate Matching of Hierarchical Data

Nikolaus Augsten

Faculty of Computer Science
Free University of Bozen-Bolzano

Dominikanerplatz 3, 39100 Bozen, Italy
Tel: +39-0471-016111 – Fax: +39-0471-016009
E-mail: augsten@inf.unibz.it

Abstract

The goal of this thesis is to design, develop, and evaluate new methods for the approximate matching of hierarchical data represented as labeled trees. In approximate matching scenarios two items should be matched if they are similar. Computing the similarity between labeled trees is hard as in addition to the data values also the structure must be considered. A well-known measure for comparing trees is the tree edit distance. It is computationally expensive and leads to a prohibitively high run time.

Our solution for the approximate matching of hierarchical data are *pq-grams.* The *pq*-grams of a tree are all its subtrees of a particular shape. Intuitively, two trees are similar if they have many *pq*-grams in common. The *pq*-gram distance is an efficient and effective approximation of the tree edit distance. We analyze the properties of the *pq*-gram distance and compare it with the tree edit distance and alternative approximations. The *pq*-grams are stored in the *pq*-gram index which is implemented as a relation and represents a *pq*-gram by a fixed-length string. The *pq*-gram index offers efficient approximate lookups and reduces the approximate *pq*-gram join to an equality join on strings. We formally proof that the *pq*-gram index can be incrementally updated based on the log of edit operations without reconstructing intermediate tree versions. The incremental update is independent of the data size and scales to a large number of changes in the data. We introduce windowed *pq*-grams for the approximate matching of unordered trees. Windowed *pq*-grams are generated from sorted trees in linear time. Sorting trees is not possible for common ordered tree distances such as the edit distance.

We present the address connector, a new system for synchronizing residential addresses that implements a *pq*-gram based distance between streets, introduces a global greedy matching that guarantees stable pairs, and links addresses that are stored with different granularity. The connector has been successfully tested with public administration databases. Our extensive experiments on both synthetic and real world data confirm the analytic results and suggest that *pq*-grams are both useful and scalable.

Acknowledgments

I would like to acknowledge the contribution of a number of people that directly or indirectly have supported this PhD thesis.

First of all I would like to thank my supervisor Michael Böhlen. He never got tired of reviewing and discussing my work, and his valuable advice was indispensable for this thesis. His uncompromising commitment to high-quality research made working with him an instructive and enjoyable experience.

I also would like to thank Johann Gamper who contributed to this thesis through many fruitful discussions and iterations over the papers. He is the principal investigator of the *eBZ – Digital City* project, a collaboration between the Municipality of Bolzano and the Free University of Bolzano. My work was mostly funded by this project. Thanks also to my colleagues from the DIS group at the Free University of Bolzano who helped me whenever needed.

In 2006 I spent six month with Curtis Dyreson at the Washington State University in Pullman, USA. I would like to thank Curtis for his valuable contribution to my research, his helpfulness in many practical matters, and the good time that we spent together in the deserts of Utah.

Thanks to Walter Costanzi, Franco Barducci, and Roberto Loperfido from the Municipality of Bolzano. They patiently introduced me to the problems that arise in public administration and they spent many hours in evaluating and discussing the solutions that I proposed.

Finally, I would like to thank my parents Gregor and Anna, my siblings Christine, Michael, Maria, Leopold, Josef, and Monika, and all my friends that supported me throughout the years. In particular I would like to thank my wife Leni. She always believed in me and encouraged me regardless of all the time that we could not spend together due to my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Approximate Matching of Hierarchical Data

The goal of this thesis is to design, develop, and evaluate new methods for the approximate matching of hierarchical data represented as labeled trees. In approximate matching scenarios two items should be matched if they are similar. In order to approximately match two sets of hierarchical data items two orthogonal problems must be solved: computing the similarity between hierarchical data items and establishing a pairwise matching of items.

Hierarchical data are represented as labeled trees. Computing the similarity between labeled trees is hard as in addition to the data values also the structure must be considered. A common reference is the tree edit distance, defined as the minimum number of node edit operations (node insertion, node deletion, node renaming) that transforms one tree to the other. Unfortunately the runtime of the tree edit distance is prohibitive and a scalable alternative is required.

A distance for labeled trees must balance the weight of the structure and the labels. A distance that considers only the tree structure and ignores the labels is an unlabeled tree distance. If the distance considers only the labels and ignores the structure it degrades to a set distance. The labels are not always atomic values and introduce a dimension on their own. In some cases it may be desirable to evaluate the similarity also between individual labels, for example, to tolerate misspellings of string labels. In addition to tree structure and labels, a distance function must deal with the sibling order in the tree. If the sibling order should matter (e.g., the order of paragraphs in a document), the data are represented as ordered, labeled trees. The best algorithms for the ordered tree edit distance have cubic runtime [16]. If the sibling order must be ignored (e.g., data-centric XML), the data are modeled as unordered, labeled

trees. A distance algorithm for unordered trees can not take advantage of a predefined order, and the edit distance problem becomes NP-complete [57].

Based on the distances between the data items we must establish a pairwise matching of items. An approach that matches all pairs of items that are within a fixed distance threshold often produces too few or too many matches. It does not guarantee that each item is matched at most once, which is relevant for many applications. A baseline approach to get unique matches for items is to match each item to its nearest neighbor. Unfortunately the nearest neighbor function is not symmetric (i.e, if a is the nearest neighbor of b, b is not necessarily the nearest neighbor of a), and the matching result depends on the order in which the items are matched. Global optimizations must be applied to produce order-independent matches.

Below we first introduce the application area of our solution to approximate tree matching. Then we describe our contributions. In the last subsection we discuss the organization of the thesis.

1.2 Application Area

In this section we present two applications that require the approximate matching of hierarchical data.

Matching Residential Addresses For many administrative tasks at the municipality of Bolzano-Bozen data from different databases must be accessed and linked. Often the databases are managed by independent departments and residential addresses are the only link between relevant information in different databases. Figure 1.1 shows two residential address databases that cover the same geographic area and should be linked on the address attributes.

Exact matches clearly fail. Also approximate string matching fails since street names have been renamed but were not updated in all databases (`'Friedensplatz'` was renamed to `'Siegesplatz'`) and different languages are used for street names in different databases (`'Friedhofplatz'` and `'Cimitero'` are the German and the Italian name for the same street).

Residential addresses are organized in hierarchies and a street can be represented as an ordered, labeled tree: the *address tree* (see Figure 1.2). Two streets are similar if their address trees are similar, and approximate matching techniques for hierarchical data must be used to match streets.

Integrating XML Data about Music CDs An online database about music CDs should integrate data from two XML sources: a song lyric store and a CD warehouse. The integrated database will store the title, artist and

Electricity Company (EC)

address	bill
Hermann-von-Gilm-Str. 1	€ 121
Hermann-von-Gilm-Str. 3/A	€ 71
Hermann-von-Gilm-Str. 3/B	€ 63
Hermann-von-Gilm-Str. 6	€ 0
Siegesplatz 2/A	€ 98
Siegesplatz 3/-/1	€ 32
Siegesplatz 3/-/2	€ 51
Siegesplatz 3/-/3	€ 43
Friedhofplatz 4	€ 143
Friedhofplatz 6	€ 0
Untervigli 1	€ 117
Mariengasse 1	€ 161

Registration Office (RO)

address	resident
Gilmstrasse 1	Peter
Gilmstrasse 3	Hans
Gilmstrasse 3	Renate
Gilmstrasse 3	Max
Gilmstrasse 5	Arturas
Friedensplatz 2/A/1	Markus
Friedensplatz 2/A/2	Klaudia
Friedensplatz 3	Igor
Cimitero 4	Linas
Cimitero 6/A	Francesco
Cimitero 6/B	Romans
Untervigil 1	Andrej
Marieng. 1/A	Josef

Figure 1.1: Two Databases with Residential Addresses that Cover the Same Geographic Area.

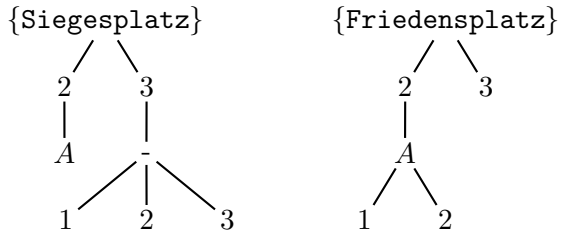
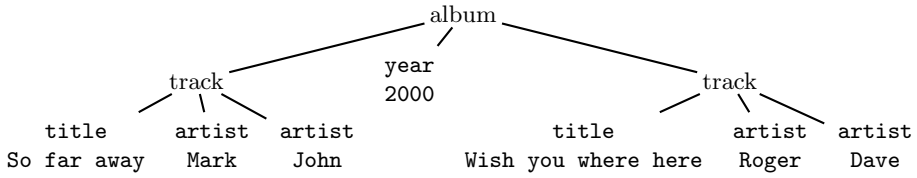


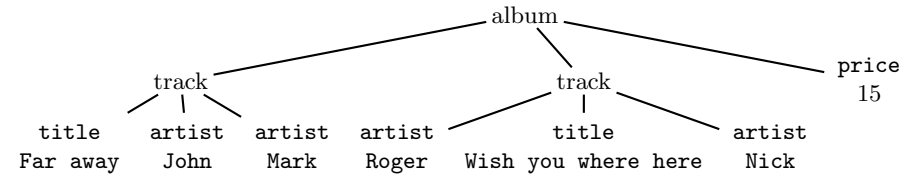
Figure 1.2: Address Trees of `'Siegesplatz'` and `'Friedensplatz'`.

songs of an album, information about individual songs such as the lyrics, guitar tabs, and information about the artists.

Figure 1.3 shows the tree representation of two different XML documents. Intuitively, both represent data about the same song album and they should be matched. Yet exact ordered tree matching would not consider the items as the same for a number of reasons. The song lyric store has an element `year` that is absent from the CD Warehouse. The CD Warehouse has a price for the album. For one track the title is slightly different, for the other track the databases list different artists. Also the document order of elements differs, i.e., the two documents have different sibling orders. As the order of the siblings should not matter, approximate matching techniques for unordered trees must be applied.



(a) Song Lyric Store Data



(b) CD Warehouse Data

Figure 1.3: Two XML Trees Representing the Same Album.

1.3 Contribution

Our solution for the approximate matching of hierarchical data are *pq-grams*. *pq*-Grams are small, besom-shaped subtrees that overlap and cover a tree. Intuitively, two trees are similar if they have many *pq*-grams in common. The *pq*-gram distance is an efficient and effective approximation of the tree edit distance. A major focus of our work is the efficient implementation of *pq*-grams in a relational database. The *pq*-grams are stored in the *pq*-gram index which supports efficient approximate lookups and joins of hierarchical data.

The incremental update of the *pq*-gram index is independent of the data size and scales to a large number of changes in the data. We also introduce windowed *pq*-grams for the approximate matching of *unordered* trees and develop an efficient approximate join algorithm for windowed *pq*-grams. Unordered tree matching is indispensable for data-centric XML. We present the address connector, a new system for synchronizing residential addresses based on *pq*-grams. The system implements the address tree distance, introduces a novel street matching algorithm, and links addresses that are stored with different granularity. The connector solves the problem of linking residential addresses between different databases without requiring a reference set of correct addresses.

Management of *pq*-Grams We present a linear time and space algorithm to compute the *pq*-grams. Our solution can easily be integrated into a relational database or implemented on top of it. The input are hierarchical data stored in a relation. The algorithm splits the trees into *pq*-grams, serializes the *pq*-grams, and hashes them to string values of a fixed length. The result is the *pq-gram index*, a relation that stores a single string for each *pq*-gram. A tree is represented by a set of *pq*-grams in the index. We give an efficient approximate join algorithm based on the *pq*-gram index that is implemented as a query in a relational database. Most joins based on distance measures, such as the edit distance, must evaluate the distance between every pair of input trees. There is no effective way to sort trees or hash them into buckets. An expensive nested-loop join must be applied. Our algorithm reduces the approximate join to an equality join on strings that takes advantage of well known join optimization techniques, for example, the sort-merge join.

Incremental Update of the *pq*-Gram Index The *pq*-gram index needs to be updated in response to structure and value changes in the indexed data. Recomputing the index from scratch is infeasible for large data. As an application scenario consider Figure 1.4. Let $\mathbf{I}_0$ be the *pq*-gram index of a tree T_0. T_0 is modified by a sequence of edit operations resulting in T_n. We provide an incremental update of the *pq*-gram index based on: the old index $\mathbf{I}_0$, the resulting document T_n, and the log of inverse edit operations that describe how T_n can be transformed to T_0. We do not require the original document to still be available, and 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 incremental update does not depend on the tree size, but only on the size of the log. We formally proof that our incremental update is correct.

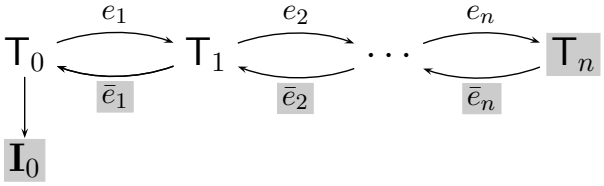


Figure 1.4: Application Scenario for the Incremental Index Update.

Windowed *pq*-Grams for Data-Centric XML XML are data of particular interest for approximate matching efforts as an increasing amount of data is stored and exchanged in XML. While order is important in document-centric scenarios (e.g., paragraph tags in XHTML), most applications of data-centric XML must ignore the sibling order. Two documents should be considered the same even if they differ in the order of their siblings. Data-centric XML items are usually modeled as *unordered*, labeled trees. The tree edit distance between unordered trees is NP-complete. We introduce windowed *pq*-grams for the integration of data-centric XML. We develop a technique to systematically generate windowed *pq*-grams from sorted trees. Sorting trees is not possible for common ordered tree distances such as the edit distance. Windowed *pq*-grams consist of a stem and a base, and they satisfy the following core properties: all base-nodes have equal frequency; the Jaccard distance between two sibling sets is preserved; and node moves to other parents are detected. The windowed *pq*-gram distance between two sorted trees approximates the unordered tree edit distance between these trees. The windowed *pq*-grams are produced in linear time and space, and the approximate join based on windowed *pq*-grams scales to large data sets.

The Address Connector We present the address connector, a new system for the synchronization of residential addresses. The connector solves the problem of matching residential addresses that are stored with different granularity and have different or unrelated street names. The connector implements a novel *address tree distance* that relies on both the structural similarity of the address trees and the similarity of the street names. Our address tree distance allows us to find similar streets also if either their names are very different (e.g., in the case of renamed streets) or if the structure of the address

trees is ambiguous (e.g., the address trees of the streets `'Mariengasse'` and `'Untervigil'` in Figure 1.1 have identical structure). The connector implements a *global greedy matching* algorithm that forms street pairs based on the distance between them. The algorithm exploits the fact that a single street can not be matched to more than one other street. No threshold parameter is required, the matching quality is independent of the matching order, and the matching is stable. We introduce the concept of address containment and link residential addresses of matching streets correctly even if they are stored with different granularity. Our system has been successfully tested in the context of the Municipality of Bolzano.

1.4 Organization of the Thesis

The thesis is organized as a collection of papers. Each of the Chapters 2–5 is based upon one paper. The chapters are self-contained. The experimental evaluation of the analytic results can be found in the respective chapters. The bibliography for all chapters is listed at the end of the thesis. All the chapters adhere to the same terminology. The exception are *stem* and *base* which are referred to as respectively *p-part* and *q-part* in Chapter 3.

Chapter 2 *pq*-Grams for Ordered Trees.

> Nikolaus Augsten, Michael Böhlen, and Johann Gamper. Approximate matching of hierarchical data using *pq*-grams. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 301–312, Trondheim, Norway, September 2005. ACM.

Chapter 3 Updating the *pq*-Gram Index.

> Nikolaus Augsten, Michael Böhlen, and Johann Gamper. An incrementally maintainable index for approximate lookups in hierarchical data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 247–258, Seoul, Korea, September 2006. ACM.

Chapter 4 *pq*-Grams for Unordered Trees.

> Nikolaus Augsten, Michael Böhlen, Curtis Dyreson, and Johann Gamper. Approximate joins for data centric XML. In *Proceedings of the International Conference on Data Engineering (ICDE)*, to appear, 2008.

Chapter 5 The Address Connector.

> Nikolaus Augsten, Michael Böhlen, and Johann Gamper. The Address Connector. Submitted.

Chapter 2

pq-Grams for Ordered Trees

When integrating data from autonomous sources, exact matches of data items that represent the same real world object often fail due to a lack of common keys. Yet in many cases structural information is available and can be used to match such data. As a running example we use residential address information. Addresses are hierarchical structures and are present in many databases. Often they are the best, if not only, relationship between autonomous data sources. Typically the matching has to be approximate since the representations in the sources differ.

We propose pq-grams to approximately match hierarchical information from autonomous sources. We define the pq-gram distance between ordered labeled trees as an effective and efficient approximation of the well-known tree edit distance. We analyze the properties of the pq-gram distance and compare it with the edit distance and alternative approximations. Experiments with synthetic and real world data confirm the analytic results and the scalability of our approach.

2.1 Introduction

When integrating data from autonomous sources, exact matches of data items representing the same real world object often fail due to missing global keys and different data representations. Approximate matching techniques must be applied instead. We focus on hierarchical data, where, in addition to data values, the data structure must also be considered.

As a running example we use an application from our local municipality. The GIS Office wants to relate data about apartments stored in different databases and display this information on a map. This requires a join on the address attributes. An equality join gives extremely poor results, mainly due to the different street names in various databases. Street names vary because different conventions are used to represent them. They may even be stored

in different languages, which prevents the use of standard string comparison techniques. To overcome this problem we exploit the hierarchical organization of addresses. Instead of comparing street names we look for similarities in the hierarchical structure imposed by the addresses of a street.

Hierarchical data can be represented as ordered labeled trees. Data is then matched based on similarities of the corresponding trees. A well-known measure for comparing trees is the tree edit distance. It is computationally very expensive and leads to a prohibitively high run time. We propose the *pq*-gram distance as an effective and efficient approximation of the tree edit distance. The *pq*-grams of a tree are all its subtrees of a particular shape. Intuitively, two trees are close to each other if they have many *pq*-grams in common. For a pair of trees the *pq*-gram distance can be computed in $O(n \log n)$ time and $O(n)$ space, where n is the number of tree nodes.

In general, the *pq*-gram distance is a good approximation of the tree edit distance. In contrast to the tree edit distance, it places more emphasis on modifications to the structure of the tree. For example, deletions of nodes with a rich structure (many descendants) are more expensive than deletions of nodes with a poor structure (e.g., leaf nodes). We show that this property yields intuitive results.

At a technical level, our contribution is a new approximation for the tree edit distance with *pq*-grams. We present an algorithm to compute the *pq*-gram distance in $O(n \log n)$ time and $O(n)$ space, and we show its scalability to large trees stored in a relational database. A core feature of the *pq*-gram distance is its sensitivity to structural changes. This sets it apart from other approximations. Our analytical results are confirmed by experiments on both synthetic and real data.

In the following section we describe the application scenario at our local municipality and give a problem definition. In Section 2.3 we discuss related work. We define the *pq*-gram distance in Section 2.4. In Section 2.5 we give an algorithm for the computation of the *pq*-grams, analyze the complexity of this algorithm, and discuss its implementation in a relational database. We analyze properties of the *pq*-gram distance in Section 2.6. In Section 2.7 we evaluate the efficiency and effectiveness of our method on synthetic and real world data and compare it to other approximations. We draw conclusions in Section 2.8.

2.2 Problem Definition

As a running example we use an application and data from the Municipality of Bozen. The GIS office in the municipality maintains maps of the city

area. It would like to enrich the maps with information retrieved from various databases of the municipality as well as external institutions. Residential addresses turn out to play a pivotal role in this process since they have to be used to access and link relevant information.

Whenever we join on address attributes, we have to know which streets correspond to each other in the joined tables. As an example consider the streets in the databases of the Registration Office (SRO) and the Land Register (SLR) shown in Figure 2.1. The exact join on the street names

$$\texttt{SRO} \bowtie [\texttt{SRO}.\mathit{street} = \texttt{SLR}.\mathit{street}]\ \texttt{SLR}$$

yields poor results since street names are different in different databases due to spelling mistakes, different naming conventions, and renamed streets which are not always updated in all databases. Moreover, in the bilingual region of Bozen two names for each street exist, and they are used interchangeably. A join on the street identifiers is not possible, as they are different in each system. In practice there is no central registry for residential addresses which maintains common keys for street names or addresses.

SRO

id	*street*
30	Giuseppe-Cesare-Abba-Str.
120	Sebastian-Altmann-Str.
5220	Bozner-Boden-Str.
3000	Hermann-von-Gilm-Str.
3030	Pater-Reginaldo-Giuliani-Str.
3540	Italienallee
4440	Musterplatzl
7180	Raffaello-Sernesi-Galerie
7590	Telsergalerie
7620	Friedensplatz
7650	Turiner Str.
7740	Trienter Str.
7860	Triester Str.
8580	Walther-v.-d.-Vogelweide-Pl.
3930	Giannantonio-Manci-Str.
...	

SLR

id	*street*
91	CESARE ABBA STRASSE
74	S. ALTMANN STRASSE
33	BOZNER BODENWEG
109	GILMWEG
185	P. R. GIULIANI STR.
115	ITALIENSTRASSE
165	MUSTERPLATZ
207	SERNESIDURCHGANG
259	TELSERDURCHGANG
139	SIEGESPLATZ
266	TURINER STRASSE
262	TRIENTER STRASSE
263	TRIESTER STRASSE
285	WALTHERPLATZ
86	MANCISTRASSE
...	

Figure 2.1: Street Names in Different Departments.

In order to improve the results we exploit the information about the streets that is stored in the address tables RO and LR (see Figure 2.2) that reference the streets in SRO and SLR, respectively. The addresses from a street are then organized into hierarchies and can be represented in a so-called address tree [2]. Figure 2.3 shows the address trees for the framed addresses in Figure 2.2. The root of the tree is the street name, the children of the street name are the house numbers, the children of house numbers are the entrance numbers, and the

children of entrance numbers are the apartment numbers. A complete address is the path from the root to any leaf node. For example, the tuple $(30, 2, \mathtt{A}, \mathtt{-})$ of table RO represents the address 'Giuseppe-Cesare-Abba-Str. 2A' and corresponds to the shaded path in Figure 2.3. We omit unnecessary empty values ("-") in the address trees.

RO

id	*num*	*entr*	*apt*	*resident*
30	1	-	1	Pichler
30	1	-	3	Rieder
30	2	A	-	Maier
30	2	B	1	Rossi
30	2	B	2	Woelk
30	2	B	3	Verdi
30	2	B	4	Verdi
30	2	C	-	Burger
30	3	-	-	Hofer
30	4	A	1	Tribus
30	4	A	2	Palermo
30	4	A	3	Palermo
30	4	B	-	Abel
30	4	C	-	Rossi
30	6	-	-	Spiro
120	3	A	1	Spiro
120	3	A	2	Barducci
120	3	A	3	Costanzi
120	3	A	4	Pichler
120	3	A	5	Spiro
120	3	A	6	Raifer
...				

LR

id	*num*	*entr*	*apt*	*owner*
91	1	-	1	Maier
91	1	-	2	Rossi
91	1	-	3	Sparber
91	2	A	-	Maier
91	2	B	1	Totti
91	2	B	2	Bracco
91	2	B	3	Mair
91	2	B	4	Lun
91	2	D	-	Tribus
91	3	-	-	Costanzi
91	4	A	-	Palermo
91	4	B	-	Abel
91	4	C	-	Rossi
91	6	-	-	Spiro
74	3	A	1	Spiro
74	3	A	2	Barducci
74	3	A	3	Costanzi
74	3	A	4	Spiro
74	3	A	6	Spiro
74	3	A	7	Hofer
74	4	-	-	Mueller
...				

Figure 2.2: Addresses Stored in Different Departments.

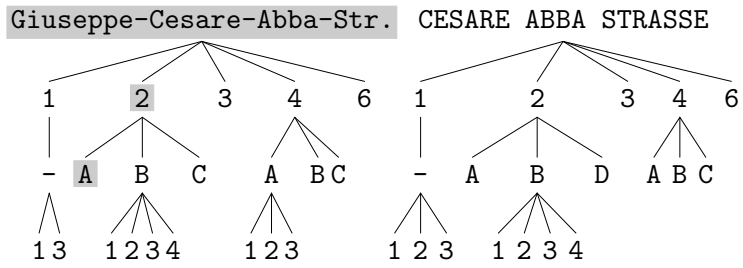


Figure 2.3: Address Trees of Streets 30 from RO and 91 from LR.

With address trees in place, we are able to compare entire address trees so as to match street names of different databases. Intuitively, two streets are identical if they have (almost) the same address tree. We use this to formulate the original join as an *approximate tree join*

$$\mathtt{SRO} \bowtie [\mathrm{dist}(\mathsf{T}(\mathtt{SRO}.id), \mathsf{T}(\mathtt{SLR}.id)) \leq \tau]\ \mathtt{SLR}.$$

Here $\mathsf{T}(id)$ are the address trees of the streets, $\text{dist}(\mathsf{T}_1, \mathsf{T}_2)$ is the distance between trees T_1 and T_2, and τ is a distance threshold. The equality match between street names has been replaced by an approximate matching of the corresponding address trees.

Our goal is to find an effective approximation for the tree edit distance that can be efficiently computed and is scalable to large trees.

2.3 Related Work

A well known distance function for trees is the tree edit distance, which is defined as the minimum cost sequence of edit operations (node insertion, node deletion, and label change) that transforms one tree into another [47]. Zhang and Shasha [56] present an algorithm to compute the tree edit distance in $O(n^2 \min^2(l, d))$ time and $O(n^2)$ space for trees with n nodes, l leaves, and depth d. Other algorithms were presented in more recent works [11, 35]. All of them have more than $O(n^2)$ runtime complexity and do not scale to large trees.

By imposing restrictions on the edit operations that can be applied to transform a tree, suboptimal solutions with better runtime complexities can be found: Alignment distance [32], isolated subtree distance [48], and top-down distance [46, 54] have runtime at least $O(n^2)$, bottom-up distance can be computed in $O(n)$ time. Bottom-up distance tries to find the largest possible common subtrees of two trees, starting with the leaf nodes. It is very sensitive to differences between the leaf nodes. If the leaves are different, the inner nodes are never compared. This makes the bottom-up distance applicable in only very specific domains.

Guha et al. [26] present a framework for approximate XML joins based on tree edit distance, where XML documents are represented as ordered labeled trees. They give upper and lower bounds for the tree edit distance that can be computed in $O(n^2)$ time and use reference sets to take advantage of the fact that the tree edit distance is a metric, thus reducing the actual number of distances to compute in a join. The success of this method depends heavily on a good choice of the reference set. We do not try to limit the number of distance calculations with the expensive tree edit distance, rather we substitute it with an efficient approximation.

Chawathe et al. [10] use a variant of the tree edit distance for change detection. Lee et al. [37] tune the algorithm presented by Chawathe et al. to XML documents. Both algorithms first compute a match between the nodes of the trees, and based on this the distance is computed in $O(ne)$ time, where e is the edit distance between the trees. Whereas in a change detection scenario

typically trees with small differences are compared, for joins the distances between all pairs of trees have to be computed. For trees that are very different the edit distance e is $O(n)$, which yields $O(n^2)$ runtime for both algorithms.

A core operation in XML query processing is to find all occurrences of a twig pattern [5, 31]. The goal of our work is not to find occurrences of a pattern to answer queries. We split the tree into subtrees in order to calculate the distance between trees. Polyzotis et al. [42] build synopsis of an XML tree optimized for approximate query answering. They introduce the Element Simulation Distance to capture the difference between the original tree and the synopsis with respect to twig queries. This distance is tailored to measure the quality of a synopsis and is not suitable as an approximation for the tree edit distance.

Garofalakis and Kumar [23] investigate an algorithm for embedding the tree edit distance (with subtree move as an additional edit operation) into a numeric vector space equipped with the standard L_1 distance norm. The algorithm computes an approximation of the tree edit distance with subtree move (to within a $O(log^2 n \times log^* n)$ factor) in $O(n \times log^* n)$ time and $O(n)$ space[1]. We implement this approximation and empirically compare it to the *pq*-gram distance. The tree embedding distance gives less weight to structural changes than the tree edit distance. The sensitivity of the *pq*-gram distance to structural changes is controlled by the parameters p and q. The *pq*-gram distance typically weights them more than the edit distance.

Navarro [40] gives a good overview of the edit distance for *strings* and its variants. Ukkonen [49] introduces the q-gram distance as a lower bound for the string edit distance. The q-gram distance between two strings is based on the number of common substrings of length q. Gravano et al. [25] present algorithms for approximate string joins based on edit distance and use q-grams as a filtering algorithm. Approximate string matching techniques are successful if the distance between corresponding strings is smaller than that of other strings in the join set. This is typically the case for spelling mistakes, where only a few characters change. The distance between corresponding street names, however, is often larger than the length of the shorter string. If streets are renamed, string matching fails completely.

2.4 The *pq*-Gram Distance

Hierarchical data can be represented as rooted, ordered, labeled trees, where the single data values are represented as labels of the tree nodes. In this

[1] $\log^* n$ denotes the number of log applications required to reduce n to a quantity that is ≤ 1, cf. [23].

section we first give a definition of trees and then define the *pq*-gram distance of trees.

2.4.1 Preliminaries

A *tree* T is a directed, acyclic, connected, non-empty graph with *nodes* $N(\mathsf{T})$ and edges $E(\mathsf{T})$. An *edge* is an ordered pair (p,c), where $\mathsf{p},\mathsf{c} \in N(\mathsf{T})$ are nodes, and p is the *parent* of c. A node can have at most one parent, and nodes with the same parent are *siblings*. An order $\leq$ is defined on the nodes, and this order is total among siblings. The siblings $\mathsf{s}_1 \leq \mathsf{s}_2$ ($\mathsf{s}_1 \neq \mathsf{s}_2$) are *contiguous* if s_1 and s_2 have no sibling x ($\mathsf{s}_1 \neq \mathsf{x} \neq \mathsf{s}_2$) with $\mathsf{s}_1 \leq \mathsf{x} \leq \mathsf{s}_2$. Node c is the i-th *child* of p with $i = |\{\mathsf{x} \in N(\mathsf{T}) | (\mathsf{p},\mathsf{x}) \in E(\mathsf{T}), \mathsf{x} \leq \mathsf{c}\}|$. The number of p's children is its *fanout* f_{p}. The node with no parent is the *root* node $\mathsf{r} = \text{root}(\mathsf{T})$, and a node without children is a *leaf*.

Each node a in the path from the root node to a node v is called an *ancestor* of v. If there is a path of length $k > 0$ from a to v, then a is the ancestor of v at distance k. The parent of a node is its ancestor at distance 1. d is a *descendant* of v if v is an ancestor of d. The *level* of a node level(v) is the length of the path from the root to v, the *depth* of a tree depth(T) is the length of the longest path from the root to any one of the leaves. A *label* is a symbol $\sigma \in \mathbf{\Sigma}$, where $\mathbf{\Sigma}$ is a finite alphabet. Each node $\mathsf{v} \in N(\mathsf{T})$ has assigned a label $\lambda(\mathsf{v})$. A node $\bullet$ with the special label $\lambda(\bullet) = *$ is a *dummy node*.

In our graphical representation of trees we represent nodes as an (identifier, label)-pair, the edges are lines between the nodes, and siblings are ordered from left to right. Whenever possible we omit the identifiers of the nodes to avoid clutter (e.g., in Figure 2.3).

Example 2.1. *Figure 2.4 shows a tree with* $N(\mathsf{T}_1) = \{\mathsf{v}_1,\mathsf{v}_2,\mathsf{v}_3,\mathsf{v}_4,\mathsf{v}_5,\mathsf{v}_6\}$, $E(\mathsf{T}_1) = \{(\mathsf{v}_1,\mathsf{v}_2), (\mathsf{v}_1,\mathsf{v}_5), (\mathsf{v}_1,\mathsf{v}_6), (\mathsf{v}_2,\mathsf{v}_3), (\mathsf{v}_2,\mathsf{v}_4)\}$, *and the order* $\mathsf{v}_2 \leq \mathsf{v}_5 \leq \mathsf{v}_6$, $\mathsf{v}_3 \leq \mathsf{v}_4$. v_1 *has 3 children, where* v_2 *is the first,* v_5 *the second, and* v_6 *the third child. The root node* $\text{root}(\mathsf{T}) = \mathsf{v}_1$. v_1 *is the ancestor of all other nodes.* v_3, v_4, v_5 *and* v_6 *are leaf nodes. The node labels of our example tree are* $\lambda(\mathsf{v}_1) = \mathsf{a}$, $\lambda(\mathsf{v}_2) = \mathsf{a}$, $\lambda(\mathsf{v}_3) = \mathsf{e}$, $\lambda(\mathsf{v}_4) = \mathsf{b}$, $\lambda(\mathsf{v}_5) = \mathsf{b}$, *and* $\lambda(\mathsf{v}_6) = \mathsf{c}$.

A *subtree* S of T is a tree with $N(\mathsf{S}) \subseteq N(\mathsf{T})$ and $E(\mathsf{S}) \subseteq E(\mathsf{T})$, retaining the node order. A *preorder traversal* of a tree visits the root node first, and then recursively traverses all the subtrees rooted in its children in preorder, preserving the children's order. We call a node v the i-th node of T in preorder if v is visited as the i-th node in a preorder traversal.

Two trees T and T' are *isomorphic* if there is a bijective mapping m between the nodes $N(\mathsf{T})$ and $N(\mathsf{T}')$ such that the following holds true: (v,w) is an edge

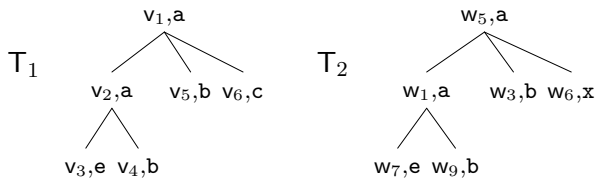


Figure 2.4: Two Example Trees T_1 and T_2.

of T and w is the i-th child of v if and only if $(m(\mathsf{v}), m(\mathsf{w}))$ is an edge of T' and $m(\mathsf{w})$ is the i-th child of $m(\mathsf{v})$.

Example 2.2. *Consider Figure 2.4. The tree* $\mathsf{S}_1 = (\{\mathsf{v}_2, \mathsf{v}_3, \mathsf{v}_4\}, \{(\mathsf{v}_2, \mathsf{v}_3), (\mathsf{v}_2, \mathsf{v}_4)\})$, $\mathsf{v}_3 \leq \mathsf{v}_4$ *is a subtree of* T_1. *The preorder traversal of* T_1 *visits the nodes in the following order:* $\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \mathsf{v}_4, \mathsf{v}_5, \mathsf{v}_6$. *Tree* T_2 *is isomorphic to* T_1 *with* $m = \{(\mathsf{v}_1, \mathsf{w}_5), (\mathsf{v}_2, \mathsf{w}_1), (\mathsf{v}_3, \mathsf{w}_7), (\mathsf{v}_4, \mathsf{w}_9), (\mathsf{v}_5, \mathsf{w}_3), (\mathsf{v}_6, \mathsf{w}_6)\}$.

2.4.2 The *pq*-Gram Distance

In the following paragraphs we define the notion of *pq*-grams and a distance measure based on *pq*-grams. Intuitively, the *pq*-grams of a tree are all subtrees of a specific shape. To ensure that each node of the tree appears in at least one of the *pq*-grams, we extend the tree with *dummy nodes*. The *pq*-grams are then defined as subtrees of the extended tree.

Definition 2.1 (*pq*-Extended Tree)**.** *Let* T *be a tree, and* $p > 0$ *and* $q > 0$ *be two integers. The pq-*extended tree, $\mathsf{T}^{p,q}$, *is constructed from* T *by adding* $p-1$ *ancestors to the root node, inserting* $q-1$ *children before the first and after the last child of each non-leaf node, and adding* q *children to each leaf of* T. *All newly inserted nodes are dummy nodes that do not occur in* T.

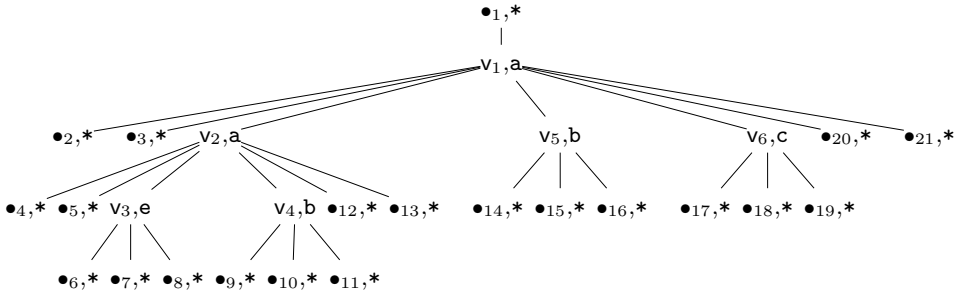


Figure 2.5: The Extended Tree $\mathsf{T}_1^{2,3}$.

Example 2.3. *Figure 2.5 shows the graphical representation of* $\mathsf{T}_1^{2,3}$*, the* $2,3$*-extended tree of our example tree* T_1*.*

Definition 2.2 (*pq*-Gram Pattern)**.** *For* $p > 0$ *and* $q > 0$*, the* *pq*-gram pattern *is a tree that consists of an anchor node with* $p-1$ *ancestors and* q *children.*

Example 2.4. *An example of a* $2,3$*-gram pattern is the tree* $(\{\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{p}_4, \mathsf{p}_5\}, \{(\mathsf{p}_1, \mathsf{p}_2), (\mathsf{p}_2, \mathsf{p}_3), (\mathsf{p}_2, \mathsf{p}_4), (\mathsf{p}_2, \mathsf{p}_5)\}), \mathsf{p}_3 \leq \mathsf{p}_4 \leq \mathsf{p}_5$*.* p_2 *is the anchor node, and it has* 1 *ancestor* (p_1) *and* 3 *children* $(\mathsf{p}_3, \mathsf{p}_4,$ *and* $\mathsf{p}_5)$*.*

Definition 2.3 (*pq*-Gram)**.** *For* $p > 0$ *and* $q > 0$*, a* *pq*-gram G *of a tree* T *is defined as a subtree of the extended tree* $\mathsf{T}^{p,q}$ *with the following properties:* G *is isomorphic to the pq-gram pattern, and contiguous siblings in* G *are contiguous siblings in* $\mathsf{T}^{p,q}$*.*

Definition 2.4 (Label-tuple)**.** *Let* G *be a pq-gram with the nodes* $N(\mathsf{G}) = \{\mathsf{v}_1, \ldots, \mathsf{v}_p, \mathsf{v}_{p+1}, \ldots, \mathsf{v}_{p+q}\}$*, where* v_i *is the* i*-th node in preorder. The tuple* $\lambda(\mathsf{G}) = (\lambda(\mathsf{v}_1), \ldots, \lambda(\mathsf{v}_p), \lambda(\mathsf{v}_{p+1}), \ldots, \lambda(\mathsf{v}_{p+q}))$ *is called the* label-tuple *of* G*.*

Subsequently, if the distinction is clear from the context, we use the term *pq-gram* for both, the *pq*-gram itself and its representation as a label-tuple.

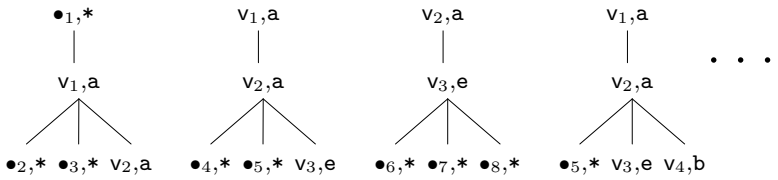


Figure 2.6: Some of the $2,3$-Grams of T_1.

Example 2.5. *Figure 2.6 shows some of the* $2,3$*-grams of the example tree* T_1*. They are constructed by moving the* $2,3$*-gram pattern over the extended tree* $\mathsf{T}_1^{2,3}$ *(see Figure 2.5). We start at the top of the tree. For the first pq-gram the anchor node of the pattern is mapped to* v_1*, and the children of the anchor are mapped to two dummy nodes and* v_2*. The corresponding label-tuple is* $(*, \mathsf{a}, *, *, \mathsf{a})$*.*

Definition 2.5 (*pq*-Gram Index)**.** *For* $p > 0$ *and* $q > 0$*, the* *pq*-gram index*,* $\mathbf{I}^{p,q}(\mathsf{T})$*, of a tree* T *is defined as the bag of label-tuples* $\lambda(\mathsf{G}_i)$ *of all pq-grams* G_i *of* T*.*

$\mathbf{I}^{2,3}(\mathsf{T}_1)$
labels
$(*, \mathtt{a}, *, *, \mathtt{a})$
$(\mathtt{a}, \mathtt{a}, *, *, \mathtt{e})$
$(\mathtt{a}, \mathtt{e}, *, *, *)$
$(\mathtt{a}, \mathtt{a}, *, \mathtt{e}, \mathtt{b})$
$(\mathtt{a}, \mathtt{b}, *, *, *)$
$(\mathtt{a}, \mathtt{a}, \mathtt{e}, \mathtt{b}, *)$
$(\mathtt{a}, \mathtt{a}, \mathtt{b}, *, *)$
$(*, \mathtt{a}, *, \mathtt{a}, \mathtt{b})$
$(\mathtt{a}, \mathtt{b}, *, *, *)$
$(*, \mathtt{a}, \mathtt{a}, \mathtt{b}, \mathtt{c})$
$(\mathtt{a}, \mathtt{c}, *, *, *)$
$(*, \mathtt{a}, \mathtt{b}, \mathtt{c}, *)$
$(*, \mathtt{a}, \mathtt{c}, *, *)$

$\mathbf{I}^{2,3}(\mathsf{T}_2)$
labels
$(*, \mathtt{a}, *, *, \mathtt{a})$
$(\mathtt{a}, \mathtt{a}, *, *, \mathtt{e})$
$(\mathtt{a}, \mathtt{e}, *, *, *)$
$(\mathtt{a}, \mathtt{a}, *, \mathtt{e}, \mathtt{b})$
$(\mathtt{a}, \mathtt{b}, *, *, *)$
$(\mathtt{a}, \mathtt{a}, \mathtt{e}, \mathtt{b}, *)$
$(\mathtt{a}, \mathtt{a}, \mathtt{b}, *, *)$
$(*, \mathtt{a}, *, \mathtt{a}, \mathtt{b})$
$(\mathtt{a}, \mathtt{b}, *, *, *)$
$(*, \mathtt{a}, \mathtt{a}, \mathtt{b}, \mathtt{x})$
$(\mathtt{a}, \mathtt{x}, *, *, *)$
$(*, \mathtt{a}, \mathtt{b}, \mathtt{x}, *)$
$(*, \mathtt{a}, \mathtt{x}, *, *)$

Figure 2.7: $2,3$-Gram Indexes of T_1 and T_2.

The tables in Figure 2.7 show the $2,3$-gram index of T_1 and T_2, respectively. Note that *pq*-grams might appear more than once in a *pq*-gram index, e.g., $(\mathtt{a}, \mathtt{b}, *, *, *)$ appears twice in the index of T_1.

We subsequently define the *pq*-gram distance 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.

Definition 2.6 (*pq*-Gram Distance)**.** *For $p > 0$ and $q > 0$, the* pq*-*gram distance*, $\Delta^{p,q}(\mathsf{T}_1, \mathsf{T}_2)$, *between two trees T_1 and T_2 is defined as follows:*

$$\Delta^{p,q}(\mathsf{T}_1, \mathsf{T}_2) = 1 - 2\frac{|\mathbf{I}^{p,q}(\mathsf{T}_1) \cap \mathbf{I}^{p,q}(\mathsf{T}_2)|}{|\mathbf{I}^{p,q}(\mathsf{T}_1) \cup \mathbf{I}^{p,q}(\mathsf{T}_2)|} \tag{2.1}$$

Example 2.6. *Consider the $2,3$-gram distance between T_1 and T_2. The corresponding $2,3$-gram indexes are shown in Figure 2.7. The bag-intersection of the two indexes is $\{(*, \mathtt{a}, *, *, \mathtt{a}),\ (\mathtt{a}, \mathtt{a}, *, *, \mathtt{e}),\ (\mathtt{a}, \mathtt{e}, *, *, *),\ (\mathtt{a}, \mathtt{a}, *, \mathtt{e}, \mathtt{b}),\ (\mathtt{a}, \mathtt{b}, *, *, *),\ (\mathtt{a}, \mathtt{a}, \mathtt{e}, \mathtt{b}, *),\ (\mathtt{a}, \mathtt{a}, \mathtt{b}, *, *),\ (*, \mathtt{a}, *, \mathtt{a}, \mathtt{b}),\ (\mathtt{a}, \mathtt{b}, *, *, *)\}$, which yields $|\mathbf{I}^{2,3}(\mathsf{T}_1) \cap \mathbf{I}^{2,3}(\mathsf{T}_2)| = 9$. For the cardinality of the bag-union we get $|\mathbf{I}^{2,3}(\mathsf{T}_1) \cup \mathbf{I}^{2,3}(\mathsf{T}_2)| = |\mathbf{I}^{2,3}(\mathsf{T}_1)| + |\mathbf{I}^{2,3}(\mathsf{T}_2)| = 26$. The pq-gram distance is*

$$\Delta^{2,3}(\mathsf{T}_1, \mathsf{T}_2) = 1 - 2\frac{9}{26} = 0.31.$$

The *pq*-gram distance is 1 if two trees share no *pq*-grams. Trees at distance 0 have the same *pq*-gram index. Note that distance 0 does not imply equality

of trees. An example of two different trees with the same pq-gram index is shown in Figure 2.8. The pq-grams responsible for detecting the swapped children of the root nodes of T' and T'' are those anchored in the root nodes. However, as all children of the root nodes have the same label, the pq-grams remain unchanged.

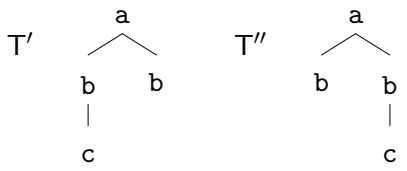


Figure 2.8: Different Trees with the Same pq-Gram Index.

The pq-gram distance can be computed in $O(n \log n)$ time by computing the bag intersection of the pq-gram indexes of size $O(n)$. Theorem 2.1 shows, how the size of the index is related to the number of leaf and non-leaf nodes.

Theorem 2.1. *Let $p > 0$, $q > 0$, and T be a tree with l leaf nodes and i non-leaf nodes. The size of the pq-gram index is*

$$|\mathbf{I}^{p,q}(\mathsf{T})| = 2l + qi - 1.$$

Proof. By structural induction:
$|N(\mathsf{T})| = 1$: The tree consists of the root node only, and according to Definition 2.3 the pq-gram index contains exactly one pq-gram. The number of leaves is 1, the number of non-leaf nodes is 0, thus $|\mathbf{I}^{p,q}(\mathsf{T})| = 2l + qi - 1 = 1$.
$|N(\mathsf{T})| > 1$: In this case $i \geq 1$ (at least the root node) and $l \geq 1$. First we delete all non-leaf nodes (except the root r) and get T'. $|\mathbf{I}^{p,q}(\mathsf{T})| - |\mathbf{I}^{p,q}(\mathsf{T}')| = (i-1) * q$. (Deleting a non-leaf node decreases the cardinality of the pq-gram index by q). The number of leaves does not change with this operation, and the tree now consists of only the leaves and the root node. Now we delete all leaf nodes and get T'', $|\mathbf{I}^{p,q}(\mathsf{T}')| - |\mathbf{I}^{p,q}(\mathsf{T}'')| = 2(l-1) + q$. (Deleting a leaf node decreases the cardinality of the pq-gram index by q if the leaf has no siblings, otherwise by 2). T'' consists only of the root node and $|\mathbf{I}^{p,q}(\mathsf{T}'')| = 1$. This means, $|\mathbf{I}^{p,q}(\mathsf{T})| = 1 + [2(l-1) + q] + [(i-1) * q] = 2l + qi - 1$. □

2.5 Algorithms

2.5.1 An Algorithm for the pq-Gram-Index

The basic idea of the pq-`Gram-Index` algorithm (see Algorithms 2.1 and 2.2) is to move the pq-gram pattern vertically and horizontally over the tree as

illustrated in Figure 2.9. After each move the nodes covered by the pattern form a *pq*-gram.

Algorithm 2.1: *pq*-`Gram-Index`(T, p, q)

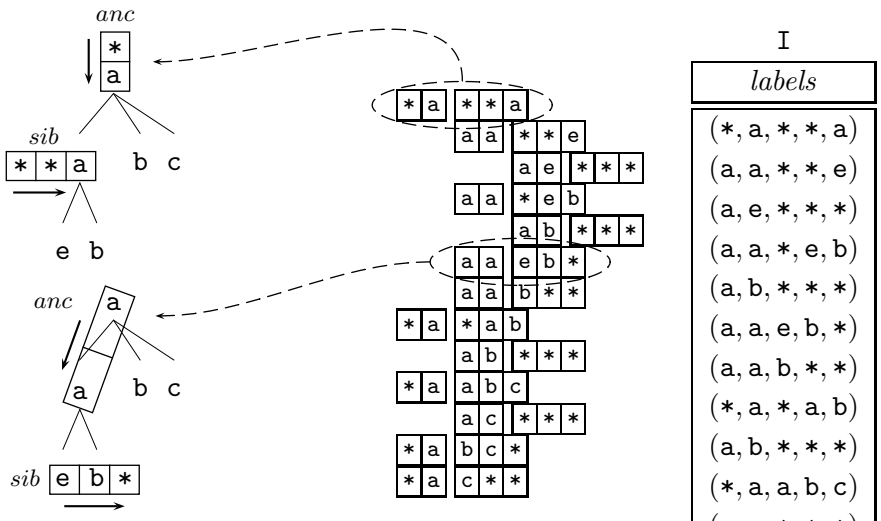
```
1  I : empty relation with schema (labels);
2  anc: shift register of size p (filled with *);
3  I ← index(T, p, q, I, root(T), anc);
4  return I;
```

Algorithm 2.2: `index`$(\mathsf{T}, p, q, \mathtt{I}, \mathtt{r}, anc)$

```
5  sib: shift register of size q (filled with *);
6  anc ← shift(anc, λ(r));
7  if r is a leaf then
8      I ← I ∪ (anc ∘ sib);
9  else
10     foreach child c (from left to right) of r do
11         sib ← shift(sib, λ(c));
12         I ← I ∪ (anc ∘ sib);
13         I ←index(T, p, q, I, c, anc);
14     for k ← 1 to q − 1 do
15         sib ← shift(sib, *);
16         I ← I ∪ (anc ∘ sib);
17 return I;
```

We use two shift registers, *anc* of size p and *sib* of size q, to represent the labels of the ancestor and the leaf nodes that are covered by the *pq*-gram pattern, respectively. A shift register *reg* supports a single operation shift(reg, el), which returns *reg* with the oldest element dequeued and *el* enqueued. For example, shift$((\mathtt{a}, \mathtt{b}, \mathtt{c}), \mathtt{x})$ returns $(\mathtt{b}, \mathtt{c}, \mathtt{x})$. The concatenation of the two registers, $anc \circ sib$, is a tuple in the *pq*-gram index, i.e., for $anc = (l_1, \ldots, l_p)$ and $sib = (l_{p+1}, \ldots, l_{p+q})$ the label-tuple of the *pq*-gram is $(l_1, \ldots, l_p, l_{p+1}, \ldots, l_{p+q})$.

pq-`Gram-Index` takes as input a tree T and the two values p and q and returns a relation that contains the *pq*-gram index of T. After the initialization, `index` calculates the *pq*-grams starting from the root node of T. First `index` shifts the label of anchor node `r` into the register *anc*, which corresponds to moving the *pq*-gram pattern one step down. Now *anc* contains the labels of `r` and its $p - 1$ ancestors. The loop at line 10 moves the register *sib* from



(a) Moving the *pq*-Gram Pattern in the Tree.

(b) Shift Registers *anc* and *sib*.

(c) Resulting Index I.

Figure 2.9: Illustration of the *pq*-Gram Index Calculation.

left to right over the children of r in order to produce all the *pq*-grams with anchor point r and calls index recursively for each child of r. Overall, index adds $f_{\mathtt{r}} + q - 1$ label-tuples to I for each non-leaf node r, and 1 label-tuple for each leaf node. The *pq*-extended tree is calculated on the fly by an adequate initialization of the shift registers (lines 2, 5, 14–16).

Example 2.7. *Assume $p = 2$, $q = 3$, and the tree T_1 from Figure 2.4. The main data structures of the* index *algorithm are visualized in Figure 2.9. After the initialization,* index$(\mathsf{T}_1, 2, 3, \{\}, \mathtt{v}_1, (*, *))$ *is called. Line* 5 *initializes sib =* $(*, *, *)$*, and line 6 shifts the label of* $\mathtt{v}_1$ *into the register anc, yielding anc =* $(*, \mathtt{a})$*. Since* $\mathtt{v}_1$ *is not a leaf we enter the loop at line 10 and process all children of* $\mathtt{v}_1$*. The label of the first child,* $\mathtt{v}_2$*, is shifted into sib, yielding sib =* $(*, *, \mathtt{a})$*, and the first label-tuple* $(*, \mathtt{a}, *, *, \mathtt{a})$ *is added to the result set* I*. Figure 2.9(b) shows the values of anc and sib each time a label-tuple is added to* I*. The indentation illustrates the recursion. The table in Figure 2.9(c) shows the result relation* I *with the label-tuples in the order in which they are produced by the algorithm.*

pq-`Gram-Index` has runtime complexity $O(n)$ for a tree T, where $n = |N(\mathsf{T})|$: Each recursive call of `index` processes one node, and each node is processed exactly once.

2.5.2 Relational Implementation

The algorithm described above requires no particular encoding of trees. This section gives a scalable implementation for trees stored in a relational database. We use an interval representation of trees, where each node of a tree is represented by a pair of numbers (interval). The interval encoding is a technique for storing hierarchical data in relations [6, 7] and has been used to store and query XML data [1, 15, 55].

We associate a unique index number to each tree in the set. Each node of a tree is then represented as a quadruple of tree index, node label, and left and right endpoint of the node's interval.

Definition 2.7 (Interval Encoding)**.** *An* interval encoding *of a tree* T *is a relation* R *that for each node* $\mathsf{v} \in \mathsf{T}$ *contains a tuple* $(id(\mathsf{T}), \lambda(\mathsf{v}), lft, rgt)$*;* $id(\mathsf{T})$ *is a unique identifier of the tree* T*,* $\lambda(\mathsf{v})$ *is the label of* v*,* lft *and* rgt *are the endpoints of the interval representing the node.* lft *and* rgt *are constrained as follows:*

- $lft < rgt$ *for all* $(id, lbl, lft, rgt) \in R$*,*
- $lft_{\mathsf{a}} < lft_{\mathsf{d}}$ *and* $rgt_{\mathsf{a}} > rgt_{\mathsf{d}}$ *if node* a *is an ancestor of* d*, and* $(id(\mathsf{T}), \lambda(\mathsf{a}), lft_{\mathsf{a}}, rgt_{\mathsf{a}}) \in R$*, and* $(id(\mathsf{T}), \lambda(\mathsf{d}), lft_{\mathsf{d}}, rgt_{\mathsf{d}}) \in R$*,*
- $rgt_{\mathsf{v}} < lft_{\mathsf{w}}$ *if node* v *is a left sibling of node* w*, and* $(id(\mathsf{T}), \lambda(\mathsf{v}), lft_{\mathsf{v}}, rgt_{\mathsf{v}}) \in R$*, and* $(id(\mathsf{T}), \lambda(\mathsf{w}), lft_{\mathsf{w}}, rgt_{\mathsf{w}}) \in R$*,*
- $rgt = lft + 1$ *if node* v *is a leaf node, and* $(id(\mathsf{T}), \lambda(\mathsf{v}), lft, rgt) \in R$*.*

We get an interval encoding for a tree by traversing the tree in preorder, using an incremental counter that assigns the left interval value *lft* to each node when it is visited first, and the right value *rgt* when it is visited last. Figure 2.10 shows an address tree of our application, where each node is annotated with the endpoints of the interval.

The interval encoding of a tree allows a scalable implementation of the algorithm *pq*-`Gram-Index` for a set of trees $\mathcal{F}$ stored in a relation `F` with schema $(treeID, label, lft, rgt)$. We define the following cursor:

```
cur = SELECT * FROM F ORDER BY treeID,lft
```

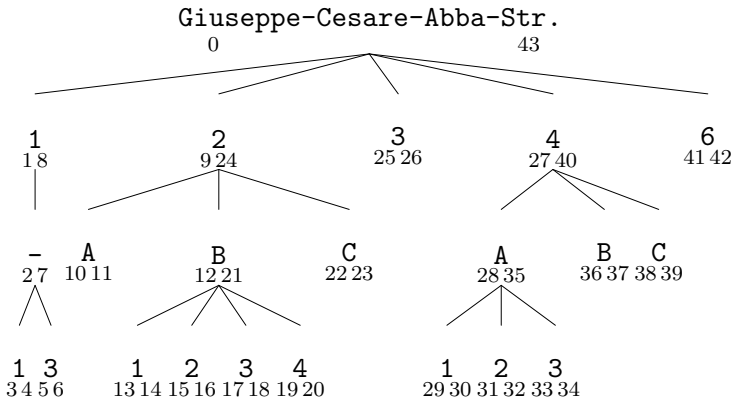


Figure 2.10: Address Tree in Interval Encoding.

Then with a single scan all trees can be processed, and each tree is processed node-by-node in preorder. Our experiments in Section 2.7.1 confirm the scalability of this approach to large trees.

The Algorithms 2.3 and 2.4 are adapted for the interval encoding and the changes are highlighted. Instead of a tree, *pq*-`Gram-Index` gets a cursor as an argument. `index` processes all nodes of the tree in preorder, and when it terminates the cursor points to the root node of the next tree in the set.

Algorithm 2.3: *pq*-`Gram-Index`(*cur*, *p*, *q*)

```
1 I : empty relation with schema (labels);
2 anc: shift register of size p (filled with *);
3 I ← index(cur, p, q, I, fetch(cur), anc);
4 return I;
```

`index` calls the following two functions:

- `isLeaf(v)`: Returns true iff `v` is a leaf node, i.e., lft(`v`) + 1 = rgt(`v`).
- `isDescendant(d, a)`: Returns true iff `d` is a descendant of `a`, i.e., lft(`a`) < lft(`d`) **and** rgt(`a`) > rgt(`d`) **and** treeId(`a`) = treeId(`d`) **and** `d` ≠ `null`.

With the interval encoding it is easier to check whether a node is a descendant than whether it is a child. In our algorithm this amounts to the same thing: When the loop in line 12 is entered the first time, `c` is the next node after `r` in preorder (or `null`). Thus, if `c` is a descendant of `r`, it must be a child. The recursive call in line 15 will process `c` and all its descendants, and

Algorithm 2.4: index($cur, p, q, \mathtt{I}, \mathtt{r}, anc$)

```
5  sib: shift register of size q (filled with *);
6  anc ← shift(anc, λ(r));
7  cur ← next(cur);
8  if isLeaf(r) then
9      I ← I ∪ (anc ∘ sib);
10 else
11     c ← fetch(cur);
12     while isDescendant(c, r) do
13         sib ← shift(sib, λ(c));
14         I ← I ∪ (anc ∘ sib);
15         I ← index(cur, p, q, I, c, anc);
16         c ← fetch(cur);
17     for k ← 1 to q − 1 do
18         sib ← shift(sib, *);
19         I ← I ∪ (anc ∘ sib);
20 return I;
```

set the cursor on the next node after the processed nodes. Again, if this is a descendant of $\mathtt{r}$, then it is a child. Thus the while-loop of Algorithm 2.4 is equivalent to the for-loop of Algorithm 2.2.

2.6 Sensitivity to Structural Changes

In this section we discuss the main properties of the *pq*-gram distance and compare it with the tree edit distance. We investigate two cases where the *pq*-gram distance behaves differently from the tree edit distance: structural and local changes. We consider the following standard edit operations [56]:

Update($\mathtt{T}, \mathtt{v}, \sigma$): Updating a node $\mathtt{v} \in N(\mathtt{T})$ means changing its label to $\sigma \in \mathbf{\Sigma}$.

Delete($\mathtt{T}, \mathtt{v}$): Deleting a node $\mathtt{v} \in N(\mathtt{T}) \setminus \{\text{root}(\mathtt{T})\}$ means substituting $\mathtt{v}$ with its children (preserving the order), i.e., remove $\mathtt{v}$ and connect $\mathtt{v}$'s children directly with $\mathtt{v}$'s parent node.

Insert($\mathtt{T}, \mathtt{v}, \mathtt{p}, i, k$): Inserting a new node $\mathtt{v} \notin N(\mathtt{T})$ as a child of a node $\mathtt{p} \in N(\mathtt{T})$ at position i means substituting k consecutive children $\mathtt{v}_i, \mathtt{v}_{i+1}, \ldots, \mathtt{v}_{i+k-1}$ of $\mathtt{p}$ with $\mathtt{v}$, and inserting them as children of $\mathtt{v}$ (pre-

serving the order). If $k = 0$, a leaf node is inserted, and the number of p's children increases by one.

The tree edit distance assigns a fixed cost to each operation. This disregards the fact that operations which change the structure (insert and delete) might have side effects on other nodes. For example, if a node is deleted, all children of this node are moved with their descendants to the parent node. This behavior leads to non-intuitive results, as shown in Figure 2.11: Tree T' is the result of deleting the leaves with labels g and k from T, T'' is obtained from T by deleting the nodes labeled c and e. Intuitively, T' and T are much more similar (in structure) than T'' and T, but the tree edit distance is 2 in both cases for a unit cost model.

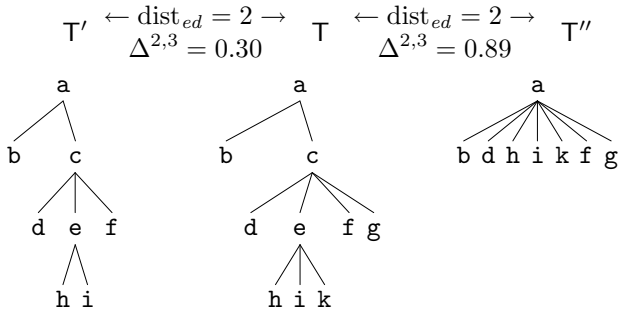


Figure 2.11: Tree Edit Distance and pq-Gram Distance for Structural Changes.

The pq-gram distance depends directly on the number of affected pq-grams, which depends on the number of descendants of v within distance p. Thus, changes to non-leaf nodes cost more than changes to leaves. The following theorem gives the number of pq-grams that contain a node v, which corresponds to the number of affected pq-grams if v is modified.

Theorem 2.2. *For a tree* T *with all leaf nodes at level* $d = \mathrm{depth}(\mathsf{T})$ *and a fixed fanout* $f > 1$ *for the non-leaf nodes, the number of pq-grams* $(p > 0, q > 0)$ *that contain a node* v *of level* $l = \mathrm{level}(\mathsf{v})$ *is:*

$$\mathrm{cnt}_{pq}(\mathsf{T}, \mathsf{v}) = q\,\mathrm{sgn}(l) + \begin{cases} \frac{f^p-1}{f-1}(f+q-1) & \text{if } p \le d-l \\ \frac{f^{d-l}-1}{f-1}(f+q-1) + f^{d-l} & \text{if } p > d-l. \end{cases}$$

Proof. Consider how the pq-gram pattern with q leaves and p non-leaves is shifted over the tree. The leaves of the pattern are shifted over all nodes of the tree but the root node, which gives q pq-grams for each non-root node

($\mathrm{sgn}(l)$ is 0 for the root, 1 for non-root nodes). If v is a non-leaf node, it appears in $f+q-1$ pq-grams as the anchor node, otherwise in a single pq-gram. While v is in the pq-gram we recursively move the pattern down the tree. We exit the recursion earlier if the anchor node of the pq-gram pattern is a leaf. For the case $p \leq d-l$, we get $(f+q-1)\sum_{i=0}^{p-1} f^i$, and for the case $p > d-l$, $(f+q-1)\sum_{i=0}^{d-l-1} f^i$ additional pq-grams that contain v. For the latter case we add the term f^{d-l} that accounts for the pq-grams that have one of the f^{d-l} leaf descendants of v as an anchor node. We evaluate the partial sum of the geometric series to get the formula in Theorem 2.2. □

Theorem 2.2 assumes a tree with all leaves at the same depth and a fixed fanout. If f is the *maximum* fanout of v and its descendants within distance p, then $\mathrm{cnt}_{pq}(\mathsf{T},\mathsf{v})$ is an upper bound for the number of pq-grams that contain v.

According to Theorem 2.2 the cost for changing a leaf node ($d=l$) is $q+1$, i.e., depends only on q. For non-leaf nodes the impact of p is prevalent, and we can control the sensitivity of the pq-gram distance to structural changes by choosing the value for p.

The difference between non-leaf and leaf nodes is relevant for hierarchical data, where values higher up in the hierarchy are more significant. For example, two streets with different house numbers (with subnumbers and apartment numbers) are considered more different than streets in which only apartment numbers differ.

We further investigate the case when part of a tree is missing, i.e., a subtree is deleted. The effect on the structure is limited as the remaining part of the tree is unchanged. An example of a subtree is a subnumber with all its apartment numbers. If it is missing in one address tree, a relatively high number of nodes changes. These changes should be weighted less than the same number of changes on different house numbers.

If a subtree is deleted, several modifications are applied within a small neighborhood. The affected sets of pq-grams overlap each other, and hence, these changes have less impact on the pq-gram distance than changes that are uniformly distributed over the tree. The following theorem gives the number of pq-grams that change with a subtree deletion.

Theorem 2.3. *Let* S *be the subtree of* T *consisting of* $\mathsf{v} \in N(\mathsf{T}) \setminus \{\mathrm{root}(\mathsf{T})\}$ *and all its descendants, and let* l *be the number of leaves of* S*, and let* i *be the number of non-leaf nodes. If all nodes of* S *are deleted or updated, then* $2l+iq+q-1$ *pq-grams change.*

Proof. All pq-grams of the subtree change. This are $2l + iq - 1$ pq-grams (Theorem 2.1). Further $\mathtt{v}$ appears as a sibling in q pq-grams. The sum is $2l + iq + q - 1$. □

Example 2.8. *We refer to Figure 2.11 and discuss the deletion of the subtree of* T *that consists of the node with label* $\mathtt{e}$ *(lets call the node* $\mathtt{v}$*) and all its descendants. An effect of this operation is that the following nodes are deleted:* $\mathtt{v}$ *plus the nodes labeled* $\mathtt{h}$*,* $\mathtt{i}$*, and* $\mathtt{k}$*. The number of* $2,3$*-grams that contain the node* $\mathtt{v}$ *is* $\mathrm{cnt}_{2,3}(\mathsf{T},\mathtt{v}) = 11$*, and* $q + 1 = 4$ *for the three other nodes. If these nodes did not share any pq-grams, the total number of affected pq-grams would be* $11 + 3 \times 4 = 23$*. However, as the deleted nodes build a subtree with* $l = 3$ *leaves and* $i = 1$ *non-leaf nodes, they do share pq-grams, and the total number of changing* $2,3$*-grams is only* $2l + iq + q - 1 = 11$*.*

2.7 Experiments

2.7.1 Scalability

We compare the scalability of our algorithm with the tree edit distance [56] and the tree embedding distance [23], and we investigate the influence of the parameters p and q on the scalability of the pq-gram distance.

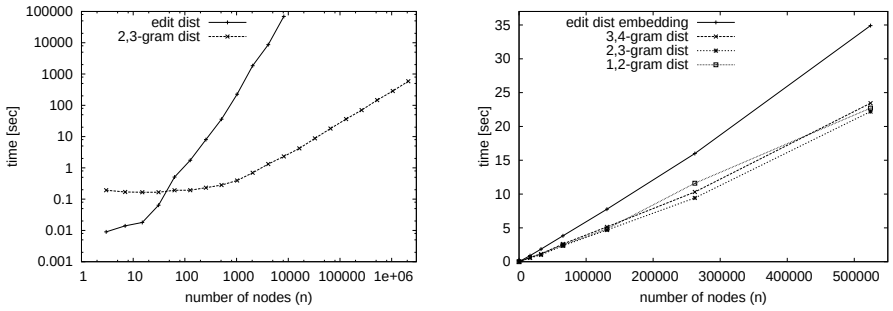
As a test set we produce pairs of trees $(\mathsf{T}_1, \mathsf{T}_2)$ of size $|N(\mathsf{T}_1)| = |N(\mathsf{T}_2)| = n$, where n ranges from 3 to 2×10^6 nodes. The depth of the trees is $\log(n)$ and the labels for each tree are randomly chosen from a set of n different labels.

Figure 2.12(a) shows the runtimes of tree edit distance and $2,3$-gram distance calculations for different tree sizes. For the tree edit distance we use the implementation of Zhang and Shasha[2], whereas for the pq-gram distance we use the relational implementation described in Section 2.5.2. For very small trees edit distance is faster than pq-gram distance. The reason being that our algorithm writes all intermediate results to the disk, while the edit distance algorithm runs in the main memory. Therefore the overhead for disk access in this range masks the actual computing time for the distance. This effect can easily be prevented by keeping all data in main memory. For large trees the computation time for the tree edit distance grows very fast. For trees of size 10,000 it is already more than 27 hours, therefore we could not run our experiment for even larger trees. For the pq-gram distance the computation time is almost linear in the tree size.

Figure 2.12(b) compares the pq-gram distance for varying parameters with the tree embedding distance. We use our own implementation for tree embed-

[2] `http://www.cs.nyu.edu/cs/faculty/shasha/papers/tree.html`

ding distance according to the algorithm of Garofalakis and Kumar [23]. For the comparison both algorithms run in main memory. The *pq*-gram distance is slightly faster, and varying values for p and q have little impact on the scalability of the *pq*-gram distance calculation.



(a) Tree Edit Distance. (b) Tree Embedding Distance.

Figure 2.12: Scalability Results.

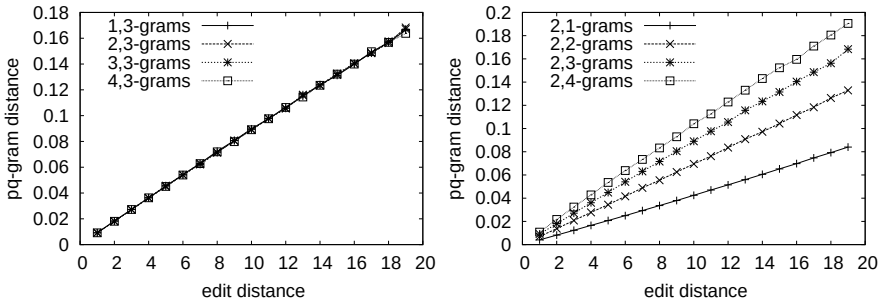
2.7.2 Sensitivity to Structural Changes

In Section 2.6 we point out that the *pq*-gram distance weights deletions of non-leaf nodes more than deletions of leaves, and the sensitivity to structural changes is controlled by the parameters p and q. We show this property in an experiment, where only non-leaf nodes or only leaf nodes are deleted for varying parameters, and calculate the *pq*-gram distance for both cases.

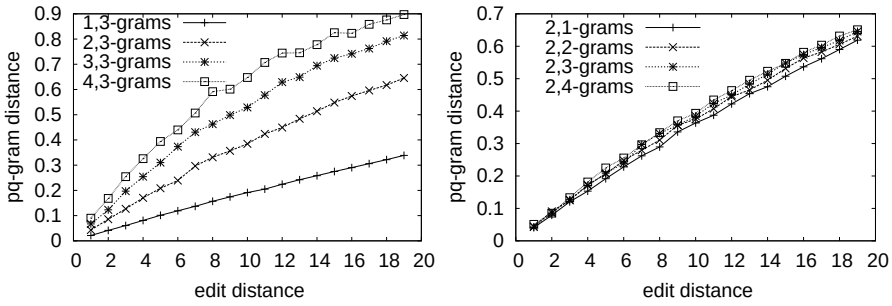
We create an artificial tree T with 144 nodes, 102 leaves, and depth 6. Each non-leaf has a fanout of between 2 and 5. Figure 2.13 shows the *pq*-gram distance for different numbers of leaf and non-leaf deletions. Each value in Figure 2.13 is an average over 100 runs.

For leaf node deletions only q has an influence (see Figure 2.13(a)). For the deletion of non-leaf nodes q has a small impact compared to p (see Figure 2.13(b)). This confirms our analytical results. Sensitivity to changes in the leaves depends only on q, and we can emphasize structural sensitivity with higher values of p. For deletions of non-leaf nodes the *pq*-gram distance is longer than for deletions of leaf nodes.

We further investigate the difference in the *pq*-gram distance for deleting a subtree or the same number of nodes randomly distributed all over the tree. For the experiment we use the same tree T as above. We randomly choose a node $\mathsf{v} \in \mathsf{T} \setminus \{\text{root}(\mathsf{T})\}$ and delete v and all its descendants. The tree edit distance between T and the resulting tree T' is the number of nodes in the



(a) Deletion of Leaf Nodes.



(b) Deletion of Non-Leaf Nodes.

Figure 2.13: Properties of the *pq*-Gram Distance.

deleted subtree. In Figure 2.14 we compare the results to distributed changes (average on 100 runs). We can see that local changes (subtree deletions) are cheaper than distributed changes.

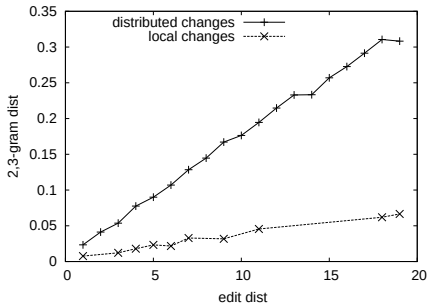


Figure 2.14: Distributed vs. Local Changes.

2.7.3 Matchmaking with Real Data

To test the accuracy for real world data we use the address tables RO and LR described in Section 2.2. We build the address trees for all streets in both tables and get the sets R and L. Each tree T in one of the tree sets R and L represents a street with all the addresses in that street. Set R from RO consists of 302 trees with 52,509 nodes in total, reflecting 43,187 addresses. Set L from table LR consists of 300 trees with 53,464 nodes and 44,447 addresses.

We say that two trees $\mathsf{T} \in \mathcal{F}$ and $\mathsf{T}' \in \mathcal{F}'$ *match* if T has only one nearest neighbor in $\mathcal{F}'$, namely T', and vice versa. For each distance function dist_x we compute a mapping $M_x \in \mathcal{F} \times \mathcal{F}'$ between all pairs of matching trees. Furthermore, we create a mapping, M_c, by hand with the correct pairs of trees, i.e., with all pairs of trees that represent the same street in the real world. We define the *accuracy* of M_x with respect to M_c as $a = \frac{|M_x \cap M_c|}{|M_c|}$. The false positives are computed as $M_x \setminus M_c$.

We compute a mapping for the tree edit distance dist_{ed}, the *pq*-gram distance $\Delta^{p,q}$, the tree embedding distance dist_{emb}, and the node intersection dist_i. The node intersection is a simple algorithm that completely ignores the structure of the tree. It is computed in the same way as the *pq*-gram distance, the only difference being that the index of a tree consists of the bag of all its node labels.

The results for the address tables RO and LR are shown in Table 2.1. There are two streets in RO that do not exist in LR, thus $|M_c| = 300$ for the calculation of the accuracy. The efficiency of the approximations is clearly greater than

that of the tree edit distance: All of them can be computed within about five minutes, whereas the tree edit distance takes more than 52 hours.

	accuracy	correct	false pos.	runtime
dist_{ed}	82.7%	248	9	187,538s
$\Delta^{1,2}$	78.3%	235	5	181s
$\Delta^{2,3}$	77.3%	232	4	204s
$\Delta^{3,2}$	79.3%	238	2	180s
dist_{emb}	69.0%	207	8	313s
dist_{i}	66.3%	199	12	82s

Table 2.1: Accuracy of the Tree Edit Distance and its Approximations.

The *pq*-gram distance clearly outperforms the other approximations with respect to both, number of correct matches and number of false positives for all tested parameters. The number of false positives is even smaller than with the tree edit distance. The tree embedding distance does not perform much better than the simple node intersection. We will now briefly discuss how the tree embedding distance works, and why it performs poorly on typical address trees.

The tree embedding distance is computed by building a parsing hierarchy for a tree T. In each phase i a tree T^i is obtained by nodes of the tree T^{i-1}. The parsing procedure starts with the tree $\mathsf{T}^0 = \mathsf{T}$, and it stops if $|\mathsf{T}^i| = 1$. Figure 2.15 shows the parse trees T^0, T^1 and T^2 for an example tree T that is shaped like a typical address tree. In our illustration we use different types of brackets to label the newly created nodes for the different situations in which nodes are merged:

- Contiguous sequences of children are split into blocks of length 2 and 3, and the blocks are contracted. The nodes $\mathtt{1}, \mathtt{3}, \mathtt{5}, \mathtt{10}, \mathtt{11}$ of T^0 become the two new nodes `(1,3,5)` and `(10,11)` of T^1.
- A lone leaf child is merged with the parent node if it is the leftmost lone leaf. The nodes `SN` and `3` in T^0 become the new node {`SN,3`} in T^1.
- Chains (paths of degree-two nodes) are split into blocks of length 2 and 3, and the blocks are contracted. The nodes $\mathtt{2}, \mathtt{A}$, `(1,3,7)` of T^1 become the new node `[2,A,(1,3,7)]` of T^2.

Each node in the parsing hierarchy corresponds to a set of nodes ("valid subtree") in the original tree. The bag of all valid subtrees corresponding to all nodes of the final hierarchical parsing structure (tagged with a phase label

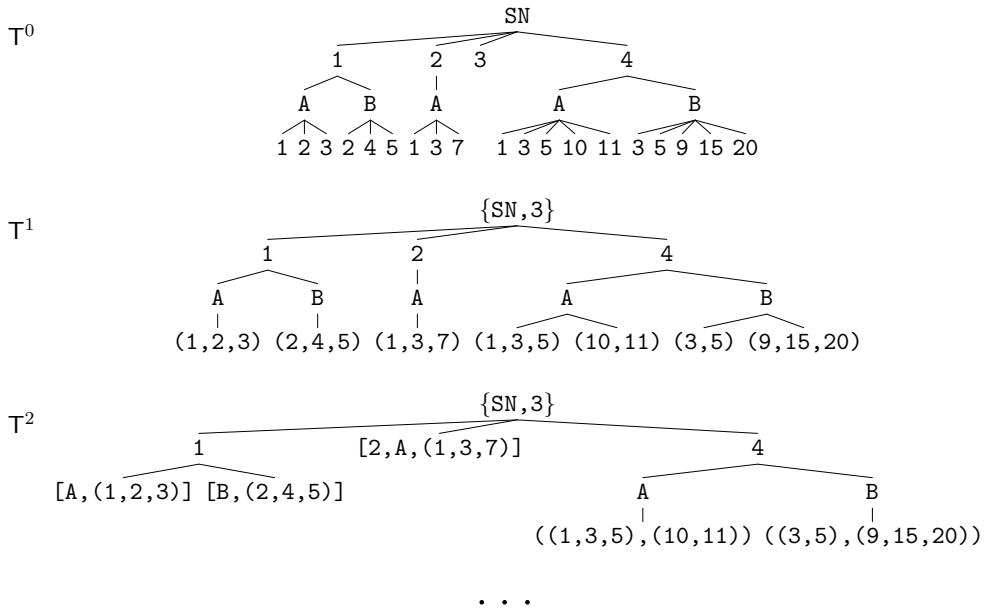


Figure 2.15: Parse Trees for an Example Tree T.

to distinguish between subtrees in different phases) is treated the same way we treat the *pq*-gram index in order to calculate the distance. It contains nodes corresponding to (1) single nodes, (2) node chains with parent-child relationship, (3) contiguous leaf children, and (4) subtrees. Single nodes contain no structural information, parent-child chains only vertical, leaf sequences contain only horizontal structure information. Only subtrees reflect both, horizontal and vertical structure. Table 2.2 gives an overview of how many nodes of each type are obtained in each phase for the example tree. We can see that 65% of all nodes are single nodes containing no structural information. Only 19% of nodes correspond to subtrees.

Trees with many leaves at the deepest level are parsed bottom-up, and the structure of the inner nodes has less impact on the distance. For this reason the tree embedding distance performs only slightly better than a simple node intersection on our real world data.

2.8 Conclusion

Our work is motivated by a data integration scenario from the Municipality of Bozen, where data from different sources have to be integrated and no common keys exist. Data have to be joined over residential addresses, which

phase	single node	chain	cont. leaf	subtree
0	29	-	-	-
1	8	1	7	-
2	4	1	2	3
3	2	-	-	4
4	1	-	-	3
5	-	-	-	2
6	-	-	-	1
total	44	2	9	13
	65%	3%	13%	19%

Table 2.2: Types of Valid Subtrees in the Different Phases.

in practice have some undesirable properties, and exact joins completely fail. To overcome these problems we introduced address trees as a representation of residential addresses. This reduces the integration to an approximate join on address trees.

We presented a new distance measure, the *pq*-gram distance, for ordered labeled trees as an effective and efficient approximation for the well known tree edit distance. We provided an algorithm for the computation of *pq*-grams in $O(n)$ time, where n is the number of tree nodes. Based on the index the *pq*-gram distance can be computed in $O(n \log n)$ time. We discussed a scalable implementation using an interval representation of trees in a relational database.

The *pq*-gram distance behaves differently from the tree edit distance for structural and local changes. It gives more weight to edit operations that cause big changes in the tree structure. This property turned out to be relevant in our application domain.

Detailed experiments on real and synthetic data confirmed that the *pq*-gram distance is orders of magnitude faster than the tree edit distance for large trees. The accuracy of the *pq*-gram distance for real world data from the municipality domain turned out to be clearly better than other approximations of the tree edit distance.

In the future we will investigate additional application areas and apply the *pq*-gram distance for data cleaning and the comparison of XML data.

Chapter 3

Updating the pq-Gram Index

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 re-construction 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 that are 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.*

3.1 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 3.1. T_0 is a document with a hierarchical structure (e.g., the DBLP file, 211MB). $\mathbf{I}_0$ 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 $\mathbf{I}_0$, (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.

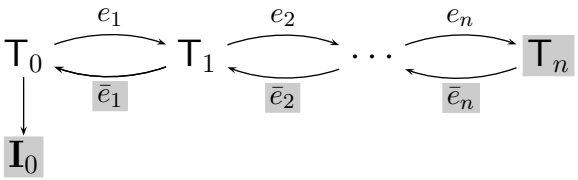


Figure 3.1: Application Scenario.

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.

More specifically, the contributions are the following:

- 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 [3], which generalize *q*-grams [49]. 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 chapter proceeds as follows: Section 3.2 discusses related work, Section 3.3 defines the *pq*-gram index, and Section 3.4 gives an outline on our approach. Section 3.5 develops the incremental maintenance for a single edit operation, Section 3.6 generalizes to a sequences of edit operations and proves the correctness. In Section 3.7 we discuss the computation of the index maintenance functions. Section 3.8 discusses the implementation. Section 3.9 gives experimental results. Section 3.10 summarizes and points to future research directions.

3.2 Related Work

Guha et al. [26] 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. [27] 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 [52] propose a framework for detecting duplicates. In change detection scenarios two versions of the same document are given and the difference is computed [12, 37]. Index use and maintenance is not addressed.

Structural joins [1, 30] 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 [5, 13, 34, 39], while others allow approximate answers [42, 44]. Schenkel et. al. [45] 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 [3, 23, 53]. None of these works addresses index maintenance.

Our index is based on the *pq*-gram distance [3], an approximation of the tree edit distance. Augsten et al. [3] 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.

3.3 The *pq*-Gram Index

3.3.1 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 *dummy 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. A node can have at most one parent, and 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* S of 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*, $\mathcal{F}$, 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 [56] that transform T_i into T_j:

- $\textsc{ins}(\mathsf{n}, \mathsf{v}, k, m)$: *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 $\bar{e}_j = \textsc{del}(\mathsf{n})$.

- $\textsc{del}(\mathsf{n})$: *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

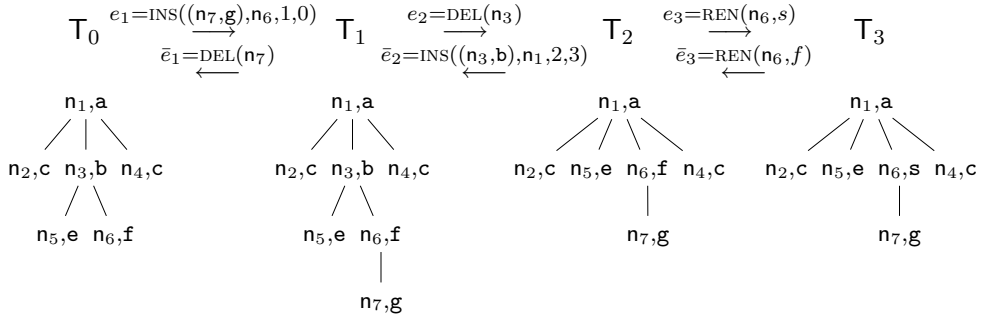


Figure 3.2: Sequence of Edit Operations that Transforms Tree T_0 into T_3.

order). The inverse operation is $\bar{e}_j = \text{INS}(\mathsf{n}, \mathsf{v}, k, (k + f_\mathsf{n} - 1))$, where n is the k-th child of v in T_i, and f_n is the fanout of n.

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

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 3.2 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:

$$(A \cap B) \cup (A \setminus B) = A \tag{3.1}$$
$$A \setminus (A \setminus B) = A \cap B \tag{3.2}$$
$$(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C) \tag{3.3}$$
$$(A \setminus B) \cup B = A \cup B \tag{3.4}$$

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

3.3.2 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 3.1. *pq*-Gram. *Let* T *be a tree,* a *be a node in* $N(\mathsf{T})$*,* $p > 0$*,* $q > 0$*, and let* $\mathsf{T}^{p,q}$ *be* T *extended with dummy 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.*

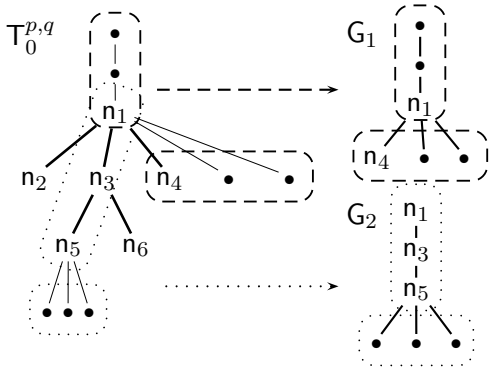


Figure 3.3: Part of $\mathsf{T}_0^{p,q}$ and Two 3, 3-Grams of Tree T_0.

*A pq-*gram*,* G*, of* T *with* anchor node a *is a subtree of* $\mathsf{T}^{p,q}$ *that is composed of the following nodes:* p *nodes* $\mathsf{a}_{p-1}, \ldots, \mathsf{a}_1, \mathsf{a}$*, denoted as* p-part *of* G*, where* a_i *is the ancestor of* a *at distance* i*;* q *contiguous children* $\mathsf{c}_i, \ldots, \mathsf{c}_{i+q-1}$ *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 $(\mathsf{a}_{p-1}, \ldots, \mathsf{a}_1, \mathsf{a}, \mathsf{c}_i, \ldots, \mathsf{c}_{i+q-1})$.

Example 3.1. *Consider tree* T_0 *in Figure 3.2. Figure 3.3 shows part of the extended tree* $\mathsf{T}_0^{p,q}$ *(*$p = q = 3$*) together with two pq-grams of* T_0*, namely* $\mathsf{G}_1 = (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_4, \bullet, \bullet)$ *with anchor node* n_1 *and* $\mathsf{G}_2 = (\mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_5, \bullet, \bullet, \bullet)$ *with anchor node* n_5*. The total number of pq-grams of* T_0 *is 13.*

Definition 3.2. 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*.*

Example 3.2. *The pq-gram profiles of* T_0 *and* T_2 *in Figure 3.2 are given as follows:*

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

l	h(l)	l	h(l)
*	0	e	8
a	1	f	4
b	3	g	7
c	2	h	5
d	6	s	9

(a) Hash Function.

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) *pq*-Gram Index.

Figure 3.4: A Hash Function and Part of the *pq*-Gram Index of T_0.

With $\lambda(\mathsf{G}) = (\lambda(\mathsf{n}_1), \ldots, \lambda(\mathsf{n}_{p+q}))$ 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 3.3. *pq*-Gram Index. *Let* T *be a tree with profile* $\mathbf{P}_\mathsf{T}$, $p > 0$, $q > 0$. *The pq-gram index,* $\mathbf{I}$, *of tree* T *is the bag of all label-tuples of* T,

$$\mathbf{I}(\mathsf{T}) = \biguplus_{\mathsf{G} \in \mathbf{P}_\mathsf{T}} \lambda(\mathsf{G}). \tag{3.5}$$

We store the *pq*-gram index of a forest $\mathcal{F} = \{\mathsf{T}_1, \ldots, \mathsf{T}_N\}$ 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 [33]) 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 3.4). Note that the only operation we need to perform on labels is to check equality.

Example 3.3. *Figure 3.4 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 $\mathcal{F}$ returns all trees of the forest that are similar to the search tree, i.e., the set $\{\mathsf{T} \in \mathcal{F} \mid \mathrm{TDist}(\mathsf{X}, \mathsf{T}) < \tau\}$, where TDist is a distance measure between trees and τ is a threshold value. We use the *pq*-gram distance [3] 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 $\mathrm{dist}^{p,q}(\mathsf{T}, \mathsf{T}') = 1 - 2\frac{|\mathbf{I}(\mathsf{T}) \cap \mathbf{I}(\mathsf{T}')|}{|\mathbf{I}(\mathsf{T}) \uplus \mathbf{I}(\mathsf{T}')|}$.

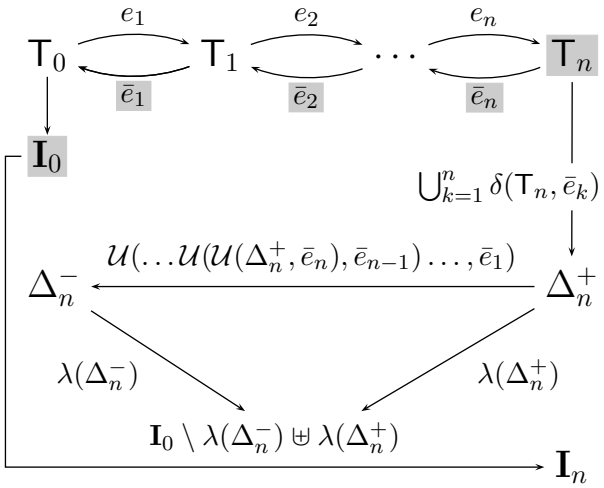


Figure 3.5: Application Scenario and Solution.

3.4 Outline

In the following we give an outline of our approach to incrementally update the index. Figure 3.5 shows the application scenario and summarizes the solution:

Input: The old index, $\mathbf{I}_0$, the log of inverse edit operations, $(\bar{e}_1, \ldots, \bar{e}_n)$, and the resulting tree, T_n (shaded in Figure 3.5).

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

Solution: The solution consists of three steps:

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

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 $\mathbf{I}_0$.

Note the difference between the profile and the index of a tree. The profile, $\mathbf{P}$, is a set of *pq*-grams, the index, $\mathbf{I} = \lambda(\mathbf{P})$, 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.

3.5 Single Edit Step

In this section we discuss the effect of a single edit operation on the profile of a tree. Figure 3.6 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 3.7 and Section 3.8.

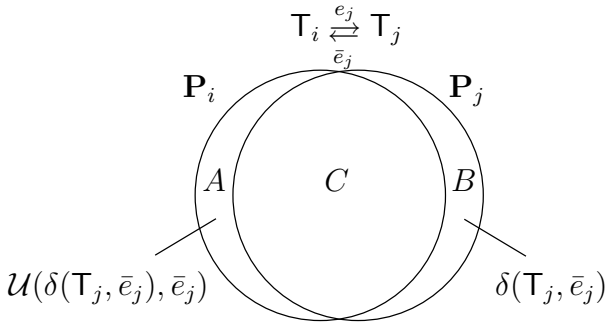


Figure 3.6: Profile Update for an Edit Operation $\bar{e}_j$.

3.5.1 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 3.4. 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*

$$\delta(\mathsf{T}_j, \bar{e}_j) \quad = \quad \begin{cases} \mathbf{P}_j \setminus \mathbf{P}_i & \textit{iff } \exists \mathsf{T}_i : \mathsf{T}_i = \bar{e}_j(\mathsf{T}_j) \\ \emptyset & \textit{otherwise} \end{cases} \tag{3.6}$$

$\mathbf{P}_i$ 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 3.6.

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

$$\mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftrightarrow \mathsf{v} \in N(\mathsf{G}) \wedge \exists \mathsf{c} \in C : \mathsf{c} \in N(\mathsf{G}). \tag{3.7}$$

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

$$\mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftrightarrow \mathsf{n} \in N(\mathsf{G}). \tag{3.8}$$

Proof. Each *pq*-gram $\mathsf{G} \in \mathbf{P}_j$ 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., $\mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_j)$.
Insert. $\mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_j) \Rightarrow \mathsf{v} \in N(\mathsf{G}) \wedge \exists \mathsf{c} \in C : \mathsf{c} \in N(\mathsf{G})$ is equivalent to $\mathsf{v} \notin N(\mathsf{G}) \vee \forall \mathsf{c} \in C : \mathsf{c} \notin N(\mathsf{G}) \Rightarrow \mathsf{G} \notin \delta(\mathsf{T}_j, \bar{e}_j)$: If $\mathsf{v} \notin N(\mathsf{G})$, 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. $\mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftarrow \mathsf{v} \in N(\mathsf{G}) \wedge \exists \mathsf{c} \in C : \mathsf{c} \in N(\mathsf{G})$: As n is inserted between v and c, all *pq*-grams that contain both of them are affected.
Delete. $\mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_j) \Rightarrow \mathsf{n} \in N(\mathsf{G})$ is equivalent to $\mathsf{n} \notin N(\mathsf{G}) \Rightarrow \mathsf{G} \notin \delta(\mathsf{T}_j, \bar{e}_j)$: If n is not in G, no node of G is affected. $\mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftarrow \mathsf{n} \in N(\mathsf{G})$: n does not exist in T_i. If n is in G, G is *only* in $\mathbf{P}_j$.
Rename. $\mathsf{n} \notin N(\mathsf{G}) \Rightarrow \mathsf{G} \notin \delta(\mathsf{T}_j, \bar{e}_j)$: If n is not in G, no node of G is affected. $\mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_j) \Leftarrow \mathsf{n} \in N(\mathsf{G})$: $\lambda(\mathsf{n}) = l$ in T_i, but $\lambda(\mathsf{n}) \neq l$ in T_j. As $\mathsf{G} \in \mathbf{P}_j$, $\lambda(\mathsf{n}) \neq l$ in G. Thus, if n is in G, G is *only* in $\mathbf{P}_j$. □

3.5.2 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 3.6). 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 3.5. 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 $\delta(\mathsf{T}_j, \bar{e}_j) \subseteq \mathbf{p}_j \subseteq \mathbf{P}_j$. The* profile update function*, $\mathcal{U} : 2^{\mathbf{P}_j} \to 2^{\mathbf{P}_i}$, is defined as follows:*

$$\mathcal{U}(\mathbf{p}_j, \bar{e}_j) = \mathbf{p}_j \setminus \delta(\mathsf{T}_j, \bar{e}_j) \cup \delta(\mathsf{T}_i, e_j) \tag{3.9}$$

If $\mathbf{p}_j = \delta(\mathsf{T}_j, \bar{e}_j)$, 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 $\mathbf{p}_j = \mathbf{P}_j$, the original profile $\mathbf{P}_i$ is computed from $\mathbf{P}_j$. Due to the symmetry of the scenario also the opposite direction holds:

$$\mathbf{P}_i = \mathcal{U}(\mathbf{P}_j, \bar{e}_j) \qquad \mathbf{P}_j = \mathcal{U}(\mathbf{P}_i, e_j) \tag{3.10}$$

3.6 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.

3.6.1 Incremental Index Update

Consider a sequence of edit operations as shown in Figure 3.5. 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 3.6. *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:*

$$\text{Invariant } pq\text{-grams:} \quad \mathbf{C}_n = \mathbf{P}_0 \cap \cdots \cap \mathbf{P}_n \tag{3.11}$$

$$\text{Old } pq\text{-grams:} \quad \Delta_n^- = \mathbf{P}_0 \setminus \mathbf{C}_n$$

$$\text{New } pq\text{-grams:} \quad \Delta_n^+ = \mathbf{P}_n \setminus \mathbf{C}_n \tag{3.12}$$

Figure 3.7 illustrates these sets for a scenario with $n = 2$. The two shaded regions in Figure 3.7(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.

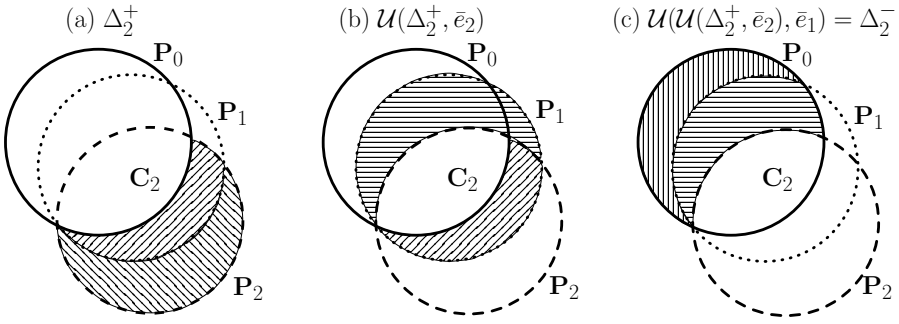


Figure 3.7: Profiles for Two Edit Operations.

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 3.7(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 3.7(b) and is passed to the next call of the profile update function. Figure 3.7(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 3.2. *Let T_0 be a tree with index $\mathbf{I}_o = \lambda(\mathbf{P}_0)$ that is transformed to T_n with index $\mathbf{I}_n = \lambda(\mathbf{P}_n)$ by a sequence of n edit operations. The new index, $\mathbf{I}_n$, can be computed from the old index, $\mathbf{I}_0$, as follows:*

$$\mathbf{I}_n = \mathbf{I}_0 \setminus \lambda(\Delta_n^-) \uplus \lambda(\Delta_n^+). \tag{3.13}$$

Proof. First we show that replacing the old by the new *pq*-grams in $\mathbf{P}_0$ results in $\mathbf{P}_n$: $\mathbf{P}_0 \setminus \Delta_n^- \overset{(3.12)}{=} \mathbf{P}_0 \setminus [\mathbf{P}_0 \setminus \mathbf{C}_n] \overset{(3.2)}{=} \mathbf{P}_0 \cap \mathbf{C}_n \overset{(3.11)}{=} \mathbf{C}_n$, thus $\mathbf{P}_0 \setminus \Delta_n^- \cup \Delta_n^+ = \mathbf{C}_n \cup \Delta_n^+ \overset{(3.12)}{=} \mathbf{C}_n \cup [\mathbf{P}_n \setminus \mathbf{C}_n] \overset{(3.4)(3.11)}{=} \mathbf{P}_n$. As $\mathbf{I}_n = \lambda(\mathbf{P}_n)$ it follows that

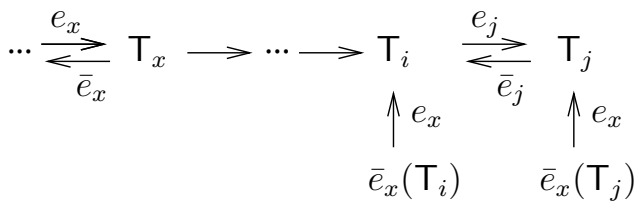


Figure 3.8: Setting in Lemma 3.3.

$\mathbf{I}_n = \lambda(\mathbf{P}_0 \setminus \Delta_n^- \cup \Delta_n^+)$. 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 $\mathsf{G} \in \Delta_n^-$ that is subtracted from $\mathbf{P}_0$ the respective label-tuple $\lambda(\mathsf{G}) \in \lambda(\Delta_n^-)$ is subtracted from $\lambda(\mathbf{P}_0)$. As $\Delta_n^- \subseteq \mathbf{P}_0$ (3.12), also $\lambda(\Delta_n^-) \subseteq \lambda(\mathbf{P}_0)$. Thus for each subtracted label-tuple $\lambda(\mathsf{G}) \in \lambda(\Delta_n^-)$ there is a *pq*-gram, $\mathsf{G} \in \Delta_n^-$, 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 $\Delta_n^+ \overset{(3.12)}{=} \mathbf{P}_n \setminus \mathbf{C}_n$. □

3.6.2 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.

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 3.8). 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 3.3. *Let e_j be an edit operation that transforms T_i into T_j (see Figure 3.8). For an edit operation $\bar{e}_x$ that transforms T_i to $\bar{e}_x(\mathsf{T}_i)$ and T_j to $\bar{e}_x(\mathsf{T}_j)$,*

$$\delta(\mathsf{T}_i, \bar{e}_x) \setminus \delta(\mathsf{T}_i, e_j) = \delta(\mathsf{T}_j, \bar{e}_x) \setminus \delta(\mathsf{T}_j, \bar{e}_j). \tag{3.14}$$

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. (3.14) is equivalent to

$$\mathsf{G} \in \delta(\mathsf{T}_i, \bar{e}_x) \wedge \mathsf{G} \notin \delta(\mathsf{T}_i, e_j) \Leftrightarrow \mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_x) \wedge \mathsf{G} \notin \delta(\mathsf{T}_j, \bar{e}_j). \tag{3.15}$$

We first show (3.15) from left to right and denote the left side with L. From L follows $\mathsf{G} \in \mathbf{P}_i \cap \mathbf{P}_j$, i.e., the *pq*-grams in $\delta(\mathsf{T}_i, \bar{e}_x)$ that are not replaced by e_j are also in $\mathbf{P}_j$: $\mathsf{G} \in \delta(\mathsf{T}_i, \bar{e}_x) \Rightarrow \mathsf{G} \in \mathbf{P}_i$ as $\delta(\mathsf{T}_i, \bar{e}_x) \subseteq \mathbf{P}_i$ (3.6); $\mathsf{G} \notin \delta(\mathsf{T}_i, e_j) \Rightarrow \mathsf{G} \notin \mathbf{P}_i \setminus \mathbf{P}_j$, as $\delta(\mathsf{T}_i, e_j) = \mathbf{P}_i \setminus \mathbf{P}_j$ (3.6); from $\mathsf{G} \in \mathbf{P}_i$ and $\mathsf{G} \notin \mathbf{P}_i \setminus \mathbf{P}_j$ follows $\mathsf{G} \in \mathbf{P}_i \cap \mathbf{P}_j$. We distinguish for $\bar{e}_x$:

Rename. We first show $L \Rightarrow \mathsf{G} \notin \delta(\mathsf{T}_j, \bar{e}_j)$: $\mathsf{G} \in \mathbf{P}_i \cap \mathbf{P}_j$ implies $\mathsf{G} \notin \delta(\mathsf{T}_j, \bar{e}_j)$, as $\delta(\mathsf{T}_j, \bar{e}_j) = \mathbf{P}_j \setminus \mathbf{P}_i$ (3.6). Now we show $L \Rightarrow \mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_x)$: L implies that the renamed node n is a node of G ($\mathsf{G} \in \delta(\mathsf{T}_i, \bar{e}_x) \Rightarrow \mathsf{n} \in N(\mathsf{G})$ (3.8)). As G is in $\mathbf{P}_j$ ($L \Rightarrow \mathsf{G} \in \mathbf{P}_i \cap \mathbf{P}_j$) 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: $\mathsf{n} \in N(\mathsf{G}) \wedge \mathsf{G} \in \mathbf{P}_j \Rightarrow \mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_x)$ (3.8).

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 $C = \{\mathsf{c}_k, \ldots, \mathsf{c}_m\}$ move under n. We show $L \Rightarrow \mathsf{G} \notin \delta(\mathsf{T}_j, \bar{e}_j)$: $L \Rightarrow \mathsf{G} \in \mathbf{P}_i \cap \mathbf{P}_j \Rightarrow \mathsf{G} \notin \delta(\mathsf{T}_j, \bar{e}_j)$. We show $L \Rightarrow \mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_x)$: L implies that (a) the parent of the inserted node and at least on of its children are in G ($\mathsf{G} \in \delta(\mathsf{T}_i, \bar{e}_x) \Rightarrow \mathsf{v} \in N(\mathsf{G}) \wedge \exists \mathsf{c} \in C : \mathsf{c} \in N(\mathsf{G})$ (3.7)), and (b) that $\mathsf{G} \in \mathbf{P}_j$ ($L \Rightarrow \mathsf{G} \in \mathbf{P}_i \cap \mathbf{P}_j$). With (a), (b): $\mathsf{v} \in N(\mathsf{G}) \wedge \exists \mathsf{c} \in C : \mathsf{c} \in N(\mathsf{G}) \wedge \mathsf{G} \in \mathbf{P}_j \Rightarrow \mathsf{G} \in \delta(\mathsf{T}_j, \bar{e}_x)$. (3.15) 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. □

3.6.3 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 3.7 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 3.4. *Let $L = (e_1, \ldots, e_n)$ be a sequence of edit operations that transforms T_0 into T_n, $\mathsf{T}_i = e_i(\mathsf{T}_{i-1})$, $1 \leq i \leq n$.*

$$\mathbf{P}_i = \mathbf{P}_0 \setminus \underbrace{\bigcup_{k=1}^{i} \delta(\mathsf{T}_{k-1}, e_k)}_{\mathbf{A}_i} \cup \underbrace{\bigcup_{k=1}^{i} \delta(\mathsf{T}_i, \bar{e}_k)}_{\mathbf{B}_i} \tag{3.16}$$

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

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

$$\begin{aligned}
\mathbf{P}_{i+1} &\overset{(3.10)}{=} \mathcal{U}(\mathbf{P}_i, e_{i+1}) \\
&\overset{(3.9)}{=} [\mathbf{P}_0 \setminus \mathbf{A}_i \cup \mathbf{B}_i] \setminus \delta(\mathsf{T}_i, e_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \\
&\overset{(3.3)}{=} \mathbf{P}_0 \setminus [\mathbf{A}_i \cup \delta(\mathsf{T}_i, e_{i+1})] \cup \\
&\qquad [\mathbf{B}_i \setminus \delta(\mathsf{T}_i, e_{i+1})] \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1})
\end{aligned}$$

$$\begin{aligned}
&\mathbf{A}_i \cup \delta(\mathsf{T}_i, e_{i+1}) = \bigcup_{k=1}^{i+1} \delta(\mathsf{T}_{k-1}, e_k) = \mathbf{A}_{i+1} \\
&\mathbf{B}_i \setminus \delta(\mathsf{T}_i, e_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \\
&\qquad \overset{(3.14)}{=} \bigcup_{k=1}^{i} \delta(\mathsf{T}_{i+1}, \bar{e}_k) \setminus \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \\
&\qquad \overset{(3.4)}{=} \bigcup_{k=1}^{i} \delta(\mathsf{T}_{i+1}, \bar{e}_k) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) = \mathbf{B}_{i+1}
\end{aligned} \tag{3.17}$$

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

Lemma 3.5. *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*

$$\mathbf{C}_n = \mathbf{P}_0 \setminus \mathbf{A}_n. \tag{3.18}$$

Proof. (a) $\mathbf{P}_0 \setminus \mathbf{A}_n \supseteq \mathbf{C}_n$: $\mathbf{C}_n \overset{(3.11)}{=} \mathbf{P}_0 \cap \bigcap_{k=1}^{n} \mathbf{P}_k \overset{(3.10)}{=} \mathbf{P}_0 \cap \bigcap_{k=1}^{n} [\mathbf{P}_{k-1} \setminus \delta(\mathsf{T}_{k-1}, e_k) \cup \delta(\mathsf{T}_k, \bar{e}_k)]$. 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, $\mathsf{G} \in \mathbf{P}_0 \setminus \mathbf{A}_n$ and $\mathsf{G} \notin \mathbf{C}_n$, leads to a contradiction: $\mathsf{G} \notin \mathbf{C}_n \overset{(3.11)}{\Rightarrow} \exists_{\mathbf{P}_i} \mathsf{G} \notin \mathbf{P}_i, 0 \leq i \leq n$. However, by induction we show that $\forall_{\mathbf{P}_i} \mathsf{G} \in \mathbf{P}_i$: $\mathsf{G} \in \mathbf{P}_0$ is true. $\mathsf{G} \in \mathbf{P}_i \Rightarrow \mathsf{G} \in \mathbf{P}_{i+1}, 0 \leq i \leq n-1$: $\mathbf{P}_{i+1} \overset{(3.10)}{=} \mathbf{P}_i \setminus \delta(\mathsf{T}_i, e_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1})$; $\mathsf{G} \in \mathbf{P}_0 \setminus \mathbf{A}_n \Rightarrow \mathsf{G} \notin \mathbf{A}_n \Rightarrow \forall_{i=0..n-1} \mathsf{G} \notin \delta(\mathsf{T}_i, e_{i+1}) \Rightarrow \mathsf{G} \in \mathbf{P}_{i+1}$. □

Lemma 3.6. *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 $\mathbf{B}_i = \bigcup_{k=1}^{i} \delta(\mathsf{T}_i, \bar{e}_k)$. Then*

$$\mathbf{B}_n \cap \mathbf{C}_n = \emptyset. \tag{3.19}$$

Proof. Proof by induction. (i) True for $i = 1$: $\mathbf{B}_1 = \delta(\mathsf{T}_1, \bar{e}_1) \Rightarrow \mathbf{B}_1 \cap \mathbf{P}_0 = \emptyset \overset{(3.11)}{\Rightarrow} \mathbf{B}_1 \cap \mathbf{C}_n = \emptyset$.

(ii) Induction hypothesis:

$$\mathbf{B}_i \cap \mathbf{C}_n = \emptyset \Rightarrow \mathbf{B}_{i+1} \cap \mathbf{C}_n = \emptyset. \tag{3.20}$$

We show $\mathbf{B}_{i+1} \cap \mathbf{C}_n \subseteq \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{C}_n$: $\mathbf{B}_{i+1} \cap \mathbf{C}_n \overset{(3.17)}{=} [\mathbf{B}_i \setminus \delta(\mathsf{T}_i, e_{i+1}) \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1})] \cap \mathbf{C}_n \subseteq [\mathbf{B}_i \cup \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1})] \cap \mathbf{C}_n = [\mathbf{B}_i \cap \mathbf{C}_n] \cup [\delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{C}_n] \overset{(3.20)}{=} [\delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{C}_n]$. Then it follows with $\delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{P}_i = \emptyset \overset{(3.11)}{\Rightarrow} \delta(\mathsf{T}_{i+1}, \bar{e}_{i+1}) \cap \mathbf{C}_n = \emptyset$ that $\mathbf{B}_{i+1} \cap \mathbf{C}_n = \emptyset$. □

Theorem 3.1. *Let $L = (e_1, \ldots, e_n)$ be a sequence of edit operations that transforms T_0 into T_n, $\mathsf{T}_i = e_i(\mathsf{T}_{i-1})$, $1 \leq i \leq n$. The set of new pq-grams, Δ_n^+, can be computed as*

$$\Delta_n^+ = \bigcup_{k=1}^{n} \delta(\mathsf{T}_n, \bar{e}_k). \tag{3.21}$$

Proof. With Lemma 3.4, $\mathbf{P}_n$ can be expressed as

$$\mathbf{P}_n = \mathbf{P}_0 \setminus \mathbf{A}_n \cup \mathbf{B}_n, \tag{3.22}$$

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^+: $\mathbf{P}_n \overset{(3.22)}{=} \mathbf{P}_0 \setminus \mathbf{A}_n \cup \mathbf{B}_n \overset{(3.18)}{=} \mathbf{C}_n \cup \mathbf{B}_n$. As $\mathbf{B}_n$ and $\mathbf{C}_n$ are disjoint (Lemma 3.6), we can rewrite $\mathbf{P}_n = \mathbf{C}_n \cup \mathbf{B}_n$ as $\mathbf{B}_n = \mathbf{P}_n \setminus \mathbf{C}_n \overset{(3.12)}{=} \Delta_n^+$. □

3.6.4 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, $(\bar{e}_n, \ldots, \bar{e}_1)$), 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 $\mathbf{P}_j$, $\mathbf{P}_i = \mathcal{U}(\mathbf{P}_j, \bar{e}_j)$ (3.10). Thus, we can compute $\mathbf{P}_0$ from $\mathbf{P}_n$ by applying the profile update function recursively, $\mathbf{P}_0 = \mathcal{U}(\ldots\mathcal{U}(\mathcal{U}(\mathbf{P}_n, \bar{e}_n), \bar{e}_{n-1})\ldots, \bar{e}_1)$. 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 $\mathbf{P}_n$ (3.12). 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^+,

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

We will use the following Lemma 3.7 to rewrite the recursive updates in an un-nested form.

Lemma 3.7. *Let Δ_i^* be the result of iteratively applying the profile update function to Δ_n^+ i times, $1 \leq i \leq n$,*

$$\Delta_i^* = \mathcal{U}(\ldots\mathcal{U}(\mathcal{U}(\Delta_n^+, \bar{e}_n), \bar{e}_{n-1})\ldots, \bar{e}_{n-i+1}). \tag{3.23}$$

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

$$\Delta_i^* = \underbrace{\bigcup_{k=1}^{n-i} \delta(\mathsf{T}_{n-i}, \bar{e}_k)}_{\mathbf{A}_i^*} \cup \underbrace{\bigcup_{k=n-i+1}^{n} \delta(\mathsf{T}_{n-i}, e_k)}_{\mathbf{B}_i^*}. \tag{3.24}$$

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 (3.24) by induction:

(i) Δ_1^* computed with (3.23) and (3.24) matches: $\Delta_1^* \overset{(3.24)}{=} \bigcup_{k=1}^{n-1} \delta(\mathsf{T}_{n-1}, \bar{e}_k) \cup \delta(\mathsf{T}_{n-1}, e_n)$. $\Delta_1^* \overset{(3.23)}{=} \mathcal{U}(\Delta_n^+, \bar{e}_n) \overset{(3.21)}{=} \mathcal{U}(\bigcup_{k=1}^{n} \delta(\mathsf{T}_n, \bar{e}_k), \bar{e}_n) \overset{(3.9)}{=} \bigcup_{k=1}^{n} \delta(\mathsf{T}_n, \bar{e}_k) \setminus \delta(\mathsf{T}_n, \bar{e}_n) \cup \delta(\mathsf{T}_{n-1}, e_n) = \bigcup_{k=1}^{n-1} \delta(\mathsf{T}_n, \bar{e}_k) \setminus \delta(\mathsf{T}_n, \bar{e}_n) \cup \delta(\mathsf{T}_{n-1}, e_n) \overset{(3.3)(3.14)}{=} \bigcup_{k=1}^{n-1} \delta(\mathsf{T}_{n-1}, \bar{e}_k) \setminus \delta(\mathsf{T}_{n-1}, e_n) \cup \delta(\mathsf{T}_{n-1}, e_n) \overset{(3.4)}{=} \bigcup_{k=1}^{n-1} \delta(\mathsf{T}_{n-1}, \bar{e}_k) \cup \delta(\mathsf{T}_{n-1}, e_n)$.

(ii) Induction hypothesis:

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

$$\begin{aligned}\Delta^*_{i+1} &\overset{(3.23)}{=} \mathcal{U}(\Delta^*_i, \bar{e}_{n-i}) = \mathcal{U}(\mathbf{A}^*_i \cup \mathbf{B}^*_i, \bar{e}_{n-i}) \\ &\overset{(3.9)}{=} [\mathbf{A}^*_i \cup \mathbf{B}^*_i] \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) \cup \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \\ &\overset{(3.3)}{=} [\mathbf{A}^*_i \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i})] \cup [\mathbf{B}^*_i \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i})] \cup \\ &\quad \delta(\mathsf{T}_{n-i-1}, e_{n-i})\end{aligned} \tag{3.25}$$

$$\begin{aligned}\mathbf{A}^*_i \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) &\overset{(3.3)}{=} \textstyle\bigcup_{k=1}^{n-i-1} \delta(\mathsf{T}_{n-i}, \bar{e}_k) \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) \\ &\overset{(3.14)}{=} \textstyle\bigcup_{k=1}^{n-i-1} \delta(\mathsf{T}_{n-i-1}, \bar{e}_k) \setminus \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \\ &= \mathbf{A}^*_{i+1} \setminus \delta(\mathsf{T}_{n-i-1}, e_{n-i})\end{aligned} \tag{3.26}$$

$$\mathbf{B}^*_i \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) \overset{(3.14)}{=} \bigcup_{k=n-i+1}^{n} \delta(\mathsf{T}_{n-i-1}, e_k) \setminus \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \tag{3.27}$$

$$\begin{aligned}&\mathbf{B}^*_i \setminus \delta(\mathsf{T}_{n-i}, \bar{e}_{n-i}) \cup \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \\ &\quad \overset{(3.27)(3.4)}{=} \bigcup_{k=n-i+1}^{n} \delta(\mathsf{T}_{n-i-1}, e_k) \cup \delta(\mathsf{T}_{n-i-1}, e_{n-i}) \\ &\quad = \bigcup_{k=n-i}^{n} \delta(\mathsf{T}_{n-i-1}, e_k) = \mathbf{B}^*_{i+1}\end{aligned} \tag{3.28}$$

With (3.25), (3.26) and (3.28) we get $\mathbf{P}^*_{i+1} = \mathbf{A}^*_{i+1} \cup \mathbf{B}^*_{i+1}$. □

Theorem 3.2. *Let $L = (e_1, \ldots, e_n)$ be a sequence of edit operations that transforms T_0 into T_n, $\mathsf{T}_i = e_i(\mathsf{T}_{i-1})$, $1 \leq i \leq n$. The set of old pq-grams, Δ^-_n, can be computed as*

$$\Delta^-_n = \mathcal{U}(\ldots \mathcal{U}(\mathcal{U}(\Delta^+_n, \bar{e}_n), \bar{e}_{n-1}) \ldots, \bar{e}_1). \tag{3.29}$$

Proof. As $\Delta^-_n = \mathcal{U}(\ldots \mathcal{U}(\mathcal{U}(\Delta^+_n, \bar{e}_n), \bar{e}_{n-1}) \ldots, \bar{e}_1) \overset{(3.23)}{=} \Delta^*_n$, with (3.24) we can rewrite (3.29) in un-nested form as

$$\Delta^-_n = \bigcup_{k=1}^{n} \delta(\mathsf{T}_0, e_k). \tag{3.30}$$

For the proof of (3.30) consider the inverse scenario, i.e., T_n is transformed to T_0 by $(\bar{e}_n, \ldots \bar{e}_1)$. 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

$$\Delta'^+_n \overset{(3.12)}{=} \mathbf{P}'_n \setminus \mathbf{C}'_n \quad \text{or} \quad \Delta'^+_n \overset{(3.21)}{=} \bigcup_{k=1}^{n} \delta(\mathsf{T}_0, e_k).$$

$\mathbf{C}'_n = \mathbf{C}_n$ as both of them are the intersection of the same profiles. With $\mathbf{P}'_n = \mathbf{P}_0$ we get $\Delta'^{+}_n = \mathbf{P}_0 \setminus \mathbf{C}_n \overset{(3.12)}{=} \Delta^{-}_n$. □

3.7 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.

3.7.1 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 dummy nodes.

Definition 3.7. 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*:*

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

If a *is a leaf node, i.e.,* $f = 0$*, the* q*-matrix is defined as a* $1 \times q$*-matrix that contains only dummy 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 3.4. *We consider tree* T_0 *in Figure 3.2, assume* $p = q = 3$*, and compute all pq-grams with anchor node* n_1 *using the p- and q-matrices.*

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

3.7.2 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 $d \geq 0$, i.e., $\mathrm{desc}_d(\mathsf{n}) = \{\mathsf{x} \mid \mathsf{x}$ is n or a descendant of n with $\mathrm{dist}(\mathsf{n}, \mathsf{x}) \leq d\}$. For $d < 0$ we define $\mathrm{desc}_d(\mathsf{n}) = \emptyset$. 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 $P^{\mathsf{a}_i/\mathsf{m}}$ replaces a_i by m. The other nodes in $P(\mathsf{a})$ are shifted as shown in Figure 3.9, where a_i is a's ancestor at distance i.

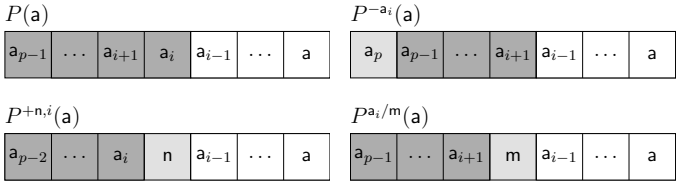


Figure 3.9: Operators on the *p*-Matrix.

The operations on *q*-matrices are illustrated in Figure 3.10. $Q(\mathsf{a})$ 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 dummy 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.

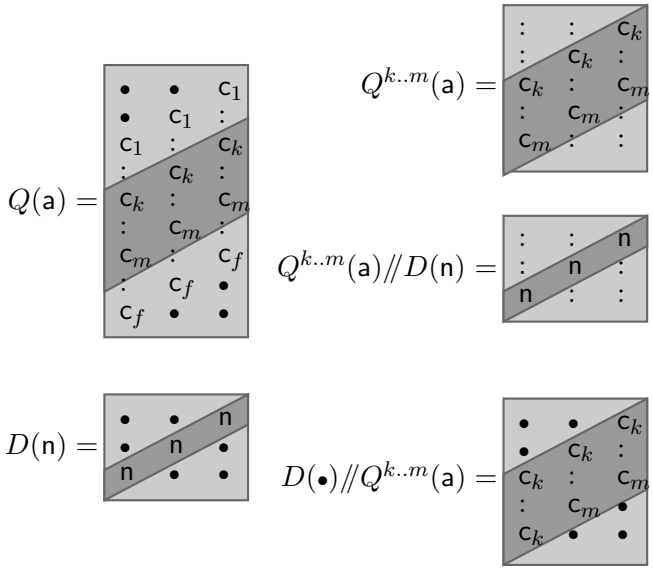


Figure 3.10: Operators on the q-Matrix.

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 dummy 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 3.2), 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 3.1 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.

3.7.3 Example

Example 3.5. *Consider the first two edit operations in Figure 3.1 that transform* T_0 *into* T_2*. The reverse edit operations are* $\bar{e}_1 = \text{DEL}(\mathsf{n}_7)$ *and* $\bar{e}_2 = \text{INS}((\mathsf{n}_3, \mathsf{b}), \mathsf{n}_1, 2, 3)$*. We determine the new pq-grams,* Δ_2^+*,* $p = q = 3$*, by*

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

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

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

$$\begin{aligned}
\delta(\mathsf{T}_j, \bar{e}) &= P(\mathsf{v}) \circ Q^{k..k}(\mathsf{v}) \cup P(\mathsf{x}) \circ Q(\mathsf{x}) && \forall \mathsf{x} \in \mathrm{desc}_{p-1}(\mathsf{n}) \\
\mathcal{U}(\delta(\mathsf{T}_j, \bar{e}), \bar{e}) &= P(\mathsf{v}) \circ [Q^{k..k}(\mathsf{v}) /\!\!/ Q(\mathsf{n})] \cup P^{-\mathsf{n}}(\mathsf{x}) \circ Q(\mathsf{x}) && \forall \mathsf{x} \in \mathrm{desc}_{p-1}(\mathsf{n}) \setminus \{\mathsf{n}\} \\
& && \mathsf{v} : \mathsf{n} \text{ is the } k\text{-th child of } \mathsf{v}
\end{aligned}$$

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

$$\begin{aligned}
\delta(\mathsf{T}_j, \bar{e}) &= P(\mathsf{v}) \circ Q^{k..k}(\mathsf{v}) \cup P(\mathsf{x}) \circ Q(\mathsf{x}) && \forall \mathsf{x} \in \mathrm{desc}_{p-1}(\mathsf{n}) \\
\mathcal{U}(\delta(\mathsf{T}_j, \bar{e}), \bar{e}) &= P(\mathsf{v}) \circ [Q^{k..k}(\mathsf{v}) /\!\!/ D(\mathsf{m})] \cup P^{\mathsf{n}/\mathsf{m}}(\mathsf{x}) \circ Q(\mathsf{x}) && \forall \mathsf{x} \in \mathrm{desc}_{p-1}(\mathsf{n}) \\
& && \mathsf{m} = (\mathrm{id}(\mathsf{n}), l') \\
& && \mathsf{v} : \mathsf{n} \text{ is the } k\text{-th child of } \mathsf{v}
\end{aligned}$$

Table 3.1: Computing the Delta Function and the Profile Update Function.

evaluating the delta functions in Table 3.1 for $\bar{e}_1$ and $\bar{e}_2$, i.e.,

$$\begin{aligned}
\Delta_2^+ &= \delta(\mathsf{T}_2, \bar{e}_1) \cup \delta(\mathsf{T}_2, \bar{e}_2) = \\
&\quad \{(\bullet, \mathsf{n}_1, \mathsf{n}_6, \bullet, \bullet, \mathsf{n}_7), (\bullet, \mathsf{n}_1, \mathsf{n}_6, \bullet, \mathsf{n}_7, \bullet), \\
&\quad\ (\bullet, \mathsf{n}_1, \mathsf{n}_6, \mathsf{n}_7, \bullet, \bullet), (\mathsf{n}_1, \mathsf{n}_6, \mathsf{n}_7, \bullet, \bullet, \bullet)\} \cup \\
&\quad \{(\bullet, \bullet, \mathsf{n}_1, \bullet, \mathsf{n}_2, \mathsf{n}_5), (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_2, \mathsf{n}_5, \mathsf{n}_6), (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_5, \mathsf{n}_6, \mathsf{n}_4), \\
&\quad\ (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_6, \mathsf{n}_4, \bullet), (\bullet, \mathsf{n}_1, \mathsf{n}_5, \bullet, \bullet, \bullet), (\bullet, \mathsf{n}_1, \mathsf{n}_6, \bullet, \bullet, \mathsf{n}_7), \\
&\quad\ (\bullet, \mathsf{n}_1, \mathsf{n}_6, \bullet, \mathsf{n}_7, \bullet), (\bullet, \mathsf{n}_1, \mathsf{n}_6, \mathsf{n}_7, \bullet, \bullet), (\mathsf{n}_1, \mathsf{n}_6, \mathsf{n}_7, \bullet, \bullet, \bullet)\} \\
&= \{(\bullet, \bullet, \mathsf{n}_1, \bullet, \mathsf{n}_2, \mathsf{n}_5), (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_2, \mathsf{n}_5, \mathsf{n}_6), (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_5, \mathsf{n}_6, \mathsf{n}_4), \\
&\quad\ (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_6, \mathsf{n}_4, \bullet), (\bullet, \mathsf{n}_1, \mathsf{n}_5, \bullet, \bullet, \bullet), (\bullet, \mathsf{n}_1, \mathsf{n}_6, \bullet, \bullet, \mathsf{n}_7), \\
&\quad\ (\bullet, \mathsf{n}_1, \mathsf{n}_6, \bullet, \mathsf{n}_7, \bullet), (\bullet, \mathsf{n}_1, \mathsf{n}_6, \mathsf{n}_7, \bullet, \bullet), (\mathsf{n}_1, \mathsf{n}_6, \mathsf{n}_7, \bullet, \bullet, \bullet)\}.
\end{aligned}$$

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 3.11 shows some of the modified q-matrices that are used in the evaluation of the update function for $\bar{e}_2 = \text{INS}((\mathsf{n}_3, \mathsf{b}), \mathsf{n}_1, 2, 3)$. The relevant p-parts in Δ_2^+ are transformed by inserting the new node n_3, e.g.,

$$\begin{aligned}
P(\mathsf{n}_1) = (\bullet, \bullet, \mathsf{n}_1) &\rightarrow P^{+\mathsf{n}_3,0}(\mathsf{n}_1) = (\bullet, \mathsf{n}_1, \mathsf{n}_3) \\
P(\mathsf{n}_5) = (\bullet, \mathsf{n}_1, \mathsf{n}_5) &\rightarrow P^{+\mathsf{n}_3,1}(\mathsf{n}_5) = (\mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_5)
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{U}(\Delta_2^+, \bar{e}_2) = \{&(\bullet, \bullet, \mathsf{n}_1, \bullet, \mathsf{n}_2, \mathsf{n}_3), (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_2, \mathsf{n}_3, \mathsf{n}_4), (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_4, \bullet), \\
&(\bullet, \mathsf{n}_1, \mathsf{n}_3, \bullet, \bullet, \mathsf{n}_5), (\bullet, \mathsf{n}_1, \mathsf{n}_3, \bullet, \mathsf{n}_5, \mathsf{n}_6), (\bullet, \mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_5, \mathsf{n}_6, \bullet), \\
&(\bullet, \mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_6, \bullet, \bullet), (\mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_5, \bullet, \bullet, \bullet), (\mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_6, \bullet, \bullet, \mathsf{n}_7), \\
&(\mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_6, \bullet, \mathsf{n}_7, \bullet), (\mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_6, \mathsf{n}_7, \bullet, \bullet), (\mathsf{n}_3, \mathsf{n}_6, \mathsf{n}_7, \bullet, \bullet, \bullet)\}.
\end{aligned}$$

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

$$\begin{aligned}
\Delta_2^- = \{&(\bullet, \bullet, \mathsf{n}_1, \bullet, \mathsf{n}_2, \mathsf{n}_3), (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_2, \mathsf{n}_3, \mathsf{n}_4), (\bullet, \bullet, \mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_4, \bullet), \\
&(\bullet, \mathsf{n}_1, \mathsf{n}_3, \bullet, \bullet, \mathsf{n}_5), (\bullet, \mathsf{n}_1, \mathsf{n}_3, \bullet, \mathsf{n}_5, \mathsf{n}_6), (\bullet, \mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_5, \mathsf{n}_6, \bullet), \\
&(\bullet, \mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_6, \bullet, \bullet), (\mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_5, \bullet, \bullet, \bullet), (\mathsf{n}_1, \mathsf{n}_3, \mathsf{n}_6, \bullet, \bullet, \bullet)\}.
\end{aligned}$$

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

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

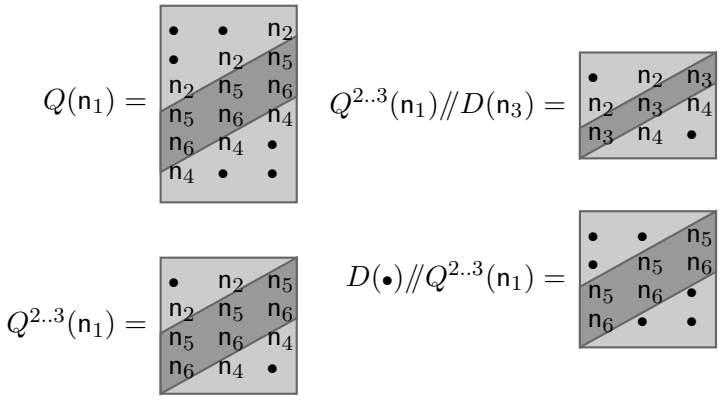


Figure 3.11: q-Matrices for Node Insertion (Example).

3.8 Implementation

3.8.1 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 3.2). 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 3.3) 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 3.3. 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

$$\lambda(\mathtt{P}, \mathtt{Q}) = \pi_{ppart \circ qpart}[\mathtt{P} \bowtie \mathtt{Q}]. \tag{3.31}$$

Subsequently, given pairs of tables we use the notation $(A, B) \leftarrow (A', B') \cup (A'', B'')$ 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.

P

anchId	*sibPos*	*parId*	*ppart*
$\mathtt{n}_1$	-	-	001
$\mathtt{n}_5$	2	$\mathtt{n}_1$	018
$\mathtt{n}_6$	3	$\mathtt{n}_1$	014
$\mathtt{n}_7$	1	$\mathtt{n}_6$	147

Q

anchId	*row*	*qpart*
$\mathtt{n}_1$	2	028
$\mathtt{n}_1$	3	284
$\mathtt{n}_1$	4	842
$\mathtt{n}_1$	5	420
$\mathtt{n}_5$	1	000
$\mathtt{n}_6$	1	007
$\mathtt{n}_6$	2	070
$\mathtt{n}_6$	3	700
$\mathtt{n}_7$	1	000

Figure 3.12: Δ_2^+ for T_2, Stored in the Table Pair $(\mathtt{P}, \mathtt{Q})$.

Example 3.6. *Figure 3.12 shows* $\Delta_2^+ = \bigcup_{i=1}^{2} \delta(\mathsf{T}_2, \bar{e}_i)$ *for our example tree in Figure 3.2. The first rows of* P *and* Q *show the hashed p-part and q-part of the label-tuple* $(*, *, \mathtt{a}, *, \mathtt{c}, \mathtt{e})$.

3.8.2 Index Update

For the index maintenance we use the old index $\mathbf{I}_0$, 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 $\mathbf{I}_0$ with $\lambda(\Delta_n^+)$ and $\lambda(\Delta_n^-)$ (see Algorithm 3.1). 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 3.1: updateIndex$(\mathtt{I}_0, \mathsf{T}_n, L)$

```
1 (P, Q) ← (∅, ∅);
2 foreach ē_i ∈ L do (P, Q) ← (P, Q) ∪ δ(T_n, ē_i);
3 I⁺ ← λ(P, Q);
4 for i ← n downto 1 do U(P, Q, ē_i);
5 I⁻ ← λ(P, Q);
6 I_n ← I_0 \ I⁻ ∪ I⁺;
7 return 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.

3.8.3 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 3.2). The relevant matrices for each edit operation are shown in Table 3.1. 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 $\mathtt{P}$ and $\mathtt{Q}$, as shown in Section 3.8.1.

Algorithm 3.2: $\delta(\mathsf{T},\bar{e})$

```
1  if (ē = REN(n, l')) ∨ (ē = DEL(n)) then
2      v ← parent of n;
3      k ← sibling position of n (n is the k-th child of v);
4      (P, Q) ← (P_T(v), Q_T^{k..k}(v));
5      foreach x ∈ desc_{p−1}(n) do
6          (P, Q) ← (P, Q) ∪ (P_T(x), Q_T(x))
7  else if ē=INS(n,v,k,m) then
8      (P, Q) ← (P_T(v), Q_T^{k..m}(v));
9      foreach child c ∈ {c_k, ..., c_m} of v do
10         foreach x ∈ desc_{p−2}(c) do
11             (P, Q) ← (P, Q) ∪ (P_T(x), Q_T(x))
12 return (P, Q);
```

3.8.4 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 $\mathtt{P}$ and $\mathtt{Q}$. 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 3.1, 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:

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

Algorithm 3.3: $\mathcal{U}(\mathtt{P}, \mathtt{Q}, \bar{e})$

```
1  switch ē do
2  case REN(n,l')
3      t ← σ_{anchId=n}(P);  v ← t[parId];  k ← t[sibPos];
4      Q ← Q \ Q^{k..k}(v) ∪ [Q^{k..k}(v)//D_v((id(n), l'))];
5      s ← subStr(t[ppart], 1, p − 1) ∘ l';
6      (P_old, P_new) ← changePParts(P, n, s, p − 1);
7      P ← P \ P_old ∪ P_new;
8  case DEL(n)
9      t ← σ_{anchId=n}(P);  v ← t[parId];  k ← t[sibPos];
10     Q ← Q \ [Q^{k..k}(v) ∪ Q(n)] ∪ [Q^{k..k}(v)//Q(n)];
11     s ← λ(•) ∘ subStr(t[ppart], 1, p − 1);
12     (P_old, P_new) ← changePParts(P, n, s, p − 1);
13     P ← P \ P_old ∪ σ_{anchId≠n}(P_new);
14 case INS(n,v,k,m)
15     Q ← Q \ Q^{k..m}(v) ∪ [Q^{k..m}(v)//D_v(n)] ∪ [D_n(•)//Q^{k..m}(v)];
16     s ← subStr(π_{ppart} σ_{anchId=v}(P), 2, p) ∘ λ(n);
17     P_old ← ∅;  P_new ← ∅;
18     foreach c ∈ π_{anchId} σ_{parId=v, k≤sibPos≤m}(P) do
19         s' ← subStr(s, 2, p) ∘ λ(c);
20         (P_old, P_new) ← (P_old, P_new) ∪ changePParts(P, c, s', p − 2);
21     P ← P \ P_old ∪ P_new ∪ {(n, k, v, s)};
```

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

In the second step we modify $\delta(\mathtt{T}, \bar{e})$ to get $\mathcal{U}(\delta(\mathtt{T}, \bar{e}), \bar{e})$. 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. $\mathtt{D}_{\mathtt{a}}(\mathtt{n})$ 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 $\mathtt{changePParts}(\mathtt{P}, \mathtt{n}, s, d)$ (see Algorithm 3.4). It implements the operators on $P(\mathtt{a})$ ($P^{+\mathtt{n},i}$, $P^{-\mathtt{a}_i}$, $P^{\mathtt{a}_i/\mathtt{m}}$) 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 $P(\mathtt{a})$ (shaded in Figure 3.9). 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 $\mathtt{n}$ or a descendant of $\mathtt{n}$ within distance d. $\mathtt{P}_{\mathtt{new}}$ is the same set of tuples with the updated values for *ppart*.

Algorithm 3.4: $\mathtt{changePParts}(\mathtt{P}, \mathtt{n}, s, d)$

```
1 P_old ← ∅; P_new ← ∅;
2 Z ← σ_{anchId=n}(P);
3 for i ← 0 to d do
4     P_old ← P_old ∪ Z;
5     P_new ← P_new ∪ π[anchId, sibPos, parId,
                        subStr(s, i + 1, |s|) ∘
                        subStr(ppart, p − i + 1, p) → ppart](Z);
6     if i < d then Z ← P ⋈ π_{anchId→parId} Z;
7 return (P_old, P_new);
```

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 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.

3.9 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.

3.9.1 Lookup Efficiency

If we look up a tree $\mathtt{T}$ in a forest $\mathcal{F}$, we have to compute the *pq*-gram distance between $\mathtt{T}$ and each of the trees in $\mathcal{F}$. 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

[1] `http://monetdb.cwi.nl/xml/`

[2] `http://www.informatik.uni-trier.de/~ley/db/`

similar size. We measure the wall clock time for the approximate lookup of an XML document.

Figure 3.13 (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.

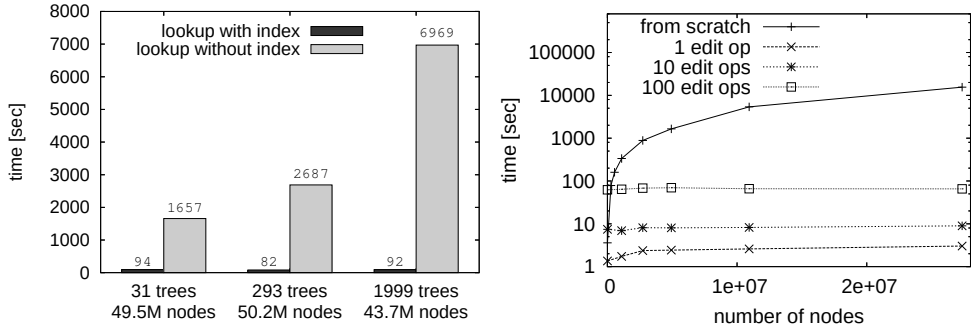


Figure 3.13: Lookup and Update Time.

3.9.2 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 3.13 (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.

3.9.3 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 3.14 (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.

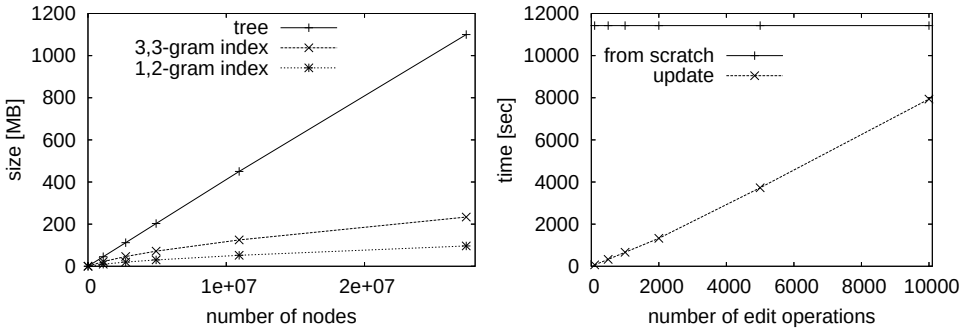


Figure 3.14: Size and Update Time of Index.

3.9.4 Experiments with Real World Data

We compute the index and perform updates on the DBLP dataset (211MB file size, 11M nodes). From Figure 3.14 (right) we see that the update time is linear in the number of edit operations. Table 3.2 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 3.2: Breakdown of the Index Update Time.

3.10 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.

Chapter 4

pq-Grams for Unordered Trees

In data integration applications, a join *matches elements that are common to two data sources. Often however elements are represented slightly different in each source, so an* approximate *join must be used. For XML data, most approximate join strategies are based on some* ordered *tree-matching technique. But in data-centric XML applications, the order is irrelevant: two elements should match even if their subelement order varies.*

We give a solution for the approximate join of unordered trees. Our solution is based on windowed *pq*-grams. *A windowed pq-gram is a small tree-shaped pattern that consists of a stem and a base. We develop a technique to systematically generate windowed pq-grams from sorted trees. Sorting trees is not possible for common ordered tree distances such as the edit distance. Windowed pq-grams satisfy the following core properties: all base-nodes have equal frequency; the Jaccard distance between two sibling sets is preserved; and node moves to other parents are detected. The pq-gram distance between two sorted trees approximates the unordered tree edit distance between these trees. We provide an algorithm to compute the windowed pq-grams in linear time. Our experiments with synthetic and real world data confirm the analytic results and suggest that our technique is both useful and scalable.*

4.1 Introduction

The amount of data that is stored and exchanged in XML is increasing. In order to integrate XML data from different sources into a single data collection, data items that correspond to the same real world object must be matched. Exact matches often fail due to inconsistent representations and missing global keys, so approximate matching techniques must be applied. For instance, when companies merge, their customer data will need to be integrated, but the companies may have different ways to represent that customer data. As another

example, an internet shop may want to enrich its product description with data provided by third parties, which each have slightly different descriptions for the same product.

One way to approximately match a pair of XML documents is to model the XML documents as *ordered*, labeled trees and to compute the *minimal edit distance* between the trees [12, 26, 37]. The edit distance between two such trees is the number of node insertions, deletions, and renamings that transform one tree into the other. While order is important in document-centric scenarios (e.g., paragraph tags in XHTML), most applications of data-centric XML must ignore the sibling order. Two documents should be considered the same even if they differ in the order of their siblings. Data-centric XML items are usually modeled as *unordered*, labeled trees. While the minimal tree edit distance between ordered trees can be computed in polynomial time, the problem has been shown to be NP-complete for unordered trees [57].

This work develops an efficient approximate join between data-centric XML. Our solution is based on *windowed pq-grams.* A windowed *pq*-gram is a small subtree that consists of a stem and a base. Intuitively, two unorderd trees are similar if they have many *pq*-grams in common. We develop a technique to systematically generate the set of windowed *pq*-grams from a sorted tree. Windowed *pq*-grams satisfy the following core properties: all base-nodes have equal frequency; the Jaccard distance between two sibling sets is preserved; and node moves to other parents are detected. Due to these properties key features of *pq*-grams for ordered trees [3] carry over to unordered trees. Specifically, the *pq*-gram distance computed on the windowed *pq*-grams of two sorted trees approximates the unordered tree edit distance between these trees.

Note that sorting trees is not possible for common ordered tree distances such as the tree edit distance. Sorting a tree permutes the subtrees rooted in the sorted nodes. While the sorted trees of two identical unordered trees are identical for the *pq*-gram distance, they may be very dissimilar for the ordered edit distance (cf. Section 4.4.3).

We provide an algorithm to compute windowed *pq*-grams in linear time and approximately join unordered trees using windowed *pq*-grams. Most joins based on distance measures, such as the edit distance, must evaluate the distance between every pair of input trees. There is no effective way to sort trees or partition them input buckets with a hash function. A nested-loop join must be applied. Our algorithm reduces the approximate join to an equality join on strings (windowed *pq*-grams are serialized and represented as strings) that takes advantage of well known join optimization techniques.

The rest of the chapter is organized as follows: We discuss related work in Section 4.2. We motivate the approximate join of data-centric XML in

Section 4.3 and introduce windowed *pq*-grams in Section 4.4. Section 4.5 discusses core properties of windowed *pq*-grams, and we tune the *pq*-grams to optimize these properties Section 4.6. Section 4.7 provides algorithms, which are experimentally evaluated in Section 4.8. In Section 4.9 we draw conclusions and point to future work.

4.2 Related Work

Most papers that compare similar XML documents represent the XML data as trees. Labels or label-value pairs are assigned to the tree nodes. Tree matching techniques are applied to compute the similarity between trees. A well known distance function for trees is the tree edit distance, which is defined as the minimum cost sequence of edit operations (node insertion, node deletion, and renaming) that transforms one tree into another [47]. The best known tree edit distance algorithms [11, 16, 35, 56] for ordered trees have at least $O(n^3)$ runtime for trees with n nodes. The problem is NP-complete for unordered trees [57].

Guha et al. [26] present an approximate XML join based on the tree edit distance. They give upper and lower bounds for the tree edit distance that can be computed in $O(n^2)$ time and use reference sets to take advantage of the fact that the tree edit distance is a metric, thus reducing the actual number of distances to compute in a join. Guha et al. [26] do not address joins of *unordered* XML.

Garofalakis and Kumar [23] correlate streams of XML data through approximate matching in small space. They present an efficient approximation of the tree edit distance, but their approximation assumes ordered trees.

pq-Grams where introduced by Augsten et al. [3, 4] as an effective and efficient approximation of the tree edit distance for ordered trees. This work extends *pq*-grams for unordered trees and develops an efficient *pq*-gram-based approximate join technique.

In change detection scenarios two versions of the same document are given and the difference is computed. Most works assume ordered trees [10, 12, 37]. Cobéna et al. [12] take advantage of existing element IDs, which can not be assumed for joins of data from different sources. Chawathe et al. [9] present a heuristic solution for unordered trees that runs in $O(n^3)$ time and for many cases in $O(n^2)$. The X-Diff algorithm by Wang et al. [51] allows leaf and subtree insertion and deletion and node renaming. To achieve $O(n^2 \times f_{max} \log(f_{max}))$ runtime, where f_{max} is the maximum fanout of the nodes, they match only nodes with the same path to the root node. Our *pq*-gram distance has $O(n \log n)$ runtime complexity. The distance measures presented

above are evaluated between pairs of documents. Used as a join predicate there is no obvious way to avoid an expensive nested-loop join. We transform the distance-based join to an equality join on *pq*-grams and can apply well known join optimization techniques.

Weis and Naumann [52] propose an XML similarity measure for a duplicate detection framework. In the worst case, all pairs of elements must be compared. Puhlmann et al. [43] improve the efficiency by applying the Sorted Neighborhood method to nested objects. Both approaches assume a known, common schema of the matched documents and require a configuration step. No join algorithm using the proposed similarity measure is presented.

A core operation in XML query processing is to find all occurrences of a twig pattern [5, 31]. The goal of our work is not to find occurrences of a pattern to answer queries. We split the tree into subtrees in order to calculate the distance between trees. Several papers deal with the related, but different problem of detecting the structural similarity in XML documents [14, 20, 41]. Two documents are considered structurally similar if they are valid for a similar DTD. The (text) values of the elements within the documents are ignored.

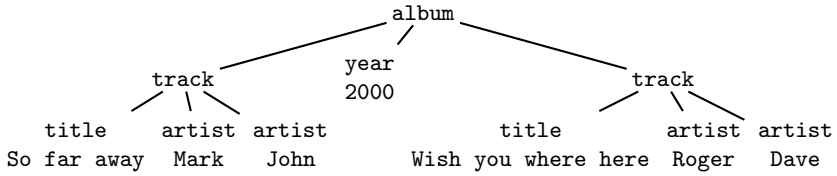
4.3 Motivation

In our application scenario we consider building an online database about music CDs that integrates data from two sources: a song lyric store and CD warehouse.[1] The integrated database will store the title, artist and songs on an album, information about individual songs such as the lyrics, guitar tabs, and information about the artists.

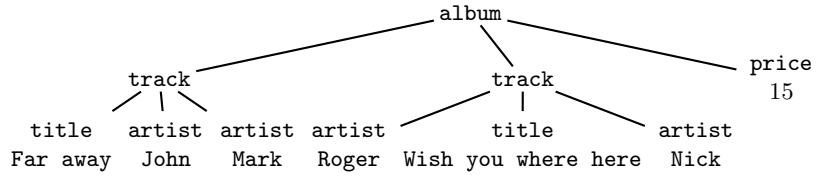
Example 4.1. *Figure 4.1 shows tree representations of two different XML documents. Intuitively, both represent data about the same song album. Yet exact ordered tree matching would not consider the items as the same for a number of reasons. The song lyric store has an element* `year` *that is absent from the CD Warehouse. The CD Warehouse has a price for the album. For one track the title is slightly different, for the other track the databases list different artists. Also the document order of elements differs, i.e., the two documents have different sibling orders.*

One way to match items from the two sources is to *join* the documents. The join attribute is (the part of) the XML document that represents the

[1] We do not assume that the sources use a common schema, but we assume a common vocabulary to describe the data; the problem of integrating data vocabularies or ontologies is separate from matching the data. Terms in one source can be converted to the vocabulary of the second source prior to matching. We focus on the data matching problem.



(a) Song Lyric Store Data



(b) CD Warehouse Data

Figure 4.1: Two XML Trees Representing the Same Album.

album. Two albums match if they are "similar." The join condition can not be equality, as the data items representing the same album in the different databases may not match exactly.

The following XQuery expression returns all `album` pairs that are within distance `$tau`. The distance function, `dist`, is a user-defined function that returns the distance between a pair of XML documents.

```
for $a in doc("lyricstore.xml")//album,
    $b in doc("warehouse.xml")//album
where dist($a,$b) <= $tau
return <match>{$a}{$b}</match>
```

In the XQuery expression, `$a` and `$b` are bound to elements of the sets `doc("lyricstore.xml")//album` and `doc("warehouse.xml")//album`, respectively. Each `album` element is a (small) XML document itself. We define the approximate XML join between two sets of XML documents as follows [26].

Definition 4.1 (Approximate XML Join)**.** *Given two sets of XML documents, $\mathcal{F}_1$ and $\mathcal{F}_2$, a distance measure,* $\text{dist}(\mathsf{T}_i, \mathsf{T}_j)$*, between document $\mathsf{T}_i \in \mathcal{F}_1$ and document $\mathsf{T}_j \in \mathcal{F}_2$, and a threshold τ. The approximate XML join computes all pairs $(\mathsf{T}_i, \mathsf{T}_j) \in \mathcal{F}_1 \times \mathcal{F}_2$, such that* $\text{dist}(\mathsf{T}_i, \mathsf{T}_j) \leq \tau$*.*

Our goal is to find a distance function for *unordered* trees that is effective for data-centric XML and can be computed efficiently. We use this function as the basis of a scalable approximate join.

4.4 Windowed *pq*-Grams

In this section we introduce windowed *pq*-grams. We define properties that we require for our solution.

In order to make *pq*-grams applicable for data-centric XML, we represent an XML document as an unordered, rooted, labeled tree. Each node in the tree is a triple (i, l, v), where i is the node index, l is the node label, and v is the node's value (text content). A node in the tree represents an XML element (or attribute) and is labeled with the name of the element (or attribute). The node index is any number that identifies the node in the document, such as its ordinal position in a pre-order traversal. The value of a node represents the text content of the corresponding element (or the value of the corresponding attribute). The node value is an empty string if the corresponding element contains only sub-elements and no content. An edge connects an element node with each of its subelements (or attributes). The function $\lambda(\mathtt{n})$ of a node $\mathtt{n} = (i, l, v)$ maps the node to the (label,value)-pair (l, v). While nodes are unique within a tree, the (label,value)-pairs are not.

4.4.1 Requirements for Windowed *pq*-Grams

pq-Grams were introduced by Augsten et al. [3] as an approximation of the edit distance for *ordered*, labeled trees. Intuitively, a *pq*-gram is a small subtree of a specific shape composed of two parts: a *stem* that consists of an anchor node with $p-1$ ancestors and a *base* that consists of q consecutive children of the anchor node. For example, consider the ordered tree T_0 in Fig. 4.2. The stem $(\mathtt{a}, \mathtt{c})$ with anchor node $\mathtt{c}$ and the base $(\mathtt{k}, \mathtt{j})$ form a *pq*-gram with $p=q=2$.

In *unordered* tree matching, the order of siblings is irrelevant. Graphically we represent an unordered tree as a set consisting of a node and the subtrees rooted in the node's children. The unordered trees T_1 and T_2 in Fig. 4.2 differ only in the node $\mathtt{g}$ that is moved between T_1 and T_2.

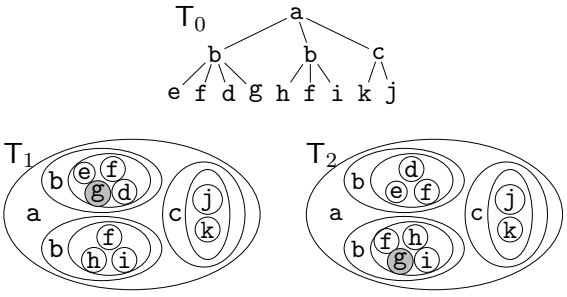


Figure 4.2: Ordered and Unordered Trees.

Stems are node chains of length p. They are invariant to order, and the strategy for choosing stems in ordered trees carries over to unordered trees. The *bases* in ordered trees are formed by consecutive siblings. This strategy is not applicable to unordered trees, since no sibling order is defined. A different strategy is required.

Let *sibling set* denote the set of all children of a tree node. A strategy to choose all possible sibling subsets of size q weights nodes differently. For f siblings, $\binom{f}{q}$ bases are formed. pq-Grams produced from large sibling sets disproportionally contribute to the total number of pq-grams. Changes covered by these pq-grams are amplified, other changes are disregarded.

Bases that consist of a single node ignore the sibling order However, pq-grams with such bases fail to detect sibling moves to an other parent if the ancestors in the old and the new position have identical labels. For example, they cannot distinguish between the trees T_1 and T_2 in Fig. 4.2. The ancestors of the moved node `g` have identical labels, resulting in identical stems, (`a`, `b`). Ancestors with identical labels are frequent in data-centric XML (e.g., all `title` elements have the ancestors `track` and `album` in the XML of Fig. 4.1).

Larger bases encode sibling information and can detect sibling moves, as nodes with homonymous ancestors may have siblings with different labels. In our example, `g` has a sibling `i` in T_2 but not in T_1. A base (`g`, `i`) exists only in T_2 and distinguishes it from T_1.

A sibling order may be given implicitly, for example, by the XML document order. This order is random for data centric XML. Bases formed over randomly ordered sibling sets may be very different even for identical sibling sets.

In our approach we sort the trees and use a window to control the computation of the bases. We seek to build bases with the following properties:

P1: Equal Base-Node Frequency. Each non-root node of the tree appears in the same number of bases, independent of the number of siblings.

P2: Preservation of the Sibling Distance. The overlap of the bases build from two different sibling sets is proportional to the overlap of the sibling sets. In Fig. 4.2, there is a 50% overlap between the sibling sets containing node `g` in T_1 and T_2, hence 50% of the bases should match, too.

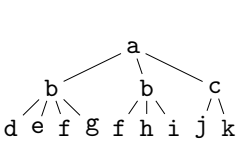
P3: Detection of Node Moves to Other Parents. Node `g` is moved to an other parent with the same label (`b`). All pq-grams with anchor node `b` have the same stem. In order to distinguish T_1 and T_2, the bases must differ.

4.4.2 Solution

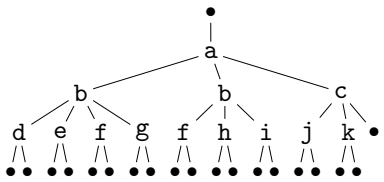
We introduce windowed *pq*-grams for unordered trees with the required base properties. We proceed in 3 steps:

1. sort the unordered tree,
2. extend the sorted tree, and
3. compute the *pq*-grams on the extended tree.

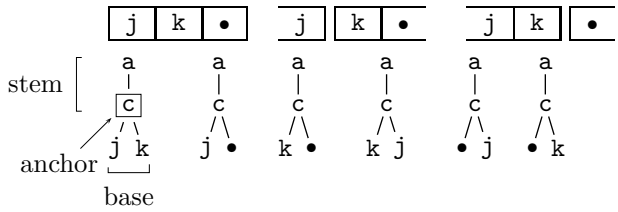
In the first step we sort the trees by imposing a horizontal order among siblings. The siblings are sorted by node label and value. Due to nodes with identical label-value pairs an unordered tree can possibly be sorted in different ways. All sorted trees of the same unordered tree result in identical *pq*-grams and are equivalent for our purpose. Figure 4.3(a) shows T_1^{sort}, the sorted example tree T_1.



(a) Sorted Tree T_1^{sort}



(b) Extended Tree T_1^{ext} for $w=3$ and $p=q=2$.



(c) Window Use and 2, 2-Grams with Anchor Node c.

Figure 4.3: Sorted Tree, Extended Tree, and Windowed *pq*-Grams.

Definition 4.2 (Sorted Tree)**.** *A tree* T *is* sorted *if its siblings are ordered and for each sibling pair,* $\mathsf{n} = (i, l, v)$ *and* $\mathsf{n}' = (i', l', v')$*, the order satisfies*

$$l < l' \vee (l = l' \wedge v < v') \Rightarrow \mathsf{n} < \mathsf{n}'.$$

The next step extends the sorted tree with *dummy nodes* (•). Dummy nodes have a special (label,value)-pair, which is the same for all such nodes, i.e., $\lambda(\bullet_i) = \lambda(\bullet_j)$ for all i, j. We also introduce the concept of a window which

is shifted over siblings for a systematic generation of *pq*-grams. Figure 4.3(b) shows the extended tree T_1^{ext} for $w = 3$ and $p = q = 2$.

Definition 4.3 (Extended Tree)**.** *Let T^{sort} be a sorted tree, $p > 0$ and $q > 0$ be the parameters determining the shape of the pq-grams, $w \geq q$ be the window size, and f denote the fanout of a node. The* extended tree, T^{ext}*, is defined as T^{sort} extended with dummy nodes as follows:*

- root: $p-1$ *ancestors are prepended to the root node;*
- leaves: q *children are added to each leaf node;*
- siblings: $w-f$ *siblings are appended to each sibling array* $(\mathsf{c}_1, \ldots, \mathsf{c}_f)$ *of size* $0<f<w$*, yielding* $(\mathsf{c}_1, \ldots, \mathsf{c}_f, \bullet_1, \ldots, \bullet_{w-f})$.

Finally, we give a definition of windowed *pq*-grams based on the extended tree.

Definition 4.4 (Windowed *pq*-Grams)**.** *Let T be an unordered tree with extended tree T^{ext}, n be a node of T, c_i be the i^{th} child of n in T^{ext} $(1 \leq i \leq f)$, and $W_i = (\mathsf{c}_i, \mathsf{c}_{i+1}, \ldots, \mathsf{c}_{(i+w-1) \bmod f})$ be a node sequence visible through a window of length $w \geq q$ that is wrapped around the right border.*

A windowed pq-gram $(p>0,\ q>0)$ of T with anchor node n is defined as an ordered subtree of T^{ext} that is composed of the stem $(\mathsf{a}_{p-1}, \ldots, \mathsf{a}_1, \mathsf{n})$*, where a_k is n's ancestor at distance k, and a* base $(\mathsf{c}_i, \mathsf{b}_2, \ldots, \mathsf{b}_q)$*, where c_i is the first node of a window W_i, $\{\mathsf{b}_2, \ldots, \mathsf{b}_q\} \subseteq \{\mathsf{c}_{i+1}, \ldots, \mathsf{c}_{(i+w-1) \bmod f}\}$, and the node order of W_i is preserved. If n is a leaf in T, the base is formed by q dummy nodes. Each base that satisfies these constraints produces a windowed pq-gram with anchor node n.*

*The set of all windowed pq-grams of a tree T (i.e. all pq-grams of all nodes of T) is called its pq-*gram profile*.*

The bases are systematically computed by producing for each window W_i only the bases that contain the first node. For each window position, $\binom{w-1}{q-1}$ bases are produced.

Theorem 4.1 (Profile Size)**.** *Let T be a tree with n nodes, then the size of its pq-gram profile, $P(\mathsf{T})$, is linear in the tree size, $|P(\mathsf{T})| \leq nq\binom{w}{q}$. If T has l leaves, and all other nodes have fanout $f \geq w$, then $|P(\mathsf{T})| = (n-1)\binom{w-1}{q-1} + l$.*

We use a linear encoding and represent a *pq*-gram as a tuple $\mathsf{G} = (\mathsf{a}_{p-1}, \ldots, \mathsf{a}_1, \mathsf{n}, \mathsf{b}_1, \ldots, \mathsf{b}_q)$. With $\lambda(\mathsf{G}) = (\lambda(\mathsf{a}_{p-1}), \ldots, \lambda(\mathsf{n}), \ldots, \lambda(\mathsf{b}_q))$ we denote a *pq*-gram's node labels and values, called its *label-tuple*. Subsequently

we omit node values and represent label-tuples as strings, e.g., the label-tuple of the first 2, 2-gram in Fig. 4.3(c) is represented as `acjk`. While a *pq*-gram is unique within a tree, different *pq*-grams may yield identical label-tuples.

Example 4.2. *Figure 4.3(c) shows all windowed pq-grams for $p = q = 2$ that can be formed in T_1^{ext} for the anchor node* `c`*. Initially, the window covers the nodes* $\boxed{j, k, \bullet}$*, which according to the above procedure yields two bases of size* 2 *and produces the first two pq-grams* `acjk` *and* `acj•`*. Next, the window is moved right and covers* $\boxed{k, \bullet, j}$*. Notice that the window is wrapped around. Two other pq-grams are produced. The final position of the window covers* $\boxed{\bullet, j, k}$*.*

Definition 4.5 (*pq*-Gram Index)**.** *Let T be a tree with pq-gram profile $\mathbf{P}(\mathsf{T})$, $p > 0$, $q > 0$. The pq-gram index, $\mathbf{I}$, of tree T is the bag of all label-tuples of T, i.e.,*

$$\mathbf{I}(\mathsf{T}) = \biguplus_{\mathsf{G} \in \mathbf{P}(\mathsf{T})} \lambda(\mathsf{G}).$$

The *pq*-gram distance is computed from 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

$$\mathrm{dist}(\mathsf{T}, \mathsf{T}') = 1 - 2\frac{|\mathbf{I}(\mathsf{T}) \cap \mathbf{I}(\mathsf{T}')|}{|\mathbf{I}(\mathsf{T}) \uplus \mathbf{I}(\mathsf{T}')|}.$$

The *pq*-gram distance is 1 if two trees share no *pq*-grams, and 0 if they have the same *pq*-gram index, which does not necessarily imply that the trees are equal.

4.4.3 Local Effect of Subtree Permutations

Sorting siblings moves the subtrees rooted in the siblings. Due to identical node labels, subtrees may be permuted between the sorted trees of two identical unordered trees. Further, if the root node of a subtree is renamed, the subtree is moved to an other position in the sorted tree. The number of *pq*-grams that changes after moving a subtree does not depend on the subtree size. This core feature qualifies *pq*-grams for our approach and rules out other distance measures such as the tree edit distance. We can not sort the siblings and apply the edit distance for ordered trees. The edit distance with node insertion, deletion and renaming moves the subtrees back node by node, thus increasing the distance. An additional subtree move operation makes the edit distance computation NP-hard [23].

Consider the two trees T_1 and T_2 in Figure 4.4. The unordered edit distance between the trees is zero as they differ only in the sibling order. Also the *pq*-gram distance is zero. The ordered edit distance is about the tree size as the subtrees t_1 and t_2 must be moved back node by node. Subtree moves have limited effect on the *pq*-gram distance. When all children of a node are permuted, only the *pq*-grams that have the permuted nodes in the bases will change.

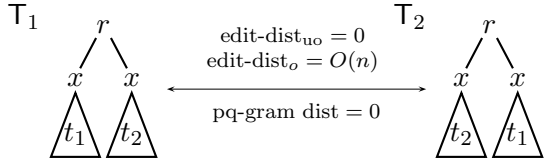


Figure 4.4: *pq*-Grams and Subtree Permutation.

Theorem 4.2 (Local Effect of Subtree Moves)**.** *Given a sorted tree* T *(index* $\mathbf{I}$*) that is transformed to a tree* T' *(index* $\mathbf{I}'$*) by permuting the order of the* f *children of a node* n*, then the permutation affects only* $O(f)$ *pq-grams:*

$$|\mathbf{I} \setminus \mathbf{I}'| \leq O(f).$$

4.5 Properties of Windowed *pq*-Gram Bases

We discuss the base properties of windowed *pq*-grams. We denote sibling sets with S, bags of sibling labels with L, and bases formed over L with B.

P1: Equal Base-Node Frequency. Tree extension with dummy nodes, windows, and window wrapping guarantee that each node of a tree is in the same number of bases, thus giving each node the same weight. Dummy nodes avoid that a node appears twice in the same window when the window is wrapped. Due to the window wrapping each node appears in all w positions of a window exactly once, independent of the number of its siblings. Only bases within windows are formed, thus each node is in the same number of bases.

P2: Preservation of the Sibling Distance. We analyze the *pq*-grams of two anchor nodes that have the same stem and differ only in the bases. The bases represent the sibling sets formed by the children of the anchor nodes. The distance between the bases should approximate the distance between the sibling sets.

Sibling Distance. Let L_1 and L_2 be the labels of two sibling sets. We use the Jaccard distance [50], modified for bags, to compute the distance between the sibling sets.

$$J(L_1, L_2) = 1 - \frac{2|L_1 \cap L_2|}{|L_1 \uplus L_2|} \quad (L_1 \neq \emptyset \text{ or } L_2 \neq \emptyset)$$

The sibling distance is 1 if all siblings are different, and 0 if L_1 and L_2 have identical labels.

Base Error. Let B_1 and B_2 be the bases formed over L_1 and L_2, respectively. We define the *base error*,

$$\epsilon(L_1, L_2, B_1, B_2) = |J(L_1, L_2) - J(B_1, B_2)|, \tag{4.1}$$

where $J(B_1, B_2)$ is the Jaccard distance between the bases. The base error ϵ ranges between 0 and 1, $\epsilon = 0$ means that the base distance is equivalent to the sibling distance. For $f \geq w$, small bases of size $q = 1$ have base error zero.

Example 4.3. *Let* $L_1 = \{a, c, d, f, g, i\}$ *and* $L_2 = \{a, \underline{b}, c, d, \underline{e}, f, g, \underline{h}, i\}$. *For* $q = 2$ *and* $w = 3$, *we get* $B_1 = \{ac, ad, cd, cf, df, dg, fg, fi, gi, ga, ia, ic\}$ *and* $B_2 = \{a\underline{b}, ac, \underline{b}c, \underline{b}d, cd, c\underline{e}, d\underline{e}, df, \underline{e}f, \underline{e}g, fg, f\underline{h}, g\underline{h}, gi, \underline{h}i, \underline{h}a, ia, i\underline{b}\}$. *With* $|B_1 \cap B_2|$=6, $|B_1 \uplus B_2|$=30, *and* $|L_1 \cap L_2|$=6 *the base error is* $\epsilon = \frac{2}{5}$. *For* $q = w = 3$ *no bases match, and* $\epsilon = \frac{4}{5}$.

P3: Detection of Node Moves to Other Parents. We define *base recall* and *base precision* to measure the sensitivity of the bases to node moves. A node move is detected if at least one of the bases changes. We consider bases of size $q = 2$ and discuss larger bases in the next section.

A base without dummy nodes encodes exactly one sibling pair. Due to the window wrapping, the same sibling pair may be encoded twice. Two bases formed from the same sibling pair are called duplicates. Bases with dummy nodes give no sibling information. Let $\#\text{pairs}(S, B)$ denote the number of sibling pairs of S encoded by the bases B, i.e., only bases without dummy nodes and only one copy of each duplicate are count.

Base Recall. For a sibling set S with f nodes, $\binom{f}{2} = \frac{f(f-1)}{2}$ pairs can be formed. Given the respective bases B, we define the *base recall*, ρ, as the ratio of sibling pairs encoded by the bases to the number of possible pairs.

$$\rho(S, B) = 2\frac{\#\text{pairs}(S, B)}{f(f-1)}, \quad f = |S| \tag{4.2}$$

$\rho = 1$ if all possible pairs of S are in B, $\rho = 0$ if none of the possible pairs is encoded. Bases with low recall may not encode relevant sibling pairs and thus miss node moves.

Base Precision. Ratio of sibling pairs encoded by the bases to the total number of bases. Given a sibling set S and the respective set of bases B, the base precision is

$$\pi(S, B) = \frac{\#\text{pairs}(S, B)}{|B|}. \tag{4.3}$$

$\pi = 1$ if the bases contain no duplicates/dummy nodes. In the original tree there are no dummy nodes. Low precision, i.e., many bases with dummy nodes, decrease the weight of the original nodes.

Example 4.4. *Let B over siblings S be the bases in Figure 4.3(c) ($q = 2, w = 3$). `jk` and `kj` are duplicates, all other bases contain dummy nodes, thus $\#\text{pairs}(S, B) = 1$. Base recall $\rho(S, B) = 1$ (all pairs of S are encoded by B), base precision $\pi(S, B) = \frac{1}{6}$ (only 1 of 6 bases is relevant for detecting node moves).*

4.6 Optimal Windowed *pq*-Grams

In this section we discuss the choice of the base size q and the window size w. We show that bases of size $q = 2$ have smaller base error than larger bases (Lemma 4.1), but can detect exactly the same sibling moves (Lemma 4.2). We choose $q = 2$ and compute base recall and precision (Lemma 4.3). We choose a window size w that optimize both recall and precision, and we show that all nodes in the resulting bases have equal weight (Theorem 4.3).

Lemma 4.1 (Optimal Base Size)**.** *Let S and S' be sibling sets with labels L and L', respectively, let S be transformed to S' by one of the following edit sequences:*

(a) k insertions of new nodes with labels not in L;
(b) k renamings of nodes with labels not in L ($k \leq S$);
(c) k node deletions ($k \leq S$).

For a given window size $w \leq \min(|S|, |S'|)$, small bases of size 2 ($B_{q=2}, B'_{q=2}$) have equal or smaller base error than larger bases ($B_{q>2}, B'_{q>2}$):

$$\epsilon(L, L', B_{q=2}, B'_{q=2}) \leq \epsilon(L, L', B_{q>2}, B'_{q>2})$$

Lemma 4.2 (Sibling Move Detection)**.** *Given the sibling sets S_1 and S'_1 with the bases B_1 and B'_1. We move a node n from S_1 to S'_1 and get the sibling sets S_2 and S'_2 with the bases B_2 and B'_2. For a given window size w, if the sibling move is detected for bases with $q > 2$, i.e., $B_1 \cup B'_1 \neq B_2 \cup B'_2$, then it is also detected for bases with $q = 2$.*

Lemma 4.3 (Recall and Precision)**.** *Let S be a sibling set with f nodes, B be the bases of size $q = 2$ formed over S with windows size w. Base recall, $\rho(S, B)$, and base precision, $\pi(S, B)$, are*

$$\rho = \begin{cases} 2\frac{w-1}{f-1} & w < \frac{f+1}{2} \\ 1 & w \geq \frac{f+1}{2} \end{cases} \qquad \pi = \begin{cases} 1 & w < \frac{f+1}{2} \\ \frac{f-1}{2(w-1)} & w \geq \frac{f+1}{2} \end{cases}$$

Theorem 4.3. *(Optimal Windowed pq-Grams) Given an unordered tree with fixed fanout f. For the base size $q = 2$ and the window size $w = \frac{f+1}{2}$ we get windowed pq-grams with the following base properties:*

(a) *All non-root nodes have equal frequency and appear in exactly $2w - 2$ bases.*

(b) $$\epsilon \leq \begin{cases} \frac{k}{f} & \textit{for rename} \\ \frac{2k}{2f+k} & \textit{for insert} \\ \frac{2k}{2f-k} & \textit{for delete} \end{cases}$$

(c) $\rho = 1$ (d) $\pi = 1$

The optimal base size w depends on the fanout f. For a degenerated tree (consisting only of the root node and $n - 1$ leafs) $w{=}\frac{f-1}{2}{=}O(n)$. Even in this case, the *pq*-gram profile can not grow larger than $O(n^2)$ (Theorem 4.1, $f{\geq}w$, $q{=}2$).

4.7 Algorithms

4.7.1 Building the *pq*-Gram Index

Algorithm 4.1 computes the windowed *pq*-gram profile $\mathbf{P}$ by recursively traversing the tree T in preorder. The algorithm is initialized with the root node n of T, the window size w, a stem of dummy nodes $(\bullet_1, \ldots, \bullet_p)$, and the empty profile $\mathbf{P}$. Whenever the last sibling (in document order) of a sibling set is reached, the siblings are sorted, and the *pq*-grams are produced. The runtime is $O(n)$ for documents with n nodes and a constant maximal fanout. Our experiments confirm the linear behavior of the algorithm.

The index, $\mathbf{I}$, is computed by aggregating and counting the *pq*-grams in the profile $\mathbf{P}(\mathit{treeId}, \mathit{pqg})$: $\mathbf{I} \leftarrow \Gamma_{\mathit{treeId},\mathit{pqg},\mathrm{COUNT}(*)\rightarrow \mathit{cnt}}(\mathbf{P})$. The complexity is $O(n \log n)$ (sorting the profile of size $O(n)$). The index of a forest is the union of the indexes of its trees.

Algorithm 4.1: `getPQGrams`$(\mathsf{T}, \mathtt{n}, w, \text{stem}, \mathbf{P})$

```
1  stem ← dequeue-first-element(stem) ∘ n;
2  if n is a leaf then return P ∪ {(T, stem ∘(•, •)};
3  C ← ∅;
4  foreach child c of n do
5      C ← C ∪ {c};
6      P ← P ∪ getPQGrams(T, c, w, stem, P);
7  C ← C ∪ ⋃_{i=1}^{w−f} {•};
8  a ← sort-by-label(C);
9  for i ← 0 to |a| − 1 do
10     for j ← i + 1 to i + w − 1 do
11         P ← P ∪ {(T, stem ∘ a[i] ∘ a[j  mod |a|])};
12 return P;
```

4.7.2 Approximate XML Join

Algorithm 4.2 computes the approximate *pq*-gram join ($q = 2$) of two sets of trees $\mathcal{F}_1$ and $\mathcal{F}_2$, given their indexes $\mathbf{I}_1$ and $\mathbf{I}_2$, and the threshold τ. All pairs $(\mathsf{T}_i, \mathsf{T}_j) \in \mathcal{F}_1 \times \mathcal{F}_2$ that satisfy $\text{dist}(\mathsf{T}_i, \mathsf{T}_j) \leq \tau < 1$ are returned. PS_i is initialized with the profile sizes for the trees in forest $\mathcal{F}_i$.

Algorithm 4.2: `pqGramJoin`$(\mathbf{I}_1, \mathbf{I}_2, \tau)$

```
1  foreach I_i do
2      I_i ← ρ_{treeId/treeId_i, cnt/cnt_i}(I_i);
3      PS_i ← Γ_{treeId_i, SUM(cnt_i)→size_i}(I_i);
4  return  π_{treeId_1, treeId_2}(σ_{1−2 cnt/(size_1+size_2) ≤ τ}(
5              Γ_{treeId_1, treeId_2, SUM(min(cnt_1, cnt_2))→cnt}(I_1 ⋈ I_2)
6          ⋈ PS_1 ⋈ PS_2))
```

As pointed out by Guha et al. [26], hash and sort-merge joins do not carry over to approximate tree joins that use the edit distance, since the function must be evaluated between every input pair. There is no effective way to sort trees or partition them into buckets with a hash function. The only approach readily applicable is the nested loop join [26].

This does not hold for the *pq*-gram distance. For the calculation of the *pq*-gram distance a tree is represented by its *pq*-gram index. Instead of computing the distance between each pair of trees directly, we check for each *pq*-gram in which pairs of trees it appears. We transform the distance-based join to

an equality join on all *pq*-grams represented as strings. We can apply well known techniques to optimize this join (e.g., sort-merge and hash join). The approximate join is computed by counting *pq*-grams in the join result.

In the worst case the joined forests consist of identical copies of the same tree. Let N be the cardinality of the forest, n the number of nodes per tree. The indexes are of size $O(Nn)$ for a constant maximal fanout. In a sort-merge join the complexity of sorting the relations is $O(Nn \log(Nn))$. Each *pq*-gram in one index matches $O(n)$ tuples in the other index. The overall complexity is $O(N^2 n)$. Note that for this worst case scenario the join result is of size $O(N^2)$, thus no algorithm can improve on the quadratic runtime.

Different from the nest loop join, our join algorithm can take advantage of the diversity of trees in a forest. In the best case, when no two trees in the forest share *pq*-grams, the runtime is almost linear in the index size $O(Nn \log Nn)$. In the our experiments we show the performance advantages of the optimized join for large forests.

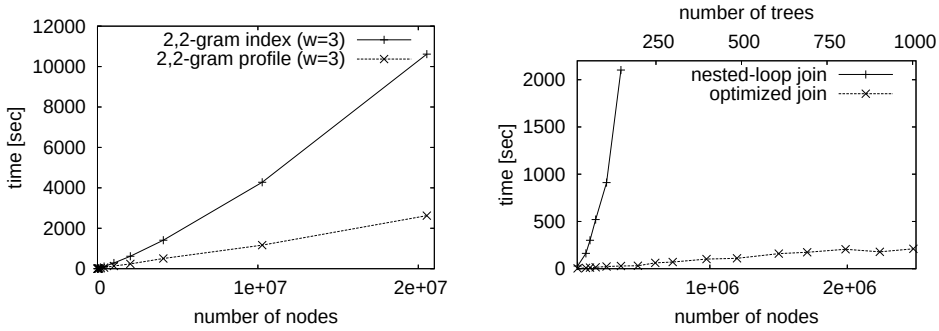
4.8 Experiments

Profile and Index Computation. We analyze the scalability of the index computation. Our test data are XML documents that range between 100kB and 1.2GB (2k to 20M nodes), $p = q = 2$ and $w = 3$. The index computation in Figure 4.5(a) includes the profile computation (Algorithm 4.1) and the aggregation of duplicate *pq*-grams within each tree. The index computation scales to very large trees. The test documents are generated with `xmlgen`, provided by the XML benchmark project XMark[2].

Approximate pq-Gram Join. We compare the scalability of the "optimized join" (Algorithm 4.2) with the scalability of a join that computes the *pq*-gram distance between each pair of documents ("nested loop join"). We use two sets of 1000 XML documents each (document size: 100 to 17000 nodes). The documents within a set are different, each document has a match in the other set. Figure 4.5(b) shows the results. The optimized join computes only the distance between documents that have *pq*-grams in common. Unlike the nested loop join, it can take advantage of the diversity of the trees that result in a small join results set. The runtime is close to linear.

Increased Stability with Window Use. We create XML documents with the following DTD:

[2] `http://monetdb.cwi.nl/xml/`

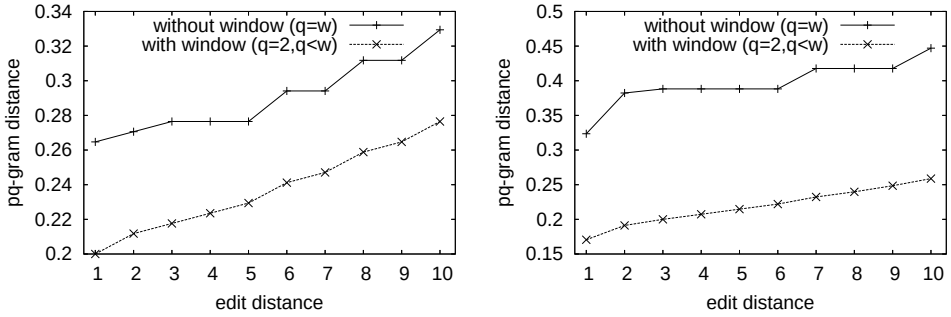


(a) Index Computation. (b) Approximate pq-Gram Join.

Figure 4.5: Index Creation and Join Scalability.

```
<!ELEMENT album (track+)>
<!ELEMENT track (artist+,title)>
<!ELEMENT artist (#PCDATA)>
<!ELEMENT title (#PCDATA)>
```

We randomly rename artist nodes and compute the pq-gram distance to the original document with and without window, i.e., $q < w$ and $q = w$, respectively. Figure 4.6 shows typical results for two different windows sizes w. Without windows the pq-gram distance shows steps and plateaus. A changed artist node changes all pq-grams that contain the renamed node. A small number of renamed artists can change all pq-grams of a track element. All following renames of artists in the same track will not further increase the distance, leading to the plateaus in the graph. This effect increases with larger values of w. With window use ($q = 2, q < w$) the graph shows no plateaus.



(a) Window Size $w = 3$. (b) Window Size $w = 5$.

Figure 4.6: Windows Increase Stability.

Quality of Matches. We use real world XML data sets and add noise (spelling mistakes and missing elements). We approximately join the original and the noisy set.

The Data Sets. We use the DBLP[3] (bibliography), the SwissProt[4] (protein sequence database), and the Treebank[5] (parts of speech tagged English sentences) XML databases. We split each database into a set of (sub)documents by deleting the root node, and we randomly choose 200 of the resulting documents for our experiments (requiring their size to be larger than the number of errors we introduce).

The resulting document sets are structurally very different: DBLP contains small and flat documents (15 nodes and depth 1.9 on average) with about ten times more elements then attributes, the SwissProt documents are larger and deeper with almost the same number of attributes and elements (104 nodes and depth 3.5 on average), the Treebank documents have deep recursive structure (49 nodes and depth 6.9 on average, with a maximum depth of 30).

Adding Noise. We modify the *original documents* by deleting and renaming random nodes. Node deletions simulate missing elements or attributes and modify the document structure. Renamed nodes represent different tag names or spelling mistakes in the text values. The resulting noisy document is the *match* of the original document, all other noisy documents are *non-matches*. In our figures we show the percentage of changed nodes (*norm-edit-dist*).

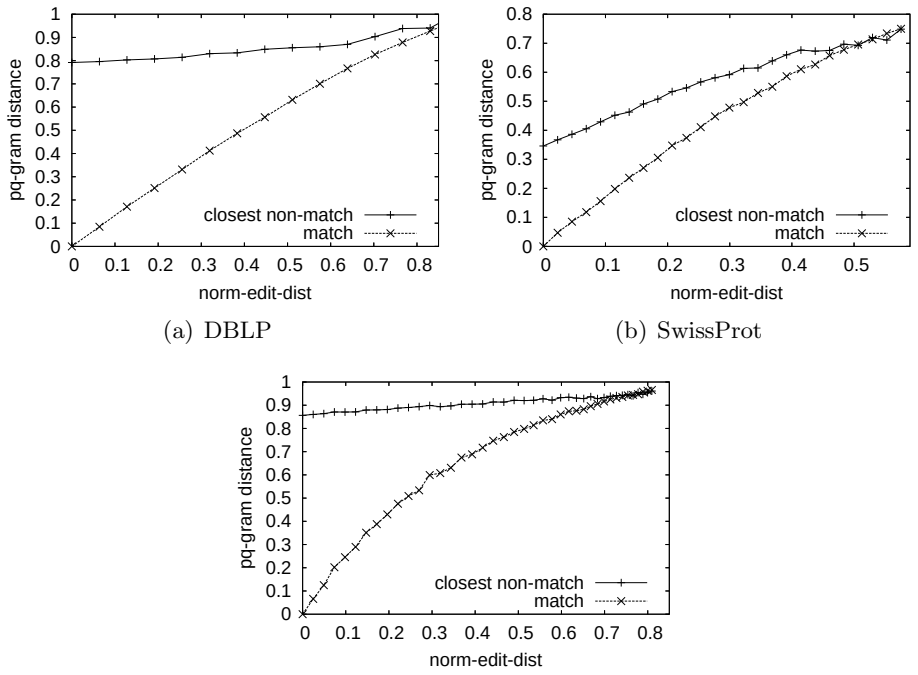
Distance between Matches and Non-Matches. Each original document has exactly one match. Figures 4.7(a)–4.7(c) show the average distance of the original documents to their match and to the closest non-match. The SwissProt documents are more similar to each other then the DBLP and Treebank documents. The *pq*-gram distance to the matches is almost linear to the number of modified nodes. It effectively approximates the edit distance. All documents are modified, thus also the distance to the non-matches increases with the number of changed nodes.

Precision and Recall. Our join algorithm matches each original document to one or more noisy documents. We count *correct* and *incorrect* matches. With *possible* we denote the maximum number of correct matches for a dataset. We compute $precision = \frac{correct}{correct+incorrect} \times 100\%$ and $recall = \frac{correct}{possible} \times 100\%$. The precision is high if the returned matches are correct, the recall is high if the algorithm does not miss correct matches.

[3] `http://dblp.uni-trier.de`

[4] `http://us.expasy.org/sprot/`

[5] `http://www.cis.upenn.edu/~treebank/`



(a) DBLP

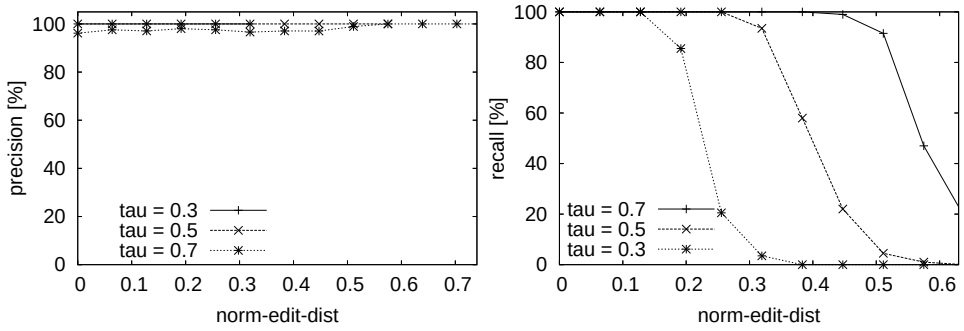
(b) SwissProt

(c) Treebank

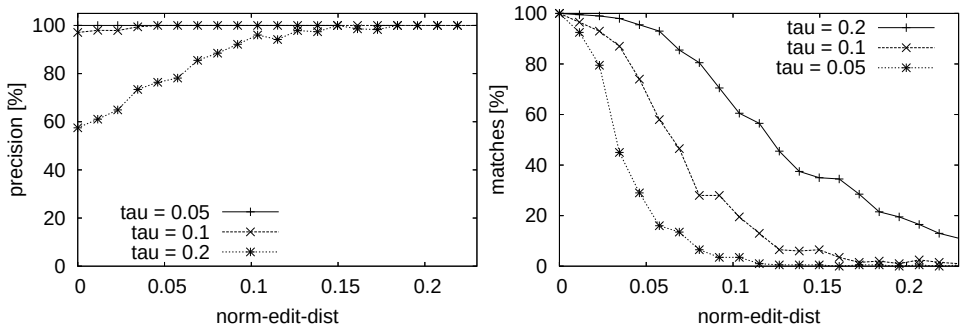
Figure 4.7: Distance between Matches and Non-Matches.

Figure 4.8 shows precision and recall for different thresholds τ. Moving up the threshold decreases the precision and increases the recall. Precision and recall for DBLP and Treebank are almost 100%, even for very noisy documents.

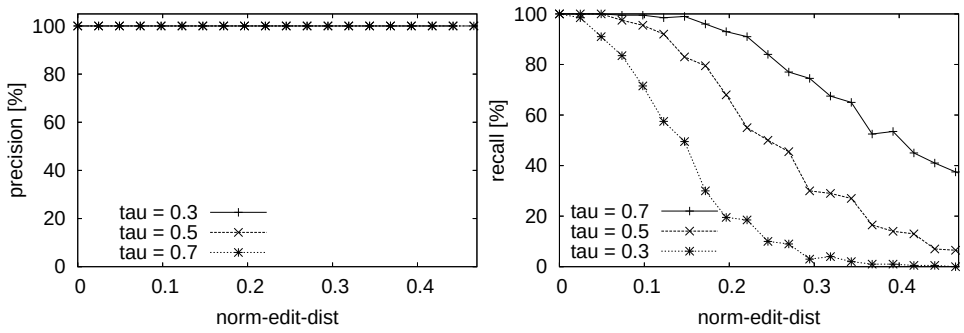
For SwissProt the precision drops as we increase the threshold. The SwissProt documents are clustered into groups of very similar documents (protein variants). For example, two documents with 64 elements have exactly the same structure and vary only in 6 text values. The clustering of the data is evident from the precision values in Figure 4.8(b) for *norm-edit-dist* = 0 (approximate self join): Already for $\tau = 0.2$ many documents match other documents then themselves. We improve the result for SwissProt using a variable threshold. Each document is matched to its nearest neighbor. If a document has more than one nearest neighbor, no match is returned. Figure 4.9 shows the results for the SwissProt database. The algorithm returns precise matches, and even for errors of 20% we miss only about 10% of the matches.



(a) DBLP



(b) SwissProt



(c) Treebank

Figure 4.8: Matching with Different Thresholds.

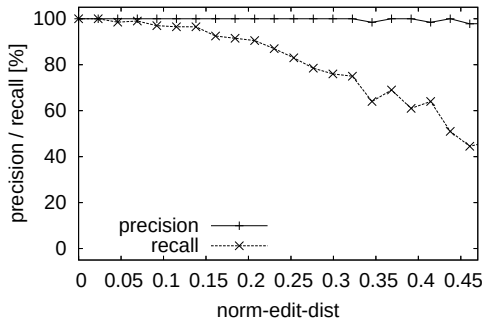


Figure 4.9: 1:1 Matches for SwissProt.

4.9 Conclusion

When XML data from different sources is integrated in a single data collection, data items that represent the same real world object must be recognized. Exact matches, however, often fail in such applications (elements may be missing in one database, content values may not match due to different coding conventions and spelling mistakes, and the data may be arranged in a different structure). Approximate matching techniques must be applied.

Previous research developed approximate join operations based on ordered tree-matching, but for data-centric XML applications the order of siblings should not matter. Data-centric XML can be represented as unordered trees. In this work we propose an approximate join technique for data-centric XML based on *pq*-grams.

pq-Grams were developed for the approximate matching of ordered trees. We introduce *windowed pq*-grams for unordered trees that are computed in a in a three-step process: sorting the tree, extending the tree with dummy nodes, and computing the *pq*-grams on the extended tree using a window mechanism. The resulting *pq*-grams consist of a stem and a base. The stems are invariant to order, the main challenge is to compute the bases. Our bases enjoy the following important properties: all base-nodes have equal frequency, the Jaccard distance between sibling sets is preserved, and node moves to other parents are detected. Due to these properties key features of *pq*-grams for ordered trees [3] carry over to unordered trees. Specifically, the *pq*-gram distance computed on the windowed *pq*-grams of two sorted trees approximates the unordered tree edit distance between these trees.

We provide an efficient algorithm for the approximate *pq*-gram join of unordered trees which is reduced to an equality join on *pq*-grams and can take advantage of well known join optimization techniques. To the best of

our knowledge, this is the first work to address the problem of approximately joining XML data without taking advantage of the document order. Extensive experiments on both synthetic and real world data confirm the analytic results and suggest that our technique is both useful and scalable.

Future work includes the investigation of persistent, updatable index structures for the windowed *pq*-gram joins. As *pq*-grams store local information, a document modification (e.g., an altered text value) affects only a limited number of *pq*-grams. The index should be updated incrementally by substituting the affected *pq*-grams only, thus avoiding the recomputation of all *pq*-grams from scratch.

Chapter 5

The Address Connector

Many different databases store information about the same or related objects in the real world. To enable collaboration between these databases, data items that refer to the same object must be identified. Residential addresses are data of particular interest as they often provide the only link between related pieces of information in different databases. Unfortunately, residential addresses that describe the same location might vary considerably and hence need to be synchronized. Non-matching street names and addresses stored at different levels of granularity make address synchronization a challenging task. Common approaches assume an authoritative reference set and correct residential addresses according to the reference set. Often, however, no reference set is available and correcting addresses with different granularity is not possible.

We present the address connector which links residential addresses that refer to the same location. Instead of correcting addresses according to an authoritative reference set, the connector defines a lookup function for residential addresses. Given a query address and a target database, the lookup returns all residential addresses in the target database that refer to the same location as the query address. The lookup supports addresses that are stored with different granularity. To achieve this the connector implements a new distance between streets that relies on both the similarity of the street names and the hierarchical structure of the addresses in a street. This allows us to match streets even if their names are completely unrelated. To align the addresses of two matching streets, we introduce a global greedy address matching algorithm that avoids the use of a threshold parameter and guarantees a stable matching. We define the concept of address containment that allows us to correctly link addresses with different granularity. The evaluation of our solution on real world data from a municipality shows that our solution is both effective and efficient.

5.1 Introduction

Large amounts of information about related objects in the real world are stored in databases. If different databases store data about the same real world object, the data must be synchronized to enable collaboration. The synchronization is non-trivial since often databases are maintained by different departments, use different coding conventions, and data items that represent the same real world object are identified through different key values.

Residential addresses are data of particular interest. They appear in many databases and are often the only link between relevant information in different databases. Unfortunately, addresses that describe the same location vary considerably as they are maintained and updated independently. A synchronization step is necessary to reconcile the addresses.

Synchronizing residential addresses is a challenging task. As an example consider Figure 5.1 with the databases from the Electricity Company and the Registration Office from the municipality of Bolzano-Bozen. We want to establish a link between residents and electricity bills using the addresses as the linking element. Both databases cover the same geographic area, but an exact match on the address attributes obviously fails.

Electricity Company (EC)

address	bill
Hermann-von-Gilm-Str. 1	€ 121
Hermann-von-Gilm-Str. 3/A	€ 71
Hermann-von-Gilm-Str. 3/B	€ 63
Hermann-von-Gilm-Str. 6	€ 0
Siegesplatz 2/A	€ 98
Siegesplatz 3/-/1	€ 32
Siegesplatz 3/-/2	€ 51
Siegesplatz 3/-/3	€ 43
Friedhofplatz 4	€ 143
Friedhofplatz 6	€ 0
Untervigli 1	€ 117
Mariengasse 1	€ 161

Registration Office (RO)

address	resident
Gilmstrasse 1	Peter
Gilmstrasse 3	Hans
Gilmstrasse 3	Renate
Gilmstrasse 3	Max
Gilmstrasse 5	Arturas
Friedensplatz 2/A/1	Markus
Friedensplatz 2/A/2	Klaudia
Friedensplatz 3	Igor
Cimitero 4	Linas
Cimitero 6/A	Francesco
Cimitero 6/B	Romans
Untervigil 1	Andrej
Marieng. 1/A	Josef

Figure 5.1: Two Databases with Residential Addresses that Cover the Same Geographic Area.

Common solutions for address synchronization assume an authoritative set of reference addresses, also termed address register, that is used to correct the residential addresses in the databases. This approach suffers from several limitations. Often an authoritative reference is not available, and it is not clear which database should be used to correct the other databases. Moreover, correcting addresses fails if the databases store ad-

dresses with different granularity levels. In Figure 5.1, `'Gilmstrasse 3'` in the RO database refers to a house, while `'Hermann-von-Gilm-Str. 3/A'` and `'Hermann-von-Gilm-Str. 3/B'` in the EC database are more detailed and refer to different entrances in the same house. It is not possible to change the less detailed address to a more detailed one since it is not clear how to assign the residents `Hans`, `Renate`, and `Max` to the more detailed addresses. Vice versa, correcting a detailed address to a less detailed one is not acceptable as information gets lost.

Contributions: We present a new data structure, called the *address connector*, which links residential addresses from different databases that refer to the same location. The address connector can be represented as a relation, where each tuple defines a residential address and establishes a link between two other residential addresses. A key feature of our solution is that an authoritative reference is not needed. Instead we establish links that equally respect all participating addresses. At the core of the address connector is the *synchronization operator*, which establishes the links between different addresses that refer to the same location. The synchronization operator faces two key problems. First, streets must be matched even if different databases use different names for the same street (e.g., due to misspellings, different coding conventions, or renamed streets). Second, addresses that are stored with different granularity must be linked correctly, although there is no one-to-one correspondence between them.

We introduce a structure-aware distance measure between two streets, called *street distance*, which relies on both the name of the two streets and the addresses of the two streets. Toward this end the residential addresses of a street are organized in an ordered, labeled tree, called address tree. The root of the address tree is the set of all known names of the street, while the rest of the tree represents house numbers, entrance numbers, and apartment numbers (see Figure 5.2). The street distance relies on both the structural similarity of the address trees and the similarity of the street names. Such a structure-aware approach allows us to match streets even if the names are completely unrelated (e.g., in the case of renamed streets) or if the structure of the address trees is ambiguous (e.g., the address trees of the streets `'Mariengasse'` and `'Untervigil'` in Figure 5.1 have identical structure).

Given the distance between all pairs of streets, the streets need to be matched. A constraint of the matching is that a street can have at most one matching partner. A threshold-based approach will not work since the same threshold may be too high for some streets (they are matched multiple times), but too low for other streets (they remain unmatched). We present the *global*

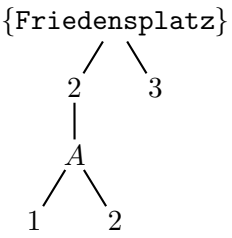


Figure 5.2: Address Tree of `'Friedensplatz'` (Registration Office Database).

greedy matching algorithm that matches each street to at most one other street and guarantees a stable matching. A matching is stable if no *new* street pair can be found such that the streets in the new pair are closer to each other than to their current partner in the matching.

Finally, the addresses of two matching streets must be linked. In general, there is no one-to-one correspondence between all addresses of two streets since they might be stored with different granularity. We introduce the *address containment*. Intuitively, an address, a, contains another address, b, if the location referred to by a contains the location referred to by b. For example, `Friedensplatz 2` is a house that contains the apartments `Fiedensplatz 2/A/1` and `Fiedensplatz 2/A/2`. We propose an efficient merge algorithm that correctly links addresses with different granularity by checking, in addition to equality, also for address containment.

To summarize, we introduce the address connector that offers lookups of residential addresses in different databases. At the core of the address connector is the synchronization operator which establishes links between residential addresses that refer to the same location. The main features of the synchronization operator are: a new street distance based on address trees; a global greedy matching algorithm that computes stable street pairs; and the concept of address containment that allows to correctly link addresses with different granularity levels. We implemented the connector and evaluated it with real-world data from the municipality of Bolzano-Bozen. The experiments show the effectiveness and efficiency of the connector.

Outline: In the next section we define and motivate the problem. We outline the solution in Section 5.3. In Section 5.4 we give solutions for computing the distances between pairs of streets, matching streets with our global greedy matching algorithm, and linking addresses with different granularity. In Section 5.5 we provide algorithms for linking residential addresses. The

algorithms are experimentally evaluated in Section 5.6. Section 5.7 discusses related work, in Section 5.8 we draw conclusions and point to future work.

5.2 Problem Definition and Motivation

5.2.1 Problem Definition

We assume different databases that store residential addresses about the same geographic area. The residential addresses reference houses, house entrances, or apartments. The street names may be spelled in different languages or a changed street name may not be reflected in some databases. The addresses may be stored with different granularity, e.g., one database may store only the address of a house without specifying entrance or apartment number while another database may store also entrance and apartment numbers for the same house.

Our goal is an effective and efficient lookup of residential addresses in different databases. The input for the lookup are a residential address defined in one of the databases (query address) and a target database. The lookup returns the set of all addresses in the target database that refer to the same location as the query address.

Example 5.1. *Consider the two address databases in Figure 5.1, and let the Registration Office be the target database. The lookup of* `'Hermann-von-Gilm-Str. 1'` *should return* {`'Gilmstrasse 1'`}*, i.e., the query and the result address are equivalent. The lookup of* `'Siegesplatz 3/-/1'` *should return* {`'Fiedensplatz 3'`}*, i.e., the query address is more detailed and is contained in the result address. Finally, the lookup of* `'Friedhofplatz 6'` *should return* {`'Cimitero 6/A'`, `'Cimitero 6/B'`}*, i.e., the result addresses are more detailed and are contained in the query address.*

5.2.2 Motivation

Our work is motivated by an application scenario from the Municipality of Bolzano-Bozen. Many administrative tasks performed by the civil servants require to combine information from different databases. The databases are maintained by internal (e.g., the registration office, the GIS office) or external departments of the municipality (e.g., the Electricity Company, the Land Registration Office, the Catastre). As residential addresses are often the only link between tuples in different databases, they must be used to access and connect related pieces of information.

Consider the two databases in Figure 5.1. The Registration Office (`RO`) stores residents of apartments, the Electricity Company (`EC`) stores the amount of the electricity bill of each apartment. For tax fraud detection the municipality wants to compute a list of all apartments for which no electricity is payed although they have residents. To answer this query, the two databases have to be joined over corresponding residential addresses.

Unfortunately, exact matches between addresses mostly fail since the addresses differ substantially for a number of reasons: `'Untervigil'` is misspelled in one database; street names are coded using different conventions, e.g., `'Hermann-von-Gilm-Str.'` vs. `'Gilmstrasse'`); `'Friedensplatz'` was renamed to `'Siegesplatz'`, but the change was not reflected in all databases; in the bilingual region of Bolzano-Bozen two names for each street exist and they are used interchangeably, e.g., `'Friedhofplatz'` and `'Cimitero'` are the German and Italian names of the same street. In addition to non-matching street names, the residential addresses are stored with different granularity in the different databases (e.g., with or without entrance/apartment numbers), and there is no one-to-one correspondence between them. For example, `'Friedhofplatz 6'` is a house that is divided into two parts with different entrances, `'Cimitero 6/A'` and `'Cimitero 6/B'`.

There is no authoritative reference database available to solve conflicts or to correct addresses. All input addresses have the same priority, and no input address can be deleted during the synchronization process. The synchronization must be extensible to additional databases. A mapping between only two address databases is of limited use as multiple departments need to interact and new services provided by the public administration require new departments to join the synchronization.

5.3 The Connector

In this section we introduce and define the connector as new data structure that supports the synchronization of residential addresses from different databases. The connector is represented as a relation. A tuple in the connector defines a residential address and establishes a link between two other residential addresses (see Figure 5.3). A residential address is a reference to a physical object that is either a house, a part of a house that has its own entrance, or an apartment in a house. Residential addresses are grouped into partitions, and the addresses of a partition are grouped into streets.

Definition 5.1 (Connector, Residential Address, Partition, Street)**.** *A connector, $\mathfrak{X}$, is a relation. A tuple $(id, a, c_1, c_2) \in \mathfrak{X}$ is identified by id, defines*

$\mathfrak{X}$

ID	*addr*	*ref1*	*ref2*
$(\mathcal{A}, \alpha_2, a_1)$	$(\{$`Hermann-von-Gilm-Str.`$\}, 1, @, @)$	ϵ	ϵ
$(\mathcal{A}, \alpha_2, a_2)$	$(\{$`Hermann-von-Gilm-Str.`$\}, 3, A, @)$	ϵ	ϵ
$(\mathcal{A}, \alpha_2, a_3)$	$(\{$`Hermann-von-Gilm-Str.`$\}, 3, B, @)$	ϵ	ϵ
$(\mathcal{A}, \alpha_2, a_4)$	$(\{$`Hermann-von-Gilm-Str.`$\}, 6, @, @)$	ϵ	ϵ
...	...	...	...
$(\mathcal{A}, \alpha_1, a_{10})$	$(\{$`Friedhofplatz`$\}, 6, @, @)$	ϵ	ϵ
...	...	...	...
$(\mathcal{B}, \beta_1, b_1)$	$(\{$`Gilmstrasse`$\}, 1, @, @)$	ϵ	ϵ
$(\mathcal{B}, \beta_1, b_2)$	$(\{$`Gilmstrasse`$\}, 3, @, @)$	ϵ	ϵ
$(\mathcal{B}, \beta_1, b_3)$	$(\{$`Gilmstrasse`$\}, 5, @, @)$	ϵ	ϵ
...	...	...	...
$(\mathcal{B}, \beta_3, b_8)$	$(\{$`Cimitero`$\}, 6, A, @)$	ϵ	ϵ
$(\mathcal{B}, \beta_3, b_9)$	$(\{$`Cimitero`$\}, 6, B, @)$	ϵ	ϵ
...	...	...	...
$(\mathcal{C}, \gamma_1, c_1)$	$(\{$`Gilmstrasse, Hermann-von-Gilm-Str.`$\}, 1, @, @)$	$(\mathcal{A}, \alpha_2, a_1)$	$(\mathcal{B}, \beta_1, b_1)$
$(\mathcal{C}, \gamma_1, c_2)$	$(\{$`Gilmstrasse, Hermann-von-Gilm-Str.`$\}, 3, A, @)$	$(\mathcal{A}, \alpha_2, a_2)$	$(\mathcal{B}, \beta_1, b_2)$
$(\mathcal{C}, \gamma_1, c_3)$	$(\{$`Gilmstrasse, Hermann-von-Gilm-Str.`$\}, 3, B, @)$	$(\mathcal{A}, \alpha_2, a_3)$	$(\mathcal{B}, \beta_1, b_2)$
$(\mathcal{C}, \gamma_1, c_4)$	$(\{$`Gilmstrasse, Hermann-von-Gilm-Str.`$\}, 5, @, @)$	ϵ	$(\mathcal{B}, \beta_1, b_3)$
$(\mathcal{C}, \gamma_1, c_5)$	$(\{$`Gilmstrasse, Hermann-von-Gilm-Str.`$\}, 6, @, @)$	$(\mathcal{A}, \alpha_2, a_4)$	ϵ
...	...	...	...
$(\mathcal{C}, \gamma_3, c_{12})$	$(\{$`Cimitero, Friedhofplatz`$\}, 6, A, @)$	$(\mathcal{A}, \alpha_1, a_{10})$	$(\mathcal{B}, \beta_3, b_8)$
$(\mathcal{C}, \gamma_3, c_{13})$	$(\{$`Cimitero, Friedhofplatz`$\}, 6, B, @)$	$(\mathcal{A}, \alpha_1, a_{10})$	$(\mathcal{B}, \beta_3, b_9)$
...	...	...	...

Figure 5.3: Connector $\mathfrak{X}$ after the Synchronization $\text{synch}_{\mathcal{A},\mathcal{B}\to\mathcal{C}}(\mathfrak{X})$.

the residential address a, and establishes a link between the two residential addresses c_1 and c_2, where c_1 and/or c_2 may be empty *(ϵ). A residential address is a tuple $(strName, num, entr, apt)$ that consists of a non-empty set of street names, a house number, an entrance and an apartment number (entrance and apartment number may be null values, @). The addresses of $\mathfrak{X}$ are grouped into partitions, and each address is in exactly one partition. A* street *is a set of addresses that are all in the same partition and have identical street names.*

The semantics of a tuple $(id, a, c_1, c_2) \in \mathfrak{X}$ in the connector is that a, c_1, and c_2 refer all to the same location. The identifier, id, of the tuple is a triple of partition identifier, street identifier, and address identifier (local to the partition). Whenever possible we refer to an address only by the local address identifier. With $\text{str}(\mathcal{A})$ we denote the set of all streets of a partition $\mathcal{A}$. $\text{names}(\alpha)$ denotes the set of street names of a street α. The *relative part* of an address c, $\text{rel}(c) = (num, entr, apt)$, is the triple of house number, entrance, and apartment number defined by the address c.

Example 5.2. *The last tuple of the connector in Figure 5.3 defines the address $c_{13} = (\{$`Cimitero, Friedhofplatz`$\}, 6, B, @)$ of partition $\mathcal{C}$, and it links the two addresses a_{10} of partition $\mathcal{A}$ and b_9 of partition $\mathcal{B}$. Address c_{13} is in street γ_3 which has two street names, i.e.,* $\text{names}(\gamma_3) =$

{Cimitero, Friedhofplatz}. *The relative part of* c_{13}, $\text{rel}(c_{13}) = (6, B, @)$, *consists of house number* 6, *entrance* B, *and a null value for the apartment number. Partition* $\mathcal{A}$ *consists of the streets* $\text{str}(\mathcal{A}) = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$ *(only two of them are shown in the figure).*

In order to support the synchronization of residential addresses, the connector provides the following main functionalities:

- *load*$(\mathfrak{X}, \texttt{DB}, \mathcal{A})$: Load an address database into the connector. The residential addresses in DB are stored as a new partition, $\mathcal{A}$, in the connector $\mathfrak{X}$. The tuples in the partition define the residential addresses of DB, and they include dummy links to empty addresses.
- *synch*$(\mathfrak{X}, \mathcal{A}, \mathcal{B}, \mathcal{C})$: Synchronize the two partitions $\mathcal{A}$ and $\mathcal{B}$ and store the result in a new partition $\mathcal{C}$. The tuples in the new partition, $\mathcal{C}$, align addresses from $\mathcal{A}$ and $\mathcal{B}$ that refer to the same location. Each tuple defines a new address.
- *lookup*$(\mathfrak{X}, (\mathcal{A}, \alpha, a), \mathcal{B})$: Retrieve from partition $\mathcal{B}$ those addresses that are aligned with the address a from partition $\mathcal{A}$.

Example 5.3. *Consider the databases in Figure 5.1 and the tax-fraud query, which requires a join of the two databases over corresponding residential addresses. Using the connector, the residential addresses of the two databases are first loaded, i.e.,* load$(\mathfrak{X}, \texttt{EC}, \mathcal{A})$ *and* load$(\mathfrak{X}, \texttt{RO}, \mathcal{B})$. *Two new partitions,* $\mathcal{A}$ *and* $\mathcal{B}$, *are created in the connector* $\mathfrak{X}$. *Next, the partitions* $\mathcal{A}$ *and* $\mathcal{B}$ *are synchronized by calling* synch$(\mathfrak{X}, \mathcal{A}, \mathcal{B}, \mathcal{C})$. *The tuples in the new partition,* $\mathcal{C}$, *establish links between addresses from* $\mathcal{A}$ *and* $\mathcal{B}$ *that refer to the same location, and each tuple defines a new address. For example,* $a_3 = (\texttt{Hermann-von-Gilm-Str.}, 3, B, @)$ *and* $b_2 = (\texttt{Gilmstrasse}, 3, @, @)$ *are linked (*a_3 *is an entrance of the house* b_2*) and define the new address* c_3. *Finally, we take the addresses that have an electricity bill with amount zero and do a lookup in the* RO *database to find residents who do not pay for their electricity. For example,* lookup$(\mathfrak{X}, (\mathcal{A}, \alpha_1, a_{10}), \mathcal{B})$ *retrieves the set* $\{b_8, b_9\}$ *representing two entrances of the house* a_{10}. *Thus, the apartments* 'Cimitero 6/A' *and* 'Cimitero 6/B' *have residents but do not pay for the electricity.*

The synchronization operator is the most important one and will be described in more detail below.

5.4 The Synchronization Operator

The synchronization operator, $\text{synch}_{\mathcal{A},\mathcal{B}\to\mathcal{C}}(\mathfrak{X})$, aligns the addresses of the partitions $\mathcal{A}$ and $\mathcal{B}$ and stores the result in a new partition $\mathcal{C}$. A tuple in the new

partition establishes a link between two addresses of $\mathcal{A}$ and $\mathcal{B}$ that refer to the same location, and the address defined by the tuple represents the linked addresses.

5.4.1 Overview

The synchronization of two partitions, $\mathcal{A}$ and $\mathcal{B}$, is a three-step process:

1. *Computing Street Distances:* Given two streets, $\alpha \in \text{str}(\mathcal{A})$ and $\beta \in \text{str}(\mathcal{B})$, the distance, $\text{dist}(\alpha, \beta)$, between the two streets is computed.

   Input: $\alpha \in \text{str}(\mathcal{A})$, $\beta \in \text{str}(\mathcal{B})$
   Output: $\text{dist}(\alpha, \beta) \in [0..1]$

2. *Matching Streets:* Assume the streets of two partitions, $\text{str}(\mathcal{A}) = \{\alpha_1, \ldots, \alpha_M\}$ and $\text{str}(\mathcal{B}) = \{\beta_1, \ldots, \beta_N\}$, $M \leq N$, and a distance matrix $D_{M \times N}$ with the distance between streets α_i and β_j, $\text{dist}(\alpha_i, \beta_j)$, in row i and column j. A matching, M, between the streets is computed, such that each street of $\mathcal{A}$ matches at most one street of $\mathcal{B}$ and no two streets of $\mathcal{A}$ match the same street of $\mathcal{B}$.

   Input: $\text{str}(\mathcal{A})$, $\text{str}(\mathcal{B})$, $D_{M \times N}$
   Output: $\mathsf{M} \subseteq \text{str}(\mathcal{A}) \times \text{str}(\mathcal{B})$,
   $\forall \alpha \in \text{str}(\mathcal{A}) \exists! \beta \in \text{str}(\mathcal{B}) : (\alpha, \beta) \in \mathsf{M}$,
   $(\alpha, \beta) \in \mathsf{M} \wedge (\alpha, \beta') \in \mathsf{M} \Rightarrow \beta = \beta'$

3. *Linking Addresses:* Links between the addresses of two streets, $(\alpha, \beta) \in \mathsf{M}$, are established. Two addresses are linked if they refer to the same location. An address that has no counterpart in the other street is linked to the empty address (ϵ). Each link produces a new connector tuple, and the address defined by the new tuple represents the linked addresses. The set $\bar{\gamma}$ of new tuples defines a new street γ.

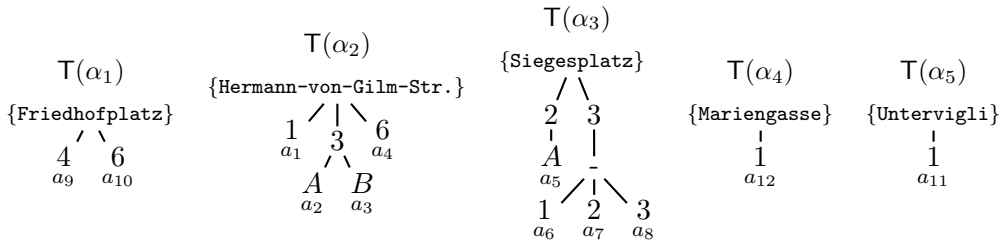
   Input: $(\alpha, \beta) \in \mathsf{M}$
   Output: $\bar{\gamma} \subseteq (\text{names}(\alpha) \cup \text{names}(\beta)) \times \text{rel}(\alpha \cup \beta) \times (\alpha \cup \{\epsilon\}) \times (\beta \cup \{\epsilon\})$

In the following we discuss each of these steps in detail.

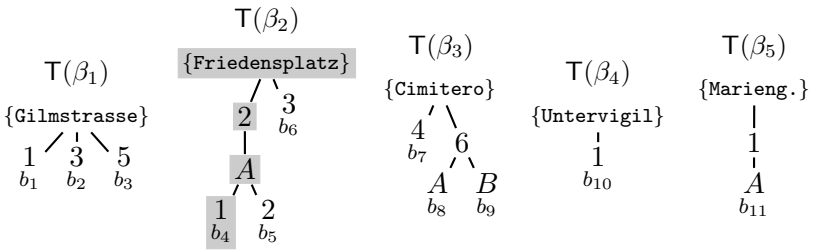
5.4.2 Step 1: Computing Street Distances

To compute the similarity of two streets we introduce and define a new street distance which is based on two independent characteristics: the name of the two streets and the structure of the addresses of the two streets. For that we have to represent each street by its address tree.

Address Trees: The addresses of a street, α, define a hierarchy and are represented as a so-called *address tree*, $\mathsf{T}(\alpha)$ [2]. Figure 5.4 shows the address trees of the streets in partitions $\mathcal{A}$ and $\mathcal{B}$ of connector $\mathfrak{X}$ (see Figure 5.3). The root of an address tree is the set of names of the corresponding street, the children of the root are the house numbers, the children of house numbers are the entrance numbers, and the children of entrance numbers are the apartment numbers. An address is a path from the root to a leaf node. For example, the shaded path in Figure 5.4(b) is the address `'Friedensplatz 2/A/1'`. The identifiers of addresses that are defined by a root-leaf path are shown below the respective leaf. We omit null values in the address trees.



(a) Address Trees of the Electricity Company (Partition $\mathcal{A}$).



(b) Address Trees of the Registration Office (Partition $\mathcal{B}$).

Figure 5.4: Example Address Trees.

The Name Distance

The root node of an address tree represents the set of all known names of the corresponding street. We define the *name distance* between two address trees, $\mathsf{T}(\alpha)$ and $\mathsf{T}(\beta)$, as the minimum distance between two of their names, $n_\alpha \in \text{names}(\alpha)$ and $n_\beta \in \text{names}(\beta)$. We use the q-gram distance to determine the distance between a single pair of street names. The q-grams of a street name are all its substrings of length q. Intuitively two street names are similar if they have many q-grams in common.

Definition 5.2 (q-Gram Distance)**.** *Given a string s of characters from a finite alphabet Σ and the extended string s' that is formed by prefixing and suffixing s with $q-1$ characters that are not in Σ. A q-gram of s is a substring of length q of the extended string s', and $\mathbf{I}(s)$ is the bag of all q-grams of s. The q-gram distance between two street names, s_1 and s_2, is defined as*

$$\text{dist}_q(s_1, s_2) = 1 - 2\frac{|\mathbf{I}(s_1) \cap \mathbf{I}(s_2)|}{|\mathbf{I}(s_1) \uplus \mathbf{I}(s_2)|}.$$

The distance is normalized and can take values between 0 and 1. The q-gram distance is 0 if $s_1 = s_2$.

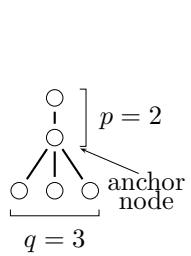
Example 5.4. *We compute the name distance between the address trees $\mathsf{T}(\alpha_5)$ and $\mathsf{T}(\beta_4)$ in Figure 5.4. Both root nodes store only one name, and the name distance is equal to the q-gram distances between these names. n_α = 'Untervigli', n_β = 'Untervigil', the respective q-gram bags are $\mathbf{I}(n_\alpha)$ = {##U, #Un, Unt, nte, ter, erv, rvi, vig, igl, gli, li#, i##} and $\mathbf{I}(n_\beta)$ = {##U, #Un, Unt, nte, ter, erv, rvi, vig, igi, gil, il#, l##}, the q-gram distance is* $\text{dist}_q(n_1, n_2) = 1 - 2\frac{|\mathbf{I}(n_1) \cap \mathbf{I}(n_2)|}{|\mathbf{I}(n_1) \uplus \mathbf{I}(n_2)|} = 1 - 2\frac{8}{24} = \frac{1}{3}$.

The Structure Distance

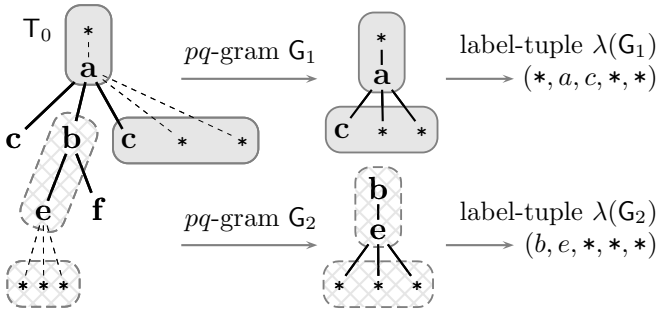
Intuitively, the structure distance of two streets considers how the (recorded) addresses of the two streets differ. If the addresses of a street are represented in an address tree, this measure can be defined as the structural distance between two address trees, and we will use pq-grams to measure the distance of two trees.

A pq-gram is a small, besom-shaped subtree consisting of an anchor node, $p-1$ ancestors, and q consecutive children. Intuitively, the pq-grams are formed by shifting a pq-gram shaped pattern over the tree (see Figure 5.5). The nodes covered by the pattern form a pq-gram. The pattern is shifted such that each node appears in the anchor node position and each non-root node also in each leaf position of the pattern. We fill in dummy nodes for the parts of the pattern that extend beyond the tree border. For the following definitions we assume an ordered, labeled, rooted tree T. Each node n of T has a label $\lambda(\mathsf{n})$. A node with the special label '$*$' is a *dummy node.*

Definition 5.3 (pq-Gram)**.** *Let T be a tree, a be a node of T, $p > 0$, $q > 0$, and let $\mathsf{T}^{p,q}$ be T extended with dummy 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 *of T with* anchor node a *is a subtree of $\mathsf{T}^{p,q}$ that is composed of the following nodes: p nodes $\mathsf{a}_{p-1}, \ldots, \mathsf{a}_1, \mathsf{a}$, where*



(a) *pq*-Gram Pattern.



(b) Example Tree T_0 and Two 2, 3-Grams of T_0.

Figure 5.5: Computing the *pq*-Grams of a Tree.

a_i *is the ancestor of* a *at distance* i*, and* q *contiguous children* $\mathsf{c}_k, \ldots, \mathsf{c}_{k+q-1}$ *of* a.

We use a linear encoding and represent a *pq*-gram G with anchor node a as a tuple of its node labels, the *label-tuple* $\lambda(\mathsf{G}) = (\lambda(\mathsf{a}_{p-1}), \ldots, \lambda(\mathsf{a}_1), \lambda(\mathsf{a}), \lambda(\mathsf{c}_k), \ldots, \lambda(\mathsf{c}_{k+q-1}))$. As the labels of a tree are not necessarily unique, two *pq*-grams of the same tree may yield identical label-tuples. The *pq*-gram distance is based on the number of label-tuples that two trees have in common.

We ignore the street names when we compute the structure distance between address trees and denote with $\mathsf{T}^*(\gamma)$ the address tree of street γ with a dummy root node. The structure of two address trees is similar if the trees are within a small *pq*-gram distance. The *pq*-gram distance is computed by splitting the trees into small subtrees of a specific shape (*pq-grams*). Trees that share a high percentage of *pq*-grams are more similar than trees that share a low percentage.

Definition 5.4 (*pq*-Gram Distance)**.** *Let* $\mathbf{I}(\mathsf{T})$ *denote the bag of all label-tuples of a tree* T*. The* *pq*-gram distance *between two trees,* T_1 *and* T_2*, is defined as*

$$\mathrm{dist}_{pq}(\mathsf{T}_1, \mathsf{T}_2) = 1 - 2\frac{|\mathbf{I}(\mathsf{T}_1) \cap \mathbf{I}(\mathsf{T}_2)|}{|\mathbf{I}(\mathsf{T}_1) \uplus \mathbf{I}(\mathsf{T}_2)|}.$$

The *pq*-gram distance is normalized and can take values between 0 and 1. If T_1 and T_2 have identical structure and labels, the *pq*-gram distance is 0.

Example 5.5. *We compute the structure distance between the address trees* $\mathsf{T}(\alpha_1)$ *and* $\mathsf{T}(\beta_3)$ *in Figure 5.4 using the pq-gram distance (*$p = 2$*,* $q = 3$*). The root nodes are substituted by dummy nodes, the label-tuples are computed, and*

the pq-distance is computed by intersecting the bags of label-tuples:

$$\begin{aligned}
\mathbf{I}(\mathsf{T}^*(\alpha_1)) &= \{(*,*,*,*,4),(*,*,*,4,6),(*,*,4,6,*),(*,*,6,*,*),\\
&\quad (*,6,*,*,*),(*,4,*,*,*)\},\\
\mathbf{I}(\mathsf{T}^*(\beta_3)) &= \{(*,*,*,*,4),(*,*,*,4,6),(*,*,4,6,*),(*,*,6,*,*),\\
&\quad (*,6,*,*,A),(*,6,*,A,B),(*,6,A,B,*),(*,6,B,*,*),\\
&\quad (*,4,*,*,*),(6,A,*,*,*),(6,B,*,*,*)\}\\
\mathrm{dist}_{pq}(\mathsf{T}^*(\alpha_1),\mathsf{T}^*(\beta_3)) &= 1-2\tfrac{|\mathbf{I}(\mathsf{T}^*(\alpha_1))\cap\mathbf{I}(\mathsf{T}^*(\beta_3))|}{|\mathbf{I}(\mathsf{T}^*(\alpha_1))\uplus\mathbf{I}(\mathsf{T}^*(\beta_3))|} = 1-2\tfrac{5}{17} = 0.42
\end{aligned}$$

The Street Distance

Depending on the input data, more reliable matches can be expected from either the name distance (e.g., both address sets use the same language and similar coding conventions) or the structure distance between the address trees (e.g., the input sets use different languages). We weight the name distance with ω and structure distance with $1-\omega$, and we combine the two distances into a single distance between address trees. Let d_n be the name distance and d_s the structure distance, the *address tree distance* is defined as

$$d = \sqrt{\omega d_n^2 + (1-\omega)d_s^2}.$$

Example 5.6. *The name distance $d_n = 0.5714$ between the renamed streets β_2 (*`Friedensplatz`*) and α_3 (*`Siegesplatz`*) is larger than the name distance $d_n = 0.3333$ between β_2 and α_1 (*`Friedhofplatz`*). As the renamed streets are structurally more similar ($d_t = 0.5758$ vs. $d_t = 1.0$ between β_2 and α_1), the street distance ($w = 0.5$) between these streets is smaller then the street distance between β_2 and α_1 and they are matched correctly (see Figure 5.6).*

5.4.3 Step 2: Matching Streets

Given the distance between all street pairs of two partitions, the streets need to be matched. We define the matching as a set of street pairs, where each street appears in only one pair. Our goal is to compute a *stable* matching. Intuitively, a matching is stable if it is not possible to break up existing matches and form a new match, such that the new match is better than the old matches for both matching partners.

Definition 5.5 (Matching and Stable Matching)**.** *A* matching, $\mathsf{M} \subseteq \mathrm{str}(\mathcal{A}) \times \mathrm{str}(\mathcal{B})$*, of the streets of two partitions,* $\mathrm{str}(\mathcal{A})$ *and* $\mathrm{str}(\mathcal{B})$*, is a set of street pairs (*matches*), where each street $\alpha \in \mathrm{str}(\mathcal{A})$ is paired with at most one street $\beta \in \mathrm{str}(\mathcal{B})$, and each street $\beta \in \mathrm{str}(\mathcal{B})$ is paired with at most one street $\alpha \in \mathrm{str}(\mathcal{A})$.* M *is* stable *if there is no pair $(\alpha,\beta) \notin \mathsf{M}$, such that α is closer*

to β than to its current partner in M, and β is closer to α than to its current partner in M:

$$\nexists(\alpha,\beta) \in (\text{str}(\mathcal{A}) \times \text{str}(\mathcal{B})) \setminus \mathsf{M}: \quad \forall(\alpha,y) \in \mathsf{M} : \text{dist}(\alpha,y) > \text{dist}(\alpha,\beta) \wedge \forall(x,\beta) \in \mathsf{M} : \text{dist}(x,\beta) > \text{dist}(\alpha,\beta) \tag{5.1}$$

Let D be the $M \times N$ distance matrix that stores the distances between the streets of the two partition, $\text{str}(\mathcal{A}) = \{\alpha_1, \ldots, \alpha_M\}$ and $\text{str}(\mathcal{B}) = \{\beta_1, \ldots, \beta_N\}$. The distance between the streets α_i and β_j is stored in row i and column j of D. Figure 5.6 shows the distance matrix for the address trees in Figure 5.4. Name distance and structure distance are equally weighted ($w = 0.5$), the correct matches are shaded.

	α_1	α_2	α_3	α_4	α_5
β_1	1.0	0.7761	1.0	0.6796	0.8498
β_2	0.7454	1.0	0.5736	0.9649	1.0
β_3	0.7647	0.9592	1.0	1.0	0.9556
β_4	1.0	0.8584	1.0	0.7071	0.2357
β_5	0.9608	0.9199	1.0	0.4241	0.7767

Figure 5.6: Distance Matrix for the Address Trees in Figure 5.4.

Our solution for the street matching problem is the *global greedy matching*. Such an approach matches close street pairs first and avoids missing good matches due to earlier miss-matches. Matched streets are marked, and no street is matched twice. The matching produced by the algorithm is stable.

Example 5.7. *Consider the distance matrix in Figure 5.6. The global greedy matching computes the matches in the following order:* (α_4, β_5), (α_5, β_4), (α_2, β_3), (α_3, β_1), (α_1, β_2).

Note that matching two streets if they are within a fixed distance threshold is not good enough. The threshold may be too low for some streets (they remain unmatched), but too high for others (they are matched to multiple streets in the other partition). Often it is impossible to set a good threshold. A local greedy approach traverses the streets of one partition in random order and matches each street to its nearest neighbor in the other partition. If the nearest neighbor of a street is already matched, the next-nearest neighbors are visited, until an unmatched street is found. Each street is matched only once, but the quality of the matching depends on the random matching order. Both approaches can not guarantee stable matches.

Example 5.8. *Consider, for example, the distance matrix in Figure 5.6. A threshold equal or above* 0.6796 *matches both,* α_1 *and* α_5*, with* β_4*. For a lower threshold* α_1 *and* α_3 *remain unmatched. The local greedy algorithm matches each row to the unmatched column with the smallest distance value in the respective row. We match the rows in the order given by the distance matrix (first row first) and get the matching* $\mathsf{M} = \{(\alpha_1, \beta_4), (\alpha_2, \beta_3), (\alpha_3, \beta_1), (\alpha_4, \beta_5), (\alpha_5, \beta_2)\}$*. As* α_1 *is miss-matched to* β_4 *in the beginning,* α_5 *can not be matched to its nearest neighbor* β_4*, but it is matched to the remaining street* β_2 *which is very distant from* α_5*.*

5.4.4 Step 3: Linking Addresses

In this section we establish links between the addresses of two streets. Each link is represented by a new tuple in the connector. The new tuples define a new street and each of the linked addresses is represented by one or more addresses in the new street.

Two addresses should be linked if they refer to the same location. It is not enough to check whether the relative parts of the addresses are equivalent as the addresses may be stored with different granularity. For example, `'Gilmstrasse 3'` should match both `'Hermann-von-Gilm-Str. 3/A'` and `'Hermann-von-Gilm-Str. 3/B'`, but the entrance is not specified in `'Gilmstrasse 3'`. We define the concept of address equivalence and address containment.

Definition 5.6 (Address Equivalence and Containment)**.** *Given two residential addresses, a and b. Address a is* equivalent *to address b (*$a \equiv b$*) if both addresses refer to the same physical object. a* contains *b (*$a \sqsupseteq b$*) if and only if b refers to an object that is part of the object referred to by a or* $a \equiv b$*.*

The input for the address linking are the street pairs provided by the street matching algorithm. The addresses of two matched streets refer to locations in the same real world street, thus addresses with identical relative parts are equivalent. Further, if all non-null values of $\text{rel}(a) = (num_a, entr_a, apt_a)$ are the same as the respective attribute values of $\text{rel}(b) = (num_b, entr_b, apt_b)$, then a contains b.

We establish a link between two addresses if they are equivalent or if one address is contained in the other. The addresses that can not be linked to an address in the other street are linked to the empty address (ϵ). Each link is represented by a new tuple in the connector. The address defined by the new tuple represents the two linked addresses, and its relative part is identical to the relative part of the linked address that is more detailed. The set of new tuples, $\bar{\gamma}$, defines a new street, γ, with $\text{names}(\gamma) = \text{names}(\alpha) \cup \text{names}(\beta)$.

Definition 5.7 (Address Linking)**.** *The address linking between two streets, α and β, is the following set of new connector tuples (n_γ = names(α) $\cup$ names(β)):*

$$\begin{aligned}\bar{\gamma} \;=\; & \{(n_\gamma \circ \mathrm{rel}(a), a, b) \mid a \in \alpha, b \in \beta, \mathrm{rel}(a) \sqsubseteq \mathrm{rel}(b)\} \cup \\ & \{(n_\gamma \circ \mathrm{rel}(b), a, b) \mid a \in \alpha, b \in \beta, \mathrm{rel}(b) \sqsubseteq \mathrm{rel}(a)\} \cup \\ & \{(n_\gamma \circ \mathrm{rel}(a), a, \epsilon) \mid \nexists b \in \beta : \mathrm{rel}(b) \sqsubseteq \mathrm{rel}(a) \vee \mathrm{rel}(a) \sqsubseteq \mathrm{rel}(b)\} \cup \\ & \{(n_\gamma \circ \mathrm{rel}(b), \epsilon, b) \mid \nexists a \in \alpha : \mathrm{rel}(a) \sqsubseteq \mathrm{rel}(b) \vee \mathrm{rel}(b) \sqsubseteq \mathrm{rel}(a)\}\end{aligned}$$

Example 5.9. *Linking the addresses of the two streets α_2 and β_1 (see Figure 5.3) results in a new set of connector tuples that define the street $\gamma_1 = \{c_1, \ldots, c_5\}$. Figure 5.7 shows the address trees of the input streets (α_2 and β_1) and the new street γ_1. A root-leaf path is an address, the identifier of the address is shown below the leaf. The dashed lines represent connector tuples that link addresses of α_2 and β_1 and define addresses in the new street γ_1. a_3 and b_4 are linked to the empty address.*

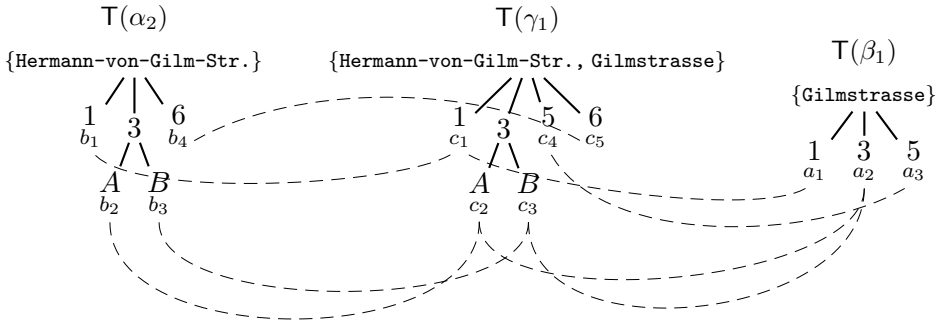


Figure 5.7: Links between the Addresses of α_2 and β_1.

5.5 Algorithms

In this section we provide algorithms for the synchronization operator, including the street distance computation, the global greedy matching, and the address linking. We prove that global greedy produces a stable matching and we discuss the complexity of our algorithms.

Synch: Algorithm 5.1 synchronizes two partitions, $\mathcal{A}$ and $\mathcal{B}$, of connector $\mathfrak{X}$. First, we compute the distance matrix, D with the distances between each pair of streets. Second, the stable street matching M is computed. Third, for each pair of matched streets $(\alpha, \beta) \in \mathsf{M}$, a new set of tuples is added to

the connector. The new tuples define a new street. The algorithm returns the connector $\mathfrak{X}$ with the new partition $\mathcal{C}$. In Section 5.5 we discuss the complexity of the synchronization and present algorithms for the street distance, the global greedy matching, and the address linking.

Algorithm 5.1: `synch`($\mathfrak{X}$,$\mathcal{A}$,$\mathcal{B}$,$\mathcal{C}$)

```
1  D[1..M, 1..N] : distance matrix;
2  foreach α_i ∈ str(A) do
3      foreach β_j ∈ str(B) do
4          D[i][j] = streetDist(α_i, β_j);
5  M = globalGreedyMatching(A, B, D);
6  foreach (α, β) ∈ M do
7      γ̄ = linkAddresses(α, β);
8      foreach c ∈ γ̄ do
9          X = X ∪ c;
10 return X;
```

Street Distance: Algorithm 5.2 computes the distance between two streets based on their address trees. The nested loop computes the minimum q-gram distance between the two street names. For the computation of the structure distance the root nodes of the address trees are substituted by dummy nodes. The name distance is weighted with ω, the structure distance with $1 - \omega$.

Algorithm 5.2: `streetDist`(α, β)

```
1  d_s ← ∞;
2  foreach s_α ∈ names(α) do
3      foreach s_β ∈ names(β) do
4          d_n ← min(d_s, dist_q(s_α, s_β));
5  d_s = dist_pq(T*(α), T*(β));
6  return sqrt(ω d_n^2 + (1 − ω) d_s^2)
```

Global Greedy Matching: Algorithm 5.3 implements the global greedy matching. The algorithm sorts the street pairs by their distance and stores them in array S. The closest street pair is matched. The respective row and column are marked in the distance matrix to prevent a street from being matched twice. The remaining streets pairs in S are matched in ascending

order of their distances if both streets in the pair are still available. This yields a stable matching.

Algorithm 5.3: `globalGreedyMatching`$(D_{M\times N}, \mathcal{A}, \mathcal{B})$

```
1  M ← ∅;
2  S[1..M * N] : array of tuples (α, β, dist);
3  for i ← 1 to M do
4      for j ← 1 to N do
5          S[(i − 1) * N + j] ← (α_i, β_j, D[i, j]);
6  S ← sort(S) by dist;
7  seen-row[1..M], seen-col[1..N] : boolean arrays initialized with false;
8  s ← 1;
9  while |M| < min(M, N) do
10     (dist, α_i, β_j) ← S[s];
11     if not seen-row[i] and not seen-col[j] then
12         M ← M ∪ {(α_i, β_j)};
13         seen-row[i] ← true; seen-col[j] ← true;
14     s++;
15 return M;
```

Theorem 5.1. *The global greedy matching (Algorithm 5.3) is stable.*

Proof. Let M'_k be the matching produced by Algorithm 5.3 after the k-th execution of line 12, thus $\mathsf{M}'_0 = \emptyset$ and $\mathsf{M}'_n = \mathsf{M}$ ($n = |\mathsf{M}|$). We substitute M by M'_k in Equation (5.1) and proof it by induction. "Equation (5.1) holds for $k = 1$": The algorithm chooses the closest street pair amongst all possible pairs. "If (5.1) holds for M'_k, then it also holds for M'_{k+1}, $k < n$": No pair $(\alpha, \beta) \notin \mathsf{M}'_{k+1}$ satisfies the right-hand condition (denoted as C). Let (u, v) be the new pair in M'_{k+1}, i.e., $\mathsf{M}'_{k+1} \setminus \mathsf{M}'_k = \{(u, v)\}$. We distinguish:

1. $u \neq \alpha$ and $v \neq \beta$: C is false as (5.1) holds for M'_k and neither u nor v appear in C.

2. $u = \alpha$ and $v \neq \beta$: The algorithm matches the closest pair of unmatched streets first. Thus, if β is unmatched in M'_k, $\forall(u, y) \in \mathsf{M}'_{k+1}$: $\text{dist}(u, y) \leq \text{dist}(u, \beta)$; if β is already matched, $\forall(x, \beta) \in M_{k+1}$: $\text{dist}(x, \beta) \leq \text{dist}(u, \beta)$. In both cases C does not hold.

3. $u \neq \alpha$ and $v = \beta$: Analog to previous case. □

Address Linking: Algorithm 5.4 links the addresses of the two streets α and β, and produces the new set of connector tuples $\bar{\gamma}$. Checking equivalence and containment for all pairs of addresses leads to a quadratic runtime in the size of the input streets. We define an order on residential addresses and we present an efficient merge-based algorithm to link the addresses of two streets. The algorithm sorts the addresses of the input streets, and i and j point to the current addresses (initially the first address) of the sorted arrays $a[]$ and $b[]$, respectively. If one of the current addresses is contained in the other (equivalence is a special case of containment), a link is established and a new tuple for $\bar{\gamma}$ is produced. The pointer of the more detailed address is moved on. If none of the current addresses is contained in the other address, or if one of the pointers reaches the end of the array, links to the empty address are produced.

Definition 5.8 (Order of Residential Addresses)**.** *Let a and b be two residential addresses with the relative parts* $\mathrm{rel}(a) = (num_a, entr_a, apt_a)$ *and* $\mathrm{rel}(b) = (num_b, entr_b, apt_b)$*, respectively. We define*

$$\begin{aligned} a > b \quad \Leftrightarrow \quad & num_a > num_b \\ & or\ (num_a = num_b \wedge entr_a > entr_b) \\ & or\ (num_a = num_b \wedge entr_a = entr_b \wedge apt_a > apt_b), \end{aligned}$$

where $num_{a/b}$, $entr_{a/b}$, and $apt_{a/b}$ are ordered lexicographically.

Complexity: For our complexity analysis we assume the synchronization of two partitions with N streets, each street has n addresses and c names of constant length. The name distance is computed in $O(c^2)$ time and constant space, and the *pq*-gram distance between two address trees has runtime $O(n \log n)$ and needs $O(n)$ space [3]. The global greedy matching algorithm requires $O(N^2)$ space (the size of the distance matrix) and runs in $O(N^2 \log N)$ time (sorting the distances). The address linking sorts the addresses of the streets in $O(n \log n)$ time and runs in $O(n)$ space. The overall time complexity of the synchronization (Algorithm 5.1) is $O(N^2(c^2 + n \log n + \log N))$, the space complexity is $O(N^2 + n)$.

5.6 Experiments

We experimentally evaluate the accuracy of our approach on real world residential addresses from the registration office (*reg*, Italian street names, 43K addresses), the electricity company (*elec*, German, 45K addresses), and the cen-

Algorithm 5.4: `linkAddresses`(α, β)

```
1  a[1...|α|] ← sort(α) by (num, entr, apt);
2  b[1...|β|] ← sort(β) by (num, entr, apt);
3  i ← 1; j ← 1; γ̄ ← ∅;
4  n_γ ← names(α) ∪ names(β);
5  while (i ≤ |α|) ∧ (j ≤ |β|) do
6      if a[i] ⊑ b[j] then
7          while a[i] ⊑ b[j] do { γ̄ ← γ̄ ∪ (n_γ ∘ rel(a[i]), a[i], b[j]); i++; }
8          j++;
9      else if a[i] ⊒ b[j] then
10         while a[i] ⊒ b[j] do { γ̄ ← γ̄ ∪ (n_γ ∘ rel(b[j]), a[i], b[j]); j++; }
11         i++;
12     else if a[i] < b[j] then { γ̄ ← γ̄ ∪ (n_γ ∘ rel(a[i]), a[i], ϵ); i++; }
13     else if a[i] > b[j] then { γ̄ ← γ̄ ∪ (n_γ ∘ rel(b[i]), ϵ, b[j]); j++; }
14 while (i ≤ |α|) do { γ̄ ← γ̄ ∪ (n_γ ∘ rel(a[i]), a[i], ϵ); i++; }
15 while (j ≤ |β|) do { γ̄ ← γ̄ ∪ (n_γ ∘ rel(b[j]), ϵ, b[j]); j++; }
16 return γ̄
```

sus database (*cens*, German, 11K addresses) of the municipality of Bolzano-Bozen. We load the data into the connector such that each database corresponds to a partition. We synchronize the partitions pairwise and verify the street matches. The runtimes for synchronizing two partitions are shown in Table 5.1 (AMD 2.6 GHz Processor, 16GB RAM).

$\mathcal{A}$	$\mathcal{B}$	name dist [s]	structure dist [s]	overall [s]
elec	*cens*	1.7	6.1	9.2
elec	*reg*	1.1	11.3	13.6
reg	*cens*	1.0	6.5	8.8

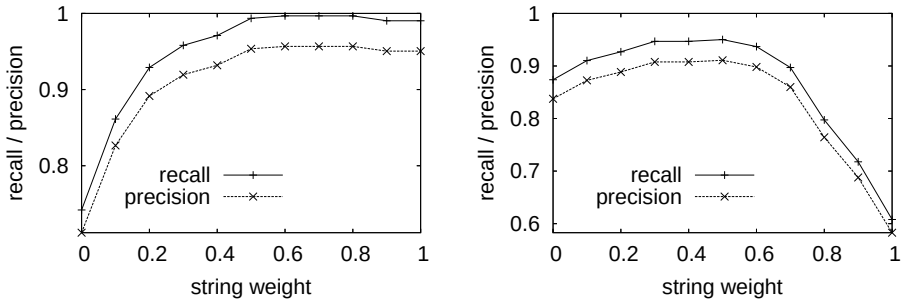
Table 5.1: Runtimes for Synchronizing two Partitions.

In the following subsections we show that the combination of name and structure increase the quality of street distance. We alse evaluate the matching accuracy of the global greedy matching algorithm and find that it consistently outperforms both the fixed threshold and the local greedy approach.

5.6.1 Name and Structure Distance

We compute *precision* (correctly found matches to total number of computed matches) and *recall* (correctly found matches to total number of correct

matches) for different weights w for name and structure distance. If $w = 0$, only the structure of the address trees is considered, if $w = 1$, only the street names are considered. The results are shown in Figure 5.8. Pure street name matching ($w = 1$) gives good results if both databases have German street names (Figure 5.8(a)), but it fails if the street names have different languages (Figures 5.8(b) and 5.8(c)). The combination of name and structure improves the results for all datasets. For $w = 0.5$ (equal weight for name and structure) we find more than 95% of the matches in all data sets, and more than 90% of our matches are correct.



(a) *elec* (Ge) ↔ *cens* (Ge)

(b) *elec* (Ge) ↔ *reg* (It)

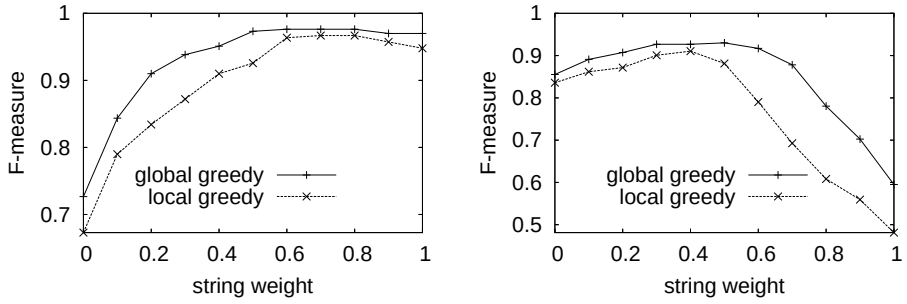
(c) *reg* (It) ↔ *cens* (Ge)

Figure 5.8: Matching Accuracy for Different Weights and Databases.

5.6.2 Global Greedy vs. Local Greedy

This section compares the global greedy matching algorithm with the local greedy algorithm (see Section 5.4.3). As in the previous experiment we match streets with different weights. In order to compare local greedy with our global greedy algorithm we compute the F-measure. The F-measure, $F = \frac{2pr}{p+r}$, is the harmonic mean of precision, p, and recall, r, and it is a well-known performance

measure in the information retrieval literature [50]. Figure 5.9(a) shows the results for matching the streets of *elec* and *cens*, in Figure 5.9(b) we match *elec* and *reg*. Matching *reg* and *cens* yields similar results. Global greedy yields better matches for all settings that we have tested.



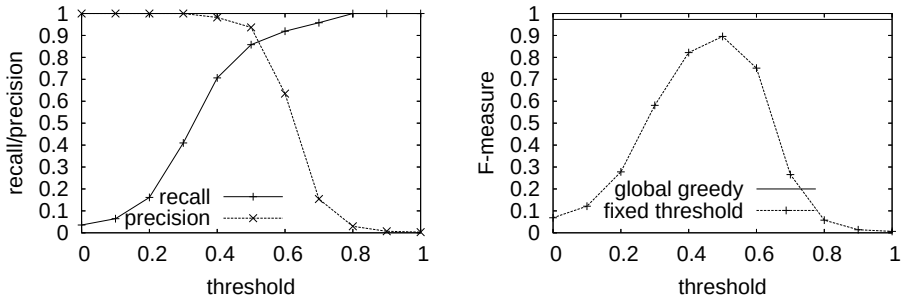
(a) *elec* (Ge) ↔ *cens* (Ge) (b) *elec* (Ge) ↔ *reg* (It)

Figure 5.9: Global Greedy vs. Local Greedy

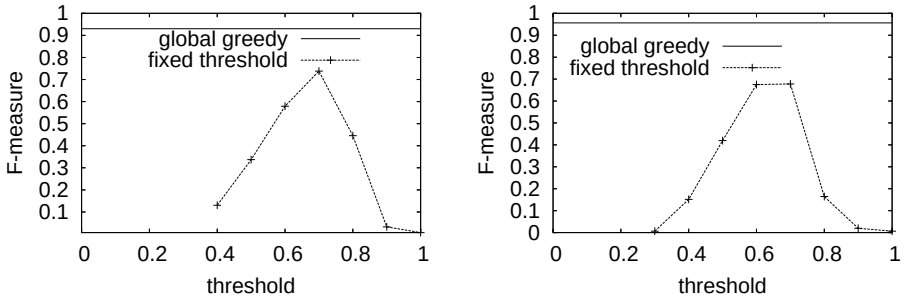
5.6.3 Global Greedy vs. Fixed Threshold

This section compares the global greedy matching algorithm with the fixed threshold approach. The fixed threshold approach matches all streets that are within a given distance (see Section 5.4.3). Figure 5.10(a) shows precision and recall for increasing threshold values (*elec* ↔ *cens*, string weight $w = 0.5$). The precision is high for small thresholds (all matches are correct), but the recall is very low (only few matches were found). As the threshold increases, more matches are found, but also the number of incorrect matches increases. Very high thresholds compute the cross product between all streets, the recall is 100%, the precision decreases to almost zero.

In Figures 5.10(b)- 5.10(d) we compare the matching accuracy of the global greedy algorithm with the fixed-threshold approach. The global greedy algorithm outperforms the fixed threshold approach for all thresholds. The results for the global greedy algorithm are independent of the threshold. The missing values in Figures 5.10(c) and 5.10(d) indicate thresholds for which both precision and recall are zero, i.e., no streets could be matched within the given threshold.



(a) Precision and Recall ($elec \leftrightarrow cens$) (b) F-Measure ($elec \leftrightarrow cens$, $w = 0.5$)



(c) F-Measure ($elec \leftrightarrow reg$, $w = 0.5$) (d) F-Measure ($reg \leftrightarrow cens$, $w = 0.5$)

Figure 5.10: Global Greedy vs. Fixed Threshold.

5.7 Related Work

Residential addresses appear in many applications, and commercial tools that deal with the synchronization of residential addresses have been developed. Customer Data Integration tools often include residential address integration. Many tools, for example DQaddress (caatoosee ag)[1] rely on string matching techniques and find typos and small spelling variations. They often include rules for common abbreviations. They can not deal with renamed streets or streets in different languages. Typical address applications in the United States use a standardized set of residential addresses and correct the input address according to the reference set. AbiliTec® (by ACXIOM®)[2], for example, relies on an extensive repository of historical name and address information. The repository stores associations between current an previous addresses, real names and nicknames, maiden names, married names and multiple variations of business names. In our case no such database is available, and we can not rely on a standardized set of addresses. Instead we build links that equally respect all participating addresses

The concept of an address tree is introduced in Augsten et al. [2], while a later work [3] defines the *pq*-gram distance and matches address trees. The similarity of street names is not considered. We show in our experiments that we can significantly improve the matching accuracy by considering the name distance in addition to the structure distance between address trees. Our previous matching algorithm [3] matches two streets if they are mutual and strict nearest neighbors. The nearest neighbor function is not symmetric, and some streets may not have a mutual nearest neighbor. Also, a street may have more than one nearest neighbors at the same distance. All these streets remain unmatched and decrease the recall of the method.

q-Grams were introduced by Ukkonen [49] as a lower bound for the more expensive string edit distance [38, 40]. Gravano et al. [25] show that *q*-grams can be implemented efficiently in a relational database. Our approach is independent of the choice of a specific string distance.

Matching data items based on the distance between them is a well known problem in data integration. Gravano et al. [25] define the approximate string join, approximate XML joins are introduced by Guha et al. [26]. Both approaches use a fixed distance threshold and match all pairs of items that are within the threshold. Chaudhuri et al. [8] point out that fixed thresholds lead to poor matching accuracy as items that should match may be more distant than items that should not match. They introduce a variable threshold for a

[1] `http://www.caatoosee.com`
[2] `http://www.acxiom.com`

duplicate detection scenario and define two criterions for duplicates, the compact set criterion (duplicates are closer to each other than to non-duplicate items) and the sparse neighborhood criterion (the local neighborhood of duplicate items is sparse). Both fixed and variable thresholds possibly match a single item to multiple other items which is undesirable in our setting.

The street matching can be modeled as a bipartite weighted graph matching problem (also known as assignment problem). The streets form the disjoint node sets of the bipartite graph, the distances between the streets are the weighted edges. The goal is to compute the minimum-weighted matching between the N nodes of the graph. The Hungarian algorithm by Kuhn [36] runs in $O(N^2|V|)$ time, which is $O(N^4)$ in our case of a dense graph with $|V| = N^2$ edges. Edmonds and Karp [18] present an algorithm based on Dijkstra's shortest paths [17] that runs in $O(N^3)$ time if implemented with the Fibonacci heap [21]. For the more general maximum-flow problem Goldberg and Tarjan [24] propose an $O(N^3)$ time algorithm. All these algorithms globally minimize the sum of the distances in a matching, but they can not guarantee a stable matching. Computing stable matchings is known as the stable marriage problem [28]: Given a population of N men and N women, each man strictly ranks each woman according to his preferences for a marriage partner, and vice versa. Gale and Shapley [22] propose a $O(N^2)$ time algorithm that computes a stable matching between men and women. The Gale-Shapley algorithm is not commutative, and the solution is optimized either for men or for women, depending on the order of the parameters. For the respective other part the worst case solution is produced. *Egalitary* stable marriage algorithms that fix this problem have been proposed [19, 29], the most efficient one runs in $O(N^3)$ time. We can take advantage of the distances that globally rank the matches and produce a commutative stable matching in $O(N^2 \log N)$ time and $O(N^2)$ space.

5.8 Conclusion

We have presented the address connector which links residential addresses of different databases that refer to the same location. The address connector does not need an authoritative reference, but it builds a new reference that equally respect all participating addresses. The core of the connector is the synchronization operator which can deal with non-matching (even completely unrelated) street names and correctly links residential addresses with different granularity. The address connector has been successfully tested in the context of the Municipality of Bolzano-Bozen. In our experiments with real world data

from the public administration we show the effectiveness and the efficiency of our approach.

The synchronization operator implements a new, context-aware street distance that considers, in addition to the street name, also the hierarchical structure that is defined by the addresses of a street. The distances between all street pairs are stored in a distance matrix. We propose a new algorithm that matches streets based on the distance matrix. Our global greedy algorithm matches each street to at most one other street, the result is independent of the matching order, and we proof that the matches are stable. We define the concept of address containment that allows us to link the addresses of two streets correctly, even if they are stored with different granularity.

Future work will extend our solution to other applications. The combination of string and tree distances defined by the address tree distance is useful when hierarchical data include string-valued nodes that (almost) identify objects. As an example consider XML data that store publications. Two publications are similar if both their title and the XML structure (defined by authors, year of publication, etc.) are similar. Our global greedy matching algorithm solves the problem of matching items based on a distance matrix, a typical problem in data integration scenarios. The key properties of the global greedy matching (independence of the matching order, one-to-one matches, and the stability of the matching) sets it apart from previous approaches in data integration. The concept of containment extends to other kinds of hierarchical data that are stored with different granularity. Our approach to link data in the connector instead of correcting the databases is useful in applications where correcting data is not possible (e.g., due to read only access or different granularity).

Chapter 6

Conclusions and Future Work

In this thesis we proposed *pq*-grams to approximately match hierarchical data. We introduce the *pq*-gram index that offers efficient approximate lookups and joins of hierarchical data in a relational database. The incremental update of the index is independent of the tree size. We introduce windowed *pq*-grams for the approximate matching of data-centric XML and develop an efficient approximate join algorithm for windowed *pq*-grams. The connector is our system for the approximate matching of residential addresses based on the address tree.

The *pq*-gram index represents a tree by its *pq*-grams and is the basis for computing the distance between trees. We have developed an efficient algorithm that computes the *pq*-grams in linear time and space. The algorithm splits the trees into *pq*-grams, serializes the *pq*-grams, and hashes them to string values of a fixed length. Our relational implementation requires only a single scan of the tree data and stores the *pq*-gram index as a relation. We give an efficient approximate join algorithm based on the *pq*-gram index that is implemented as a query in a relational database. Most joins based on distance measures, such as the edit distance, must evaluate the distance between every pair of input trees. There is no effective way to sort trees or hash them into buckets, and an expensive nested-loop join must be applied. We reduce the approximate join to an equality join on strings that takes advantage of well known join optimization techniques, for example, the sort-merge join.

We provide incremental updates for the *pq*-gram index in response to structure and value changes in the indexed trees. We formally proof that the index can be updated based on the resulting tree and the log of edit operations without reconstructing intermediate tree versions. 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 incremental update is independent of the tree size and scales to large numbers of edit operations. The experimental results validate the approach for the DBLP dataset and logs with several thousand edit operations.

Data-centric XML is represented as unordered trees. The tree edit distance between unordered trees is NP-complete. We introduce *windowed pq*-grams for unordered trees that are computed in a three-step process: sorting the tree, extending the tree with dummy nodes, and computing the *pq*-grams on the extended tree using a window mechanism. The resulting *pq*-grams consist of a stem and a base. The stems are invariant to order, the main challenge is to compute order-invariant bases. Our bases enjoy the following important properties: all base-nodes have equal frequency, the Jaccard distance between sibling sets is preserved, and node moves to other parents are detected. The *pq*-gram distance computed on the windowed *pq*-grams of two sorted trees approximates the unordered tree edit distance between these trees. The windowed *pq*-grams are produced in linear time and space, and the approximate join based on windowed *pq*-grams scales to large data sets. To the best of our knowledge, this is the first work to address the problem of approximately joining XML data without relying on the document order. Extensive experiments on both synthetic and real world data confirm the analytic results and suggest that our technique is both useful and scalable.

The connector solves the problem of matching residential addresses that are stored with different granularity and have different or unrelated street names. The connector can produce high quality matches by relying on the address tree for the distance computation between streets. Our global greedy matching algorithm forms street pairs based on a distance matrix. No threshold parameter is required, the matching quality is independent of the matching order, and the matching is stable. In our experiments we evaluate recall and precision of the global greedy matching algorithm and show a significant accuracy improvement over local greedy and the fixed-threshold approach. We introduce the concept of address containment and link residential addresses even if they are stored with different granularity. The address connector has been successfully tested in the context of the Municipality of Bolzano.

Future Work Previous work, including ours, has used the tree edit distance as a reference for the similarity between labeled trees. While the tree edit distance is intuitive, it has never been shown that it is a good reference. We show that it produces rather non-intuitive results in some situation, for example, when non-leaf nodes are deleted. It would be interesting future work to further investigate this issue. As an alternative reference, a set of properties could be defined such that different distance functions can be positioned with

respect to these properties. As an example consider the sensitivity to structure change. The empirical study in this thesis suggests that the *pq*-gram distance emphasizes structure changes more than the tree edit distance. The edges define the structure of a tree. Parallel to the node edit distance an edge edit distance could be defined to quantify structure changes. For example, the deletion of a non-leaf node is a single node edit operation, although the deletion may substantially change the tree structure. The edge edit distance captures this change, as the edges of all children of the deleted node must be connected to a new parent and are affected.

We further plan to introduce weights for *pq*-grams. Currently all *pq*-grams have the same weight, and the *pq*-gram distance is (inversely) proportional to the cardinality of the profile intersection. All *pq*-grams that match between the profiles decrease the *pq*-gram distance by the same amount. When different weights are assigned to the *pq*-grams, some matches have more impact on the distance than others. The distance algorithm must be adapted to match *pq*-grams with small weights first in order to get the *minimum* *pq*-gram distance. The sensitivity of unweighted *pq*-grams to structure change is controlled by the parameter p. We will weight *pq*-grams depending on their structural importance (for example, the fanout of the anchor node), which will allow us to fine-tune their sensitivity to structure changes. We think that appropriate weights will make the *pq*-gram distance a lower bound of the tree edit distance.

Bibliography

[1] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 141–152, San Jose, California, 2002. IEEE Computer Science Press.

[2] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. Reducing the integration of public administration databases to approximate tree matching. In Roland Traunmüller, editor, *Electronic Government – Third International Conference*, Lecture Notes in Computer Science 3183, pages 102–107, Zaragoza, Spain, August 2004.

[3] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. Approximate matching of hierarchical data using *pq*-grams. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 301–312, Trondheim, Norway, September 2005. ACM.

[4] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. An incrementally maintainable index for approximate lookups in hierarchical data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 247–258, Seoul, Korea, September 2006. ACM.

[5] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 310–321, Madison, Wisconsin, June 2002. ACM Press.

[6] Joe Celko. Trees, databases and SQL. *Database Programming and Design*, 7(10):48–57, September 1994.

[7] Joe Celko. *Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann Publishers Inc., 2004.

[8] Surajit Chaudhuri, Venkatesh Ganti, and Rajeev Motwani. Robust identification of fuzzy duplicates. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 865–876, Tokyo, Japan, April 2005. IEEE Computer Science Press.

[9] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tucson, Arizona, United States, May 1997. ACM Press.

[10] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montreal, Canada, June 1996. ACM Press.

[11] Weimin Chen. New algorithm for ordered tree-to-tree correction problem. *Journal of Algorithms*, 40(2):135–158, August 2001.

[12] Grégory Cobéna, Serge Abiteboul, and Amélie Marian. Detecting changes in XML documents. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 41–52, San Jose, California, 2002. IEEE Computer Science Press.

[13] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 341–350, Roma, Italy, September 2001. Morgan Kaufmann Publishers Inc.

[14] Theodore Dalamagas, Tao Cheng, Klaas-Jan Winkel, and Timos Sellis. A methodology for clustering XML documents by structure. *Information Systems*, 31(3):187–228, May 2006.

[15] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 623–634, San Diego, California, June 2003. ACM Press.

[16] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP 2007)*, Wroclaw, Poland, 2007.

[17] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[18] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, April 1972.

[19] Tomás Feder. A new fixed point approach for stable networks and stable marriages. *Journal of Computer and System Sciences*, 45(2):233–284, October 1992.

[20] Sergio Flesca, Giuseppe Manco, Elio Masciari, Luigi Pontieri, and Andrea Pugliese. Fast detection of XML structural similarity. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(2):160–175, February 2005.

[21] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, July 1987.

[22] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, January 1962.

[23] Minos Garofalakis and Amit Kumar. XML stream processing using tree-edit distance embeddings. *ACM Transactions on Database Systems*, 30(1):279–332, 2005.

[24] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, October 1988.

[25] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 491–500, Roma, Italy, September 2001. Morgan Kaufmann Publishers Inc.

[26] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate XML joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 287–298, Madison, Wisconsin, 2002. ACM Press.

[27] Sudipto Guha, Nick Koudas, Divesh Srivastava, and Ting Yu. Index-based approximate XML joins. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 708–710, Bangalore, India, March 2003. IEEE Computer Science Press.

[28] Dan Gusfield and Robert W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. The MIT Press, August 1989.

[29] Robert W. Irving, Paul Leather, and Dan Gusfield. An efficient algorithm for the "optimal" stable marriage. *Journal of the ACM (JACM)*, 34(3):532–543, July 1987.

[30] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-tree: Indexing XML data for efficient structural joins. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 253–263, Bangalore, India, March 2003. IEEE Computer Science Press.

[31] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 273–284, Berlin, Germany, September 2003. Morgan Kaufmann Publishers Inc.

[32] Tao Jiang, Lusheng Wang, and Kaizhong Zhang. Alignment of trees—an alternative to tree edit. *Theoretical Computer Science*, 143(1):137–148, July 1995.

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

[34] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Pradeep Shenoy. Updates for structure indexes. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 239–250, Hong Kong, China, August 2002. Morgan Kaufmann Publishers Inc.

[35] Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, pages 91–102, Venice, Italy, 1998. Springer.

[36] Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[37] Kyong-Ho Lee, Yoon-Chul Choy, and Sung-Bae Cho. An efficient algorithm to compute differences between structured documents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(8):965–979, August 2004.

[38] Vladimir I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[39] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 361–370, Roma, Italy, September 2001. Morgan Kaufmann Publishers Inc.

[40] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[41] Andrew Nierman and H. V. Jagadish. Evaluating structural similarity in XML documents. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, USA, June 2002.

[42] Neoklis Polyzotis, Minos Garofalakis, and Yannis Ioannidis. Approximate XML query answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 263–274, Paris, France, June 2004. ACM Press.

[43] Sven Puhlmann, Melanie Weis, and Felix Naumann. XML duplicate detection using sorted neighborhoods. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, volume 3896 of *Lecture Notes in Computer Science*, Munich, Germany, March 2006. Springer.

[44] Chen Qun, Andrew Lim, and Kian Win Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 134–144, San Diego, California, USA, June 2003. ACM Press.

[45] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 360–371, Tokyo, Japan, April 2005. IEEE Computer Science Press.

[46] Stanley M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

[47] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, July 1979.

[48] Eiichi Tanaka and Keiko Tanaka. The tree-to-tree editing problem. *Int. Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 2(2):221–240, 1988.

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

[50] C. J. van Rijsbergen. *Information Retrieval*, chapter 3. Butterworth-Heinemann, 2nd edition, March 1979.

[51] Yuan Wang, David J. DeWitt, and Jin-yi Cai. X-Diff: An effective change detection algorithm for XML documents. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 519–530, Bangalore, India, March 2003. IEEE Computer Science Press.

[52] Melanie Weis and Felix Naumann. DogmatiX tracks down duplicates in XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 431–442, Baltimore, Maryland, USA, June 2005. ACM Press.

[53] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. Similarity evaluation on tree-structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 754–765, Baltimore, Maryland, USA, June 2005. ACM Press.

[54] Wuu Yang. Identifying syntactic differences between two programs. *Software—Practice & Experience*, 21(7):739–755, July 1991.

[55] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 425–436, Santa Barabara, California, 2001.

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

[57] Kaizhong Zhang, Richard Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133–139, 1992.