Department of Informatics, University of Zürich

BSc Thesis

Temporal Filtering to Improve Temporal Duplicate Detection

Gionata Genazzi

of Locarno, Switzerland

Matrikelnummer: 09-742-669

Email: gionata.genazzi@uzh.ch

July 24, 2015

supervised by Prof. Dr. Michael Böhlen and Pei Li

University of Zurich[UZH]

Department of Informatics

Dedicated to my family

Acknowledgements

I would like to thank Prof. Dr. Michael Böhlen for the opportunity to write my bachelor thesis at the Database Technology Group of the University of Zürich. I thank Pei Li for her help and feedback regarding my work. Very special thanks go to my family, my friends, and my girlfriend.

Abstract

Duplicate detection studies the problem of identifying records in a given data set that refer to the same real-world entity. A quantitative way of solving duplicate detection is to perform a *similarity join*. A large collection of algorithms performs the similarity join using the *filter-verification framework*. Algorithms of this type exploit techniques as *prefix filtering*, *positional filtering*, and *suffix filtering* to perform the join in an efficient way. However, implementations of these algorithms ignore *temporal information* of records, which could be exploited to enhance duplicate detection.

In this thesis, we refine prefix filtering, positional filtering, and suffix filtering techniques in order to perform similarity joins utilizing also temporal information of records. Specifically, we propose three algorithms that can perform *temporal similarity joins* with a *temporal Jaccard similarity* threshold. Experimental results show that these algorithms can improve considerably the performance of exact temporal similarity joins with temporal Jaccard when compared to the brute force approach.

Zusammenfassung

Duplikaterkennung (*duplicate detection*) ist das Finden mehrerer Repräsentationen desselben Realweltobjekts. Die Ausführung einer *similarity join* stellt eine quantitative Lösung von Duplikaterkennungsprobleme dar. Eine breite Kategorie von similarity join Algorithmen verwenden das sogenannte *filter-verification framework*; diese Algorithmen nutzen Techniken wie *prefix filtering*, *positional filtering* und *suffix filtering* mit dem Ziel similarity join effizient auszuführen. Jedoch, diese Techniken berücksichtigen nicht *temporale Information* von Daten. Noch, die Verwendung von temporaler Information kann Duplikaterkennung verbessern.

In dieser Arbeit, wir verfeinern prefix filtering, positional filtering und suffix filtering Techniken, sodass sie während similarity joins temporaler Information von Daten ausnutzen können. Wir präsentieren drei Algorithmen die *temporal similarity joins* Probleme lösen. Die an realen Datensätzen durchgeführten Experimente zeigen, dass die drei vorgeschlagenen Algorithmen die Effizienz von temporal similarity joins im Vergleich zum brute force Vorgehen erheblich verbessern können.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Duplicate detection is the problem of identifying records in a given data set that refer to the same real-world entity. The problem has many names across different research areas; some common synonyms are *record matching*, *record linkage*, *data deduplication* and *near duplicate detection*. In particular, it arises in data integration scenarios, when data coming from different sources lacks a common identifier.

Many methods for solving the duplicate detection problem are quantitative-based and use a *similarity function*. Given two records x and y, a similarity function returns a value, named similarity value, expressing the similarity degree between x and y. The process of finding all pairs in a given data set with a similarity value higher than a given threshold is known as *similarity join*.

A simple and raw method to perform the similarity join is based on classical brute force approach, which compares every pair of records in the data set. An algorithm of this type is extremely expensive and attains a quadratic time complexity. Performing similarity join in an efficient and scalable way is a very active field of database research. A large collection of algorithms uses the "filter-verfication" framework to perform the similarity join. This consist in two steps: in the filter step (1), an effective filtering algorithm should prune large numbers of non-similar pairs and generate a set of candidate-pairs that must be a superset of the final result; in the verification step (2), the candidate pairs are verified against the chosen similarity threshold to obtain the final result. Techniques for filtering many pairs in step (1) include "prefix filtering" [2] [1], "positional filtering" [16] and "suffix filtering" [16].

In addition, study on temporal data sets brings new challenges to duplicate detection. In particular, Li et al. [9] and Chiang et al. [3] showed that techniques that make use of temporal information can enhance duplicate detection. However, algorithms that use temporal information are usually computationally expensive.

This work focuses on extending prefix filtering, positional filtering and suffix filtering techniques in order to perform similarity joins using also temporal information of records. We make the following contributions:

- We extend prefix filtering, positional filtering and suffix filtering techniques to handle the temporal Jaccard metric. This metric considers also temporal information of records and requires a temporal model to compute temporal weights for attributes. Specifically, we propose three algorithm: TPREFIX, an algorithm that exploits prefix filtering for computing similarity joins with temporal Jaccard; TPOSITIONAL, which extends TPREFIX with positional filtering; and TSUFFIX, which further extends TPOSITIONAL with suffix filtering.

- We show that these algorithms improve considerably the performance of exact similarity joins with temporal Jaccard when compared to the brute force algorithm. In particular,

TSUFFIX proved to be the fastest among the three in our experiments on real-world data sets, followed by TPOSITIONAL which proved to be only modestly slower. Finally, TPREFIX proved to be the slowest.

- Marginally, we propose a variation of the temporal model based on recurrence rates devised by Chiang et al. [3]. Our variation enables to calculate similarity between two records using recurrence rates but no clustering is required.

The remaining parts of the thesis are organized as follows. The next chapter discusses related work. Then, chapter 3 presents the problem definition and some preliminaries. In particular, we explain the similarity metrics used in the thesis and the temporal model that we use. In chapter 4 we illustrate the refinement of the prefix filtering, positional filtering, and suffix filtering techniques. In this chapter we also show our algorithms to perform similarity joins with a temporal Jaccard threshold. Finally, our experimental results are discussed in chapter 5 and conclusions in chapter 6.

2 Related work

2.1 Duplicate detection

The goal of duplicate detection is to link records in the same or different datasets that refer to the same real-world object. The problem has been studied in many research fields – the most important are statistics, database and artificial intelligence – and has developed different names across them. The statistics community was the first to analyze the problem more than 50 years ago under the name of *record linkage* or *record matching*. Other common synonyms are *merge-purge*, *data deduplication* and *near duplicate detection*. Elmagarmid et al. [7] provide a recent survey on this topic.

Duplicate detection contributes to reach a high information quality; a fundamental requirement in today's IT-based society. In particular, it is a crucial activity when dealing with data heterogeneity, which usually originates from integration processes of different data sources that lack global identifiers.

Numerous techniques for duplicate detection have been developed in the past. A first differentiation can be made between *single field* and *multiple fields* records matching techniques. However, this is not a clear-cut separation, as many single field matching techniques are often used as a basis for multiple fields techniques.

Duplicate detection with single field records is usually done with the aid of a *similarity function* (also called *similarity metric*). Given two records, a similarity function returns a value indicating the degree of similarity between them. Similarity functions for matching string data fields can be divided into *character-based*, *token-based* and *phonetic*. A well-known character-based similarity function is *edit-distance*, which consists in counting the number of edit operations needed to transform a string into the other. Token-based similarity functions first need to transform strings into sets of tokens and use a set-based similarity function (e.g. *Jaccard*, *overlap* or *cosine*) to compute the similarity. A commonly used phonetic similarity measure is *Soundex*. In addition, some techniques for detecting similarity of numeric data have been also researched.

Duplicate detection with records consisting of multiple fields is more complicated. A common approach relies on training data to learn how to match the records. Techniques developed with this approach are *supervised and semisupervised machine learning*, *active-learning* and *probabilistic matching*. Other approaches include *distance-based techniques*, which roughly consist in using a similarity function that do not need tuning (e.g., Jaccard), and *rule-based techniques*, which uses rules developed by domain-experts to classify records. In addition, some recent works use the temporal information of records to increase accuracy; this research area is reviewed in more detail in section 2.3.

Focusing on efficiency, methods have been proposed to reduce the number of record com-

parisons needed during duplicate detection processes. These include *blocking*, *clustering* and *set joins*. Methods to improve the efficiency of a single record comparison have been also researched. Methods to improve the efficiency of duplicate detection in the context of records with a single string field are examined closer in the next section.

The reader is referred to [7] for further references to the techniques cited in this section.

2.2 String similarity join

Given a similarity function, the process of finding all pairs in the same or different datasets which have a similarity value higher than a given threshold is known as *similarity join*. Similarity join is principally studied in its application on datasets composed by string data records (single field or multiple fields tokenized and generalized). For this reason, the most common similarity functions used in similarity join algorithms are *character-based* and *token-based* (see section 2.1). Jiang et al. [8] provide a survey on a wide spectrum of string similarity join algorithms and propose a classification schema.

Signature-based algorithms for computing similarity joins are the largest category and include the algorithms that we extend in this thesis. Signature-based algorithms are based on the *filter-verification framework*, which consists in two steps. In the filter step (1), an effective filtering algorithm should prune large numbers of non-similar pairs and generate a set of candidate-pairs that must be a superset of the final result. In the verification step (2), the candidate pairs are verified against the chosen similarity metric to obtain the final result. Common filtering algorithms can be classified into four main branches: *count filtering* (GRAMCOUNT, LISTMERGER), *partition filtering* (PARTENUM, PASSJOIN), *neighborhood filtering* (FASTSS) and *prefix filtering*. The first algorithm to use prefix filtering was SSJOIN [2], which focuses on enabling similarity joins inside DBMS. The technique is then extended by the ALL-PAIRS algorithm [1], which provides a prefix filtering-based framework. Xiao et al. [16] introduced PPJOIN, which applies *positional filtering* and *suffix filtering* principles to further reduce the number of set joins operations needed when dealing with Jaccard, overlap or cosine similarity functions. This is the algorithm we extend with temporal dimension in the thesis. Further extensions of the prefix filtering technique are EDJOIN [15], QCHUNK [11], VCHUNG [14] and ADAPTJOIN [13].

Some non-signature algorithms that directly compute the final result of the join were also devised. The most important are *tree-based algorithms* (M-TREE, TRIEJOIN) and *disk algorithms* (LISTMERGER). In addition, algorithms based on LSH (locality-sensitive hashing) can perform *approximate similarity join*. In this case, the result is not guaranteed to contain all the similar pairs.

Other related research areas are *top-k similarity join* and *similarity search*. Top-k similarity join refers to the problem of finding the top-k similarity pairs with the largest similarity in a given dataset, while similarity search deals with finding the records whose similarity with the inserted query exceeds a given threshold. The reader is referred to [8, pp. 627–630] for a condensed survey on this topic and for further references to the algorithms cited in this section.

2.3 Temporal duplicate detection

Several *temporal data models* [10] and *temporal knowledge discovery paradigms* [12] have been developed. However, these works are not focused on duplicate detection. Yakout et al. [17] proposed *behavior based linkage*, which uses periodical behavior patterns of each single entity for linking pairs of records. The behavior patterns of an entity are learned from transaction logs recording the "actions" of the entity with respect to a given data source. The patterns of two entities are then merged to check if a well recognized behavior (indicating that the two entities are the same) emerges. Cohen and Strauss [5] and Cormode et al. [6] proposed the notion of *time decay* in the context of data warehouses and streaming, with the goal of reducing the effect of older tuples on data analysis. Cohen and Strauss introduced *backward decay*, which measures time difference backward from the latest time, while Cormode et al. devised *forward decay*, which measures time difference forward from a fixed landmark.

Li et al. [9] were the first to apply decay with the goal of duplicate detection enhancement. They introduced the notion of *agreement decay*, i.e. the probability that two entities share a common attribute in a given time interval, and *disagreement decay*, i.e. the probability that an entity's attribute will change after a given time interval. On these basis, their model assigns different weights to the different attributes for similarity calculation. In addition, Li et al. developed specific temporal clustering methods to further improve the process and showed that their model has higher accuracy than models that do not use temporal information. Very recently, Chiang et al. [3] proposed a temporal model based on the concept of *entity mutation*. The model allows for a better prediction of attributes' evolution, because it considers also the possibility that a value could reappear over time. In addition, during the clustering phase, it takes into account the past history of the different entities being formed. Lastly, in another work, Chiang et al. [4] introduced the *static first, dynamic second* approach to cluster similar records using a temporal model. In the thesis, we use the temporal model of Chiang et al. [3] to calculate recurrence rates of attributes, and we adapt their entity mutation's idea to be used in a non-clustering scenario.

3 Preliminaries

This chapter gives the problem definition and defines temporal Jaccard and others similarity functions that will be useful throughout the thesis. In addition, it defines the concept of temporal model and describes our specific model choice.

3.1 Problem definition

We now define the problem of similarity join with temporal records and the notation that we will use throughout the thesis.

Definition 1 (Temporal similarity join) *Consider a set $\mathcal{R}$ of records. Each record $x \in \mathcal{R}$ consist of a set of tokens $x = \{x_1, x_2, ...\}$ drawn from a finite universe $\mathcal{U} = \{u_1, u_2, ...\}$ and a timestamp $x.t$. Tokens in $\mathcal{U}$ all represent the same multi-valued attribute, which is the only attribute of records. Consider a temporal similarity function $sim_T(x, y)$ which takes as input two records and returns a similarity value in $[0, 1]$. Consider threshold θ in $[0, 1]$. Given $\mathcal{R}$, θ, $sim_T(x, y)$ a temporal similarity join finds all pairs of records $\langle x, y \rangle$ with $x, y \in \mathcal{R}$, such that their similarities are greater or equal to the given threshold, i.e., $sim_T(x, y) \geq \theta$.*

We want a record to be a set of tokens and not a bag. Therefore, since a token can occur multiple times in the same record, we treat each subsequent occurrence of the same token as a new token. This transformation enables to perform set operations (intersection, difference, etc.) with records.

In addition, we denote by $|x|$ the *size* of record x, i.e., the number of tokens in x; the same notation applies to sets derived from operations on two or more records (e.g., $|x \cup y|$ denotes the size of the union set between x and y). A global ordering $\mathcal{O}$ defines an ordering for all elements of $\mathcal{U}$. A record can be canonicalized by arranging its tokens according to a global ordering $\mathcal{O}$. For each distinct token u, we name *document frequency* the number of records that contain u. We can use document frequency as global ordering, and we denote it by $\mathcal{O}_{df}$. Using $\mathcal{O}_{df}$ to canonicalize records is a heuristic for accelerating similarity joins [2]. However, for the ease of illustration, in our examples we will use the alphabetical ordering $\mathcal{O}_a$ among tokens.

Several traditional similarity functions can be transformed into temporal versions, and the choice between one function or another is usually dependent on the application domain. In this thesis we only consider the temporal version of the Jaccard similarity function. We name this function "temporal Jaccard", and we denote it by $J_T(x, y)$. Thus, the specific similarity join problem that we will solve deals with finding all pairs $\langle x, y \rangle$, such that $J_T(x, y) \geq \theta$. Temporal Jaccard is defined in the following section.

As stated in the above definition, all tokens refer to the same attribute of records. E.g., tokens could be titles of web pages visited by users or meals ordered by customers of a restaurant. In this thesis, for illustrating the temporal model and for the experimental evaluation we use a set of records that refer to different scientific papers. Specifically, each record describes the main author of a paper by its coauthors attribute. Then, we can tokenize the coauthors attribute of a record and get a set of tokens of this type:

$$x = \{\text{"James"}, \text{"Rodriguez"}, \text{"Thomas"}, \text{"Mueller"}, \text{"John"}, \text{"Smith"}\}$$

3.2 Similarity functions

For any two records, a *similarity function* returns the degree of similarity between them, computed on the basis of an arbitrarily defined metric. The similarity degree is usually expressed by a value in $[0, 1]$, where 0 means records are completely different and 1 means they are perfectly equal.

The thesis focuses on developing algorithms for computing similarity joins based on *temporal Jaccard* similarity. However, in order to develop some theoretical concepts, in the following chapters we also employ two other similarity functions, namely *overlap similarity* and *Hamming distance*. We describe these three similarity functions in this section.

Temporal Jaccard similarity is a variation of classical Jaccard similarity [1]. What differ are weights for intersection (agreement) and difference (disagreement) between records, which are based on records' temporal data.

Definition 2 (Temporal Jaccard) *We denote temporal Jaccard by symbol J_T. Weight for agreement is denoted by w_a, while weight for disagreement by w_d. For two records x, y, each described by a set of tokens, temporal Jaccard is defined by the following formula:*

$$J_T(x, y) = \frac{w_a|x \cap y|}{w_a|x \cap y| + w_d(|x \cup y| - |x \cap y|)} \tag{3.1}$$

$|x \cap y|$ constitutes the number of tokens shared between x and y, while $(|x \cup y| - ||x \cap y|)$ is the total number of different tokens between x and y. Weights for agreement and disagreement are computed using labeled data (see section 3.3).

In general, agreement and disagreement weights help handling duplicate detection problems arising from ambiguity caused by *between-entity temporal agreement* and *within-entity temporal disagreement* [3]. Between-entity temporal agreement arises when two records that refer to different entities are very similar; e.g., two persons can have the same name. On the other hand, within-entity temporal disagreement arises when two records that refer to the same entity at different time points have different values. This happens because entities evolve over time from a certain state to another; e.g., a person can change its address or its job over time. The use of temporal information can enhance duplicate detection as shown by Li et al. [9] and

[1] Classical Jaccard similarity is usually denoted by J and is defined by the following formula: $J(x, y) = \frac{|x \cap y|}{|x \cup y|}$. We will also use this notation in the thesis.

Chiang et al [3]. We cite the comprehensive definitions of between-entity temporal agreement and within-entity temporal disagreement [3]:

Definition 3 (Between-entity temporal agreement) *Between-entity temporal agreement (also called "temporal agreement") arises when two records referring to two different entities have identical or highly similar values on one attribute because over time one of the entities evolved to have the same value in some attribute as that previously held by the other.*

Definition 4 (Within-entity temporal disagreement) *Within-entity temporal disagreement (also called "temporal disagreement") arises when two records referring to the same entity have dissimilar values on one attribute because over time their associated entity evolves its state on that attribute.*

The idea is to define a temporal model that, given records and their temporal information, calculates different agreement and disagreement probabilities. Then, weights should be set accordingly. If agreement probability is high, agreement between records should not be used as evidence that records refer to the same entity; thus, agreement should be low weighted. On the other hand, if disagreement probability is high, differences between records do not imply that records refer to different entities; in this case, "penalty" for disagreement should be reduced. The following example illustrates temporal Jaccard calculation.

Example 1 *Consider two records x, y, each described by a set of tokens:*

$$x = \{A, B, C, D, E, G\}$$
$$y = \{A, B, D, F\}$$

Records have temporal information (i.e., time point when they were up-to-date) denoted by x_t and y_t:

$$x_t = 2001$$
$$y_t = 2005$$

In addition, consider a temporal model that defines functions $w_a(x, y)$ and $w_d(x, y)$. Given two records and their temporal information, the functions return respectively agreement and disagreement weights. Assume that for records above we get $w_a = 0.9$ and $w_d = 0.1$. The weight for disagreement represents a "penalty" in similarity calculation; the higher the weight, the lower is the similarity. In this example, we note that disagreement between entities shouldn't be taken as evidence that records refer to different entities.

Finally, temporal Jaccard similarity between x and y is computed as follows:

$$J_T(x, y) = \frac{0.9 \cdot 3}{0.9 \cdot 3 + 0.1 \cdot 4} \approx 0.87$$

On the other hand, the same two records give the following classical Jaccard similarity:

$$J(x, y) = \frac{3}{7} \approx 0.43$$

Overlap similarity is simply the size of the intersection of two records, while *Hamming distance* is the size of the symmetric difference. Hamming distance does not return a value in $[0, 1]$ but a "distance" value between two records. Intuitively, when distance is great, similarity is low; when distance is zero records are perfectly equal (similarity is one). The definitions of overlap similarity and Hamming distance are the following:

Definition 5 (Overlap similarity) *We denote overlap similarity by O. For two records x, y, each described by a set of tokens, overlap similarity is defined by the following formula:*

$$O(x, y) = |x \cap y| \tag{3.2}$$

Definition 6 (Hamming distance) *We denote Hamming distance by H. For two records x, y, each described by a set of tokens, Hamming distance is defined by the following formula:*

$$H(x, y) = |x \cup y| - |x \cap y| \tag{3.3}$$

Finally, we consider some reformulations of temporal Jaccard. In light of definitions 5 and 6 we can reformulate temporal Jaccard this way:

$$J_T(x, y) = \frac{w_a O(x, y)}{w_a O(x, y) + w_d H(x, y)} \tag{3.4}$$

Another reformulation that will be useful throughout the thesis is the following:

$$J_T(x, y) = \frac{w_a O(x, y)}{w_a O(x, y) + w_d(|x| + |y| - 2O(x, y))} \tag{3.5}$$

3.3 Temporal model

A temporal model provides the basis to capture entity evolution and to apply it to calculate agreement and disagreement weights. We can say that it defines between-entity temporal agreement and within-entity temporal disagreement in terms that can be numerically calculated. Normally, a temporal model needs a set of labeled training data set to learn how entities evolve. Then, after this training phase it generates some functions to calculate the different weights.

When dealing with temporal Jaccard similarity, a temporal model must provide two functions $w_a(x, y)$ and $w_d(x, y)$ that return respectively weights for agreement and disagreement to be used in the calculation of $J_T(x, y)$.

Note that we choose a specific temporal model for illustration purpose. The solutions we propose can be applied to compute temporal similarity joins irrespective of the specific temporal model chosen.

The temporal we choose is a variation of the temporal model developed by Chiang et al. [3]. This model is based on the concepts of value recurrence and mutation function. It acts in three step.

1. Learns recurrence rates of attributes from a labeled training data set.

2. Generates a mutation function.

3. Applies mutation during the clustering phase of the algorithm to calculate attributes weights.

In our variation we keep the first step as the original. Since we do not execute clustering and since the mutation function uses already formed clusters to compute its output, we use a modified version of the mutation function.

We next describe our variation of the temporal model developed by Chiang et al. [3]. The interested reader is referred to [3] for the original model.

3.3.1 Value recurrence

Let E be an entity and $E.R$ a set of records associated with the entity. Consider multi-valued attribute A. Given an attribute value u and a time interval Δt, the recurrence rate $rec(\Delta t)$ of u indicates the probability of record x having attribute value u at time $t + \Delta t$, given that x has the same value u at time t.

To calculate recurrence rates, we need the notions of recurrence hit and miss [3, p. 1179].

Definition 7 (Δt recurrence hit) *A value u has a Δt recurrence hit if u satisfies following conditions: 1) value u occurs in a record associated with some entity E, and 2) u recurs Δt time units later in a record associated with the same entity E.*

Definition 8 (Δt recurrence miss) *A value u has a Δt recurrence miss if u satisfies following conditions: 1) value u occurs in a record associated with some entity E, and 2) u does not recur in any record associated with E over the next Δt time units.*

rID	eID	main author	co-authors	year
r_1	e_1	Geoffrey Chu	Aaron Harwood, Peter Stuckey	2009
r_2	e_1	Geoffrey Chu	Peter Stuckey	2012
r_3	e_1	Geoffrey Chu	M. de la Banda, Peter Stuckey	2012
r_4	e_1	Geoffrey Chu	M. de la Banda, Christopher Mears, Peter Stuckey	2014
r_5	e_1	Geoffrey Chu	Peter Stuckey	2015

Table 3.1: Records of the same main author.

We now describe the steps of the algorithm for learning recurrence rates. Consider a training data set, where records are grouped according to their associated entities.

1. For each entity E with its records sorted in increasing temporal order, maintain the occurrence history of $E.t_u$ for each value u that appears in at least one record associated with E. Each occurrence history is an ordered list of timestamps sorted in increasing temporal order, where each timestamp describes a point when a record associated with entity E has a value u. E.g., if we consider the entity shown in table 3.1, the occurrence history of value "M. de la Banda" would be $\{2012, 2014\}$.

2. For each occurrence history $E.t_u$ identify and fill in possible missing occurrences. For each consecutive pair of timestamps t_i, t_{i+1} in $E.t_u$, if entity E does not have any records with time stamp within (t_i, t_{i+1}), then we assume that entity E has value equal to u from time t_i to t_{i+1} and insert all possible timestamps $t_i < t < t_{i+1}$ into $E.t_u$. E.g., consider again the example in table 3.1. We notice that e_1 worked with "M. de la Banda" in 2012 and 2014, and that e_1 has no records in 2013. Therefore, it is possible that e_1 worked with "M. de la Banda" also in 2013, and we fill in the missing occurrence of "M. de la Banda" in the occurrence history. The updated occurrence history of value "M. de la Banda" would be $\{2012, 2013, 2014\}$.

3. Count and aggregate the numbers of hits $h_{\Delta t}$ and misses $m_{\Delta t}$ of Δt-recurrences on attribute A from the updated occurrence histories of all attribute values for each $0 < t \leq t_{max}$. This can be done by using a sliding window that iterates through each occurrence history. Miss that fall on years beyond the latest timestamp of entity E must not be counted.

4. Construct the recurrence function $rec(\Delta t)$ based on the following equation:

$$rec(\Delta t) = \begin{cases} 1 & \Delta t = 0 \\ \dfrac{h_{\Delta t}}{h_{\Delta t} + m_{\Delta t}} & 1 < \Delta t \leq t_{max} \\ \dfrac{h_{\Delta t_{max}}}{h_{\Delta t_{max}} + m_{\Delta t_{max}}} & \Delta t \geq t_{max} \\ 0 & \text{otherwise} \end{cases}$$

Example 2 *Consider the example in table 3.1, where we would like to learn the recurrence function. Assume that we are currently learning hits and miss for value "M. de la Banda". As seen in step 2 of the procedure, the occurrence history of value "M. de la Banda" (after filling in the missing occurrences) is* $\{2012, 2013, 2014\}$. *Thus, for* $\Delta t = 1$ *we get 2 hits and 1 miss, and for* $\Delta t = 2$ *we get 1 hit and 1 miss.*

In addition, for $\Delta t = 1$ *we get the following hits and misses for the other attributes of entity* e_1. *"Peter Stuckey": 6 hits, 0 misses; "Aaron Harwood": 0 hits, 1 miss; "Christopher Mears": 0 hits, 1 miss. Therefore, we get in total 8 hits and 3 misses for* $\Delta t = 1$. *Assume entity* e_1 *is the only entity in our training data set. Then* $rec(\Delta t = 1) = \frac{8}{8+3} \approx 0.73$.

3.3.2 Weights calculation

Given two records x, y, with $x.t \leq y.t$, and the recurrence function $rec(\Delta t)$, we compute the weights for agreement and disagreement as follows. First, we calculate value m in the following way[2]:

$$m(x, y) = (1 - rec(y.t - x.t))^{\sqrt{|x| \cdot |y|}} \tag{3.6}$$

[2] m should not be confused with the mutation function M proposed in [3], although the concept is similar.

We now explain how m should be interpreted. Since the recurrence rate gives the probability (given $x.t < y.t$) that a value that appears in x will appear also in y, then $1 - rec(y.t - x.t)$ gives the probability that this value will not reappear. Thus, if we would like to compute the probability that no value in x will reappear after $y.t - x.t$ time, we should raise $1 - rec(y.t - x.t)$ to a power of $|x|$. This can be interpreted as the probability that no value of record x will recur in y. However, since we already know the size of y, we cannot ignore it in the calculation, because it is obvious that when y has many values, the probability of recurrence is higher than when y has only few values. E.g., assume that $|x| = 4$ and $|y| = 1$; then the probability of recurrence should be less than in case $|x| = 4$ and $|y| = 20$. For this reason, we decide to use the geometric mean to calculate the exponent of the formula. Finally, we can say that:

- When m is great, the probability that a value in x will recur in y is low.
- When m is small, it is likely that a value in x will recur in y.

Once the m value for two records has been computed, we need to apply it for calculating the weights for agreement and disagreement. If the values in x are not likely to reappear in y (i.e., m is great), then similar values between x and y come probably from between-entity temporal agreement. As a result, a lower weight should be assigned to agreement.

On the other hand, if the values in x are not likely to reappear in y, the fact that x and y do not have similar values should not be used as evidence that x and y refer to different entities. Therefore, the penalty for disagreement between x and y should be reduced. Since in temporal Jaccard formula disagreement is in the denominator part, to reduce the penalty we need to assign a lower weight to disagreement.

After these consideration, we decide to use the following formulas for agreement and disagreement weights in temporal Jaccard calculation:

$$w_a = 1 + \sigma \cdot (1 - m) \tag{3.7}$$

$$w_d = 1 - \sigma \cdot m \tag{3.8}$$

These formulas are proposed in [3, p. 1182], but they do not apply them to temporal Jaccard calculation. σ is a factor that controls the importance of m and could be changed. For simplicity we will keep it fixed to 0.5 throughout the whole thesis. In addition, we use the heuristic that weights for agreement are always higher than weights for disagreement. This ensures that the similarity result will always be dominated by the common values between the records rather than by the unequal values. The same heuristic was used also in [3] with a similar purpose.

4 Solution

In this chapter, we illustrate the refinement of the prefix filtering, positional filtering, and suffix filtering techniques. In particular, in section 4.1 we illustrate conversions of temporal Jaccard into overlap and Hamming distance metrics. These conversions are essential for developing prefix, positional, and suffix filtering techniques. In chapter 4.2 we describe our extension of the prefix-filtering technique, mainly developed by Chaudhuri et al. [2] and Bayardo et al. [1], in order to handle the temporal Jaccard metric. Then, chapters 43 and 4.4 present respectively our extensions of the positional filtering and suffix filtering algorithms, originally devised by Xiao et al. [16], in oder to handle the temporal Jaccard metric.

4.1 Conversions of similarity functions

A temporal similarity join computes the similarity of every pair of records and checks it against a similarity constraint $J_T(x, y) \geq \theta$. The conversion of this constraint into an equivalent overlap constraint is the premise that enabled the development of prefix and positional filtering techniques. In particular, positional filtering deals with finding the maximum possible overlap value between two records after having seen only some parts of them. This maximum overlap value would be of no help if it could not be checked against an equivalent temporal Jaccard constraint. We will see in chapters 4.2 and 4.3 how this works in detail.

The conversion between a temporal Jaccard constraint and an overlap constraint is the following:

Proposition 1 *Consider temporal Jaccard similarity function $J_T(x, y)$,, overlap similarity function $O(x, y)$, and two records x, y. Let $\theta \in [0, 1]$ be the threshold for similarity calculation. Then,*

$$J_T(x, y) \geq \theta \iff O(x, y) \geq \alpha = \frac{\theta w_d}{(1-\theta)w_a + 2\theta w_d} \cdot (|x| + |y|) \tag{4.1}$$

Proof: By definition,

$$J_T(x, y) = \frac{O(x, y)w_a}{O(x, y)w_a + (|x| + |y| - 2O(x, y))w_d}$$

Thus,

$$J_T(x, y) \geq \theta \iff \frac{O(x, y)w_a}{O(x, y)w_a + (|x| + |y| - 2O(x, y))w_d} \geq \theta$$

Finally, if we rearrange the right-hand equation, we get

$$\begin{aligned}
&\frac{O(x,y)w_a}{O(x,y)w_a + (|x| + |y| - 2O(x,y))w_d} \geq \theta \\
\iff &(w_a - \theta w_a + 2\theta w_d) \cdot O(x,y) \geq \theta w_d(|x| + |y|) \\
\iff &O(x,y) \geq \frac{\theta w_d}{(1-\theta)w_a + 2\theta w_d} \cdot (|x| + |y|)
\end{aligned}$$

□

On the other hand, the suffix filtering technique needs to convert the constraint into an equivalent Hamming distance one. The conversion between an overlap constraint and an Hamming distance constraint is the following:

Proposition 2 *Consider overlap similarity function $O(x,y)$, Hamming distance $H(x,y)$, and two records x, y. Let $\alpha \geq 0$. Then,*

$$O(x,y) \geq \alpha \iff H(x,y) \leq |x| + |y| - 2\alpha \tag{4.2}$$

Proof: By definition,

$$H(x,y) = |(x-y) \cup (y-x)| = |x| + |y| - 2O(x,y)$$

We can reformulate $O(x,y) \geq \alpha$ in the following way:

$$\begin{aligned}
O(x,y) \geq \alpha &\iff -2O(x,y) \leq -2\alpha \\
&\iff |x| + |y| - 2O(x,y) \leq |x| + |y| - 2\alpha
\end{aligned}$$

From the definition of $H(x,y)$ it follows

$$H(x,y) \leq |x| + |y| - 2\alpha$$

□

In addition, by the transitive property, we have the following conversion between a temporal Jaccard constraint and an Hamming distance constraint:

Proposition 3

$$J_T(x,y) \geq \theta \iff H(x,y) \leq |x| + |y| - 2\alpha \tag{4.3}$$

with $\alpha = \frac{\theta w_d}{(1-\theta)w_a + 2\theta w_d} \cdot (|x| + |y|)$.

4.2 Prefix filtering

In this chapter, we first review the prefix filtering technique, then we refine it in order to handle the temporal Jaccard similarity metric, and finally we present an algorithm that exploits prefix filtering to perform temporal similarity joins. The algorithm presented here will then be extended in chapters 4.3 and 4.4 with positional and suffix filtering. Prefix filtering was mainly developed by Chaduri et al. [2] and Bayardo et al. [1]. In addition, this chapter is also based on concepts illustrated in [16].

Existing prefix filtering-based methods for computing similarity joins with *classical Jaccard* metric exploit the conversion of the Jaccard constraint into an overlap constraint[1]. We have seen that a similar conversion can also be applied to the *temporal Jaccard* constraint (see equation 4.1). In addition, existing methods are based on two other important concepts, namely *inverted indexes* and *prefix filtering principle*.

The *inverted index* of a given token u simply keeps track of all records containing u. In reality, the use of inverted indexes is sufficient for designing an algorithm that computes similarity joins. An algorithm of this type must consist roughly of the following steps:

1. Build inverted indexes for all tokens in the record set.
2. Scan each record x.
3. Probe indexes using every token in x and obtain a (multi)set of candidate pairs[2].
4. Merge the elements of this (multi)set together to get the actual overlap of x with every other candidate record.
5. Extract the final result by removing records whose overlap is less than the overlap constraint α.

However, this approach presents two relevant drawbacks [16, p. 133]. First, some tokens (known as "stop words") can generate very long inverted indexes, causing significant overhead. Second, all pairs of records that share at least one token have to be stored during the calculation; this is often a prohibitive task.

The *prefix filtering principle* reflects the following intuition: if two canonicalized records have a total overlap value over a given threshold, then they must necessarily produce a minimum overlap value also when considering only a portion of their tokens. In particular, the principle exploits the global ordering of tokens among records to augment the benefit of this intuition. The prefix filtering principle is formalized in the following lemma originally devised by Chaudhuri et al. [2] and then rephrased by Xiao et al. [16] in this way:

[1]The equation for converting a *classical Jaccard* similarity constraint into an overlap constraint is the following:

$$J(x, y) \geq t \iff O(x, y) \geq \frac{t}{1+t} \cdot (|x| + |y|)$$

For more information about prefix filtering with classical Jaccard see Xiao et al. [16, pp. 132–133].

[2]A candidate pair is a pair of records that couldn't be pruned by the algorithm and constitutes a potential result. The set of candidate pairs is a superset of final result of the similarity join.

Lemma 1 (Prefix filtering principle) *Consider an ordering $\mathcal{O}$ of the token universe $\mathcal{U}$ and a set of records, each with tokens sorted in the order of $\mathcal{O}$. Let the p-prefix of a record x be the first p tokens of x. If $O(x, y) \geq \alpha$, then the $\lfloor |x| - \alpha + 1 \rfloor$-prefix of x and the $\lfloor |y| - \alpha + 1 \rfloor$-prefix of y must share at least one token.*

We can intuitively understand that when two records have no overlap between their prefixes, then the remaining tokens (i.e., those in the suffixes) are not enough to get a total overlap that is at least equal to α. Very roughly, the reason is that suffixes of both records have length equal to $\alpha - 1$ and, in addition, the global order among tokens prevent tokens in the suffix of one record from overlapping with tokens in the prefix of the other[3].

The prefix filtering principle suggests the creation of an algorithm that seeks all pairs of records that share at least one token in their prefixes. All other pairs can be safely pruned. Then, the remaining pairs (named "candidate pairs") must be verified, i.e., their actual overlap must be computed and compared with the overlap constraint. The reason for this final step is easily understood: lemma 1 states that prefix filtering is only a necessary (and not a sufficient) condition for an overlap similarity greater than α between two records. In addition, we note that the principle is well suited for being used in combination with inverted indexes. Actually, the use of inverted indexes can speed up the search of candidate pairs.

In summary, an algorithm that combines prefix filtering and inverted indexes should consist of the following general phases:

1. *Indexing phase*: build inverted indexes for tokens that appear in the prefix of each record.

2. *Candidate generation phase*: probe inverted indexes for tokens in the prefix of each record; merge record identifiers returned to generate a set of candidate pairs.

3. *Verification phase*: evaluate the similarity of every candidate pair and prune those that do not meet the threshold.

If we adopt the theoretical division of the flow of signature-based algorithms seen in section 2.2, we can say that the first two phases together form the *filtering step* of the algorithm. Throughout these phases, the initial set of all possible pairs (which has size $\frac{n!}{(n-2)!\cdot 2}$) is filtered and reduced to only a subset of candidate pairs. This set of candidate pairs still contains all pairs with a similarity greater than α.

Example 3 *Consider the alphabetical order $\mathcal{O}_a$ of the token universe $\mathcal{U}$ and a set of four records, each with tokens sorted in the order of $\mathcal{O}_a$:*

$$w = \{\underline{A}, \underline{B}, C, F\}$$
$$x = \{\underline{C}, \underline{D}, \underline{E}, F, G\}$$
$$y = \{\underline{A}, \underline{C}, D, E\}$$
$$z = \{\underline{E}, G, H\}$$

[3] See Chaudhuri et al. [2] for more details.

Assume that we want to compute a similarity join to get pairs with $O(x, y) \geq \alpha = 3$. Tokens in the prefixes are indexed and underlined. Prefix lengths are calculated according to the prefix filtering principle. After the indexing phase*, we get the following inverted indexes:*

$$A = \{w, y\}$$
$$B = \{w\}$$
$$C = \{x, y\}$$
$$D = \{x\}$$
$$E = \{x, z\}$$

*Then, after probing inverted indexes and merging candidates (*candidate generation phase*), we get the following set of candidate pairs:*

$$\mathcal{S} = \{\langle w, y\rangle, \langle x, y\rangle \langle x, z\rangle\}$$

Since $\langle w, y\rangle$ and $\langle x, z\rangle$ have overlap equal to 2, they are pruned during the verification phase*. Thus, the only pair in the final result is $\langle x, y\rangle$.*

Unfortunately, the algorithm outlined above is only suitable for computing similarity joins using a fixed overlap similarity threshold. When the overlap threshold (α) is derived from a fixed temporal Jaccard threshold (θ), we face a new issue, i.e., α is no more fixed but changes frequently, depending on the length of different records being compared. Therefore, since the prefix of a record depends on the size of α, it cannot be determined before hand. In order to solve this issue, we need to find a prefix value which depends only on the particular record being currently probed by the algorithm. We tackle this problem in the following chapter.

4.2.1 Prefix length

Lemma 2 is an extension of lemma 1, and it is the first step to define a prefix length that can be used for computing temporal similarity joins with a temporal Jaccard threshold. It states the new prefix length to be probed for a record; this length is more independent of other records in the data set than prefix length in lemma 1, since for a given record x (y) only x's size (y's size) is needed for the calculation, while prefix length in lemma 1 needs always also the size of another record to be calculated. In addition, lemma 2 considers a fixed temporal Jaccard threshold instead of a general overlap threshold.

Lemma 2 *Consider an ordering $\mathcal{O}$ of the token universe $\mathcal{U}$ and a set of records, each with tokens sorted in the order of $\mathcal{O}$. Let $J_T(x, y)$ be the temporal Jaccard similarity function defined in section 3.2. For any two records x, y, with $|x| \geq |y|$, if $J_T(x, y) \geq \theta$, then the prefix of y of length*

$$\left\lfloor \frac{(1-\theta)w_a}{(1-\theta)w_a + 2\theta w_d} \cdot |y| + 1 \right\rfloor \tag{4.4}$$

and the prefix of x of length

$$\left\lfloor \frac{(1-\theta)w_a}{(1-\theta)w_a + \theta w_d} \cdot |x| + 1 \right\rfloor \tag{4.5}$$

must share at least one token[4]*.*

Proof: The proof is divided into two parts. First, we need to prove that temporal Jaccard similarity function increases monotonically when the overlap value increases and all other values remain constant. Second, we calculate the general maximum possible temporal Jaccard for variable prefix lengths, and we find out the minimum number of tokens that a record needs in its prefix to avoid pruning false negatives.

We prove now that temporal Jaccard similarity function increases monotonically when the overlap value increases. By definition, we know that temporal Jaccard function can be expressed this way (see section 3.2):

$$J_T(x,y) = \frac{w_a O(x,y)}{w_a O(x,y) + w_d(|x| + |y| - 2O(x,y))}$$

If $O(x,y)$ is the only variable, the domain of the function given x and y values is $0 \leq O(x,y) \leq min(|x|,|y|)$. We also know that $O(x,y)$ is always an integer, and thus, the general domain of the function is $\mathbb{N}$. In addition, consider $w_a, w_d > 0$ and $|x|, |y| \geq 1$. Since $|x|$ and $|y|$ cannot be smaller than $O(x,y)$, the denominator of the equation is never negative or 0. After rearranging equation 4.6, we get

$$J_T(x,y) = \frac{w_a O(x,y)}{(w_a - 2w_d)O(x,y) + w_d(|x| + |y|)}$$

Since $O(x,y)$ is the only variable, $w_d(|x|+|y|)$ is a constant part of the function, and we name it c. In addition, we know that $c > 0$.

$$J_T(x,y) = \frac{w_a O(x,y)}{(w_a - 2w_d)O(x,y) + c}$$

We compute the derivative and we get

$$\frac{dJ_T(x,y)}{dO(x,y)} = \frac{w_a c}{[(w_a - 2w_d)O(x,y) + c]^2}$$

Since both the numerator and the denominator are positive for every $O(x,y) \geq 0$, we can conclude that the derivative is always positive for every $O(x,y) \geq 0$. Thus, $J_T(x,y)$ increases monotonically for variable $O(x,y)$ and constant w_a, w_d, $|x|$, $|y|$.

Consider now two records x, y, and, without loss of generality, $|x| \geq |y|$. Consider an ordering $\mathcal{O}$ of the token universe $\mathcal{U}$, and let the tokens of each record be sorted in the order of $\mathcal{O}$.

[4]Note the different denominators in the fraction of the equations.

Let x_n be the token of record x at position n (consider positions starting from 1). The notation $x_m \prec y_n$ means that token y_n comes after token x_m in the global order $\mathcal{O}$. In addition, let i be the prefix length of x, while j the prefix length of y. See figure 4.1 for a graphical description of the situation. The goal is to find the minimum values of i and j that, for a given threshold θ, guarantee the following: if x and y share no common token in their prefixes, then temporal Jaccard similarity between x and y is less than θ. In order to prove this, we must distinguish two cases.

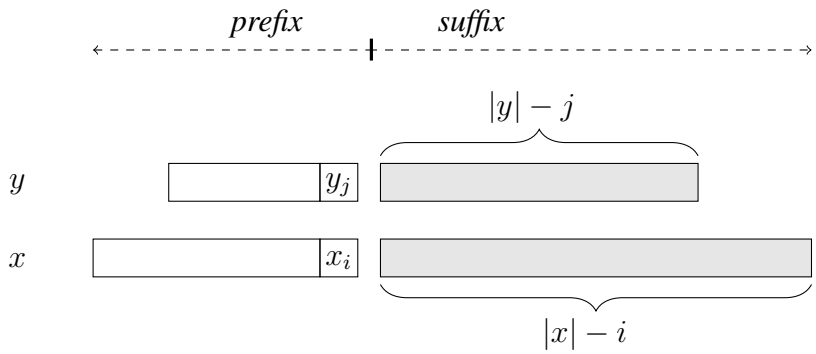

Figure 4.1: Prefix and suffix of records x and y.

Case 1: $y_j \preceq x_i$. In this case, if there is no overlap between x's prefix and y's prefix, the maximum possible overlap between the two whole records is $min(|y| - j, |x|) = |y| - j$. According to equation 4.6, we get the following maximum possible temporal Jaccard similarity between x and y:

$$\max J_T(x, y) = \frac{w_a(|y| - j)}{w_a(|y| - j) + w_d[|x| + |y| - 2(|y| - j)]}$$

Since $|x| \geq |y|$, we can substitute $|x|$ with $|y|$ and get a value greater or equal to $\max J_T(x, y)$.

$$\max J_T(x, y) \leq \frac{w_a(|y| - j)}{w_a(|y| - j) + 2w_d \cdot j}$$

Now, we must ensure that this value is always less than threshold θ, i.e., the following condition must hold:

$$\frac{w_a(|y| - j)}{w_a(|y| - j) + 2w_d \cdot j} < \theta$$

If this condition holds, then also $\max J_T(x, y)$ is always less then θ. We solve the equation for j, and we get

$$\frac{(1 - \theta)w_a}{(1 - \theta)w_a + 2\theta w_d} \cdot |y| < j$$

Thus, since prefix must be an integer, the prefix length for record y is

$$\left\lfloor \frac{(1-\theta)w_a}{(1-\theta)w_a + 2\theta w_d} \cdot |y| \right\rfloor$$

Case 2: $x_i \prec y_j$. In this case, if there is no overlap between x's prefix and y's prefix, then the maximum possible overlap between the two whole records is $min(|y|, |x| - i)$. In any case, the maximum overlap is less or equal to $|x| - i$. In addition, since no token in the prefix of x will overlap with y, minimum possible Hamming distance is i. It can be easily shown that $J_T(x, y)$ increases monotonically when $H(x, y)$ decreases. Thus, from equation 3.4 we get

$$\max J_T(x, y) = \frac{w_a(|x| - i)}{w_a(|x| - i) + w_d \cdot i}$$

Since J_T must be less than θ, we have

$$\frac{w_a(|x| - i)}{w_a(|x| - i) + w_d \cdot i} < \theta$$

We solve the equation for i, and we get

$$\frac{(1-\theta)w_a}{(1-\theta)w_a + \theta w_d} \cdot |x| < i$$

Thus, since prefix must be an integer, the prefix length for record x is

$$\left\lfloor \frac{(1-\theta)w_a}{(1-\theta)w_a + \theta w_d} \cdot |x| \right\rfloor$$

□

From lemma 2 we can now deduce a new prefix length to be probed and indexed. Since

$$\frac{(1-\theta)w_a}{(1-\theta)w_a + \theta w_d} \geq \frac{(1-\theta)w_a}{(1-\theta)w_a + 2\theta w_d}$$

We deduce from lemma 2 that for a generic record x we only need to index and probe its prefix of length

$$\left\lfloor \frac{(1-\theta)w_a}{(1-\theta)w_a + \theta w_d} \cdot |x| + 1 \right\rfloor \tag{4.6}$$

In addition, we note from lemma 2 that we could probe even a smaller prefix for the record with smaller size between the two records considered in the lemma. In section 4.2.2 we show how this property can be exploited to further enhance the performance of a prefix filtering-based algorithm to compute temporal similarity joins.

Unfortunately, prefix length stated in equation 4.6 still present an issue, i.e., w_a and w_d are not fixed values but also change according to the specific two records that are being compared. The solution is to index and probe the maximum possible prefix length for every record. The idea is formalized in the following corollary of lemma 2:

Corollary 1 *Consider an ordering $\mathcal{O}$ of the token universe $\mathcal{U}$ and a set of records, each with tokens sorted in the order of $\mathcal{O}$. In addition, let $J_T(x, y)$ be the temporal Jaccard similarity function defined in section 3.2 and θ be a fixed threshold for temporal similarity join. Let p_i be the pre-calculated factor equal to:*

$$p_i = \max_{w_a, w_d} \frac{(1-\theta)w_a}{(1-\theta)w_a + 2\theta w_d} \tag{4.7}$$

And let p_p be the pre-calculated factor equal to:

$$p_p = \max_{w_a, w_d} \frac{(1-\theta)w_a}{(1-\theta)w_a + \theta w_d} \tag{4.8}$$

Then, for any two records x, y, with $|x| \geq |y|$, if $J_T(x, y) \geq \theta$, the prefix of y of length

$$\lfloor p_i \cdot |y| + 1 \rfloor \tag{4.9}$$

and the prefix of x of length

$$\lfloor p_p \cdot |x| + 1 \rfloor \tag{4.10}$$

must share at least one token.

After this step we have finally defined a *prefix length for a record being probed which only depends on itself*. This prefix length is shown in equation 4.10. Without such a data set–independent prefix length, prefix filtering could not be applied.

Specifically, if we consider temporal model defined in section 3.3, we have the following values for factors p_i and p_p:

$$p_i = 1 - \theta \tag{4.11}$$

And

$$p_p = \frac{1-\theta}{1-0.5\theta} \tag{4.12}$$

Proof: We know from equations 3.7 and 3.8 that w_a and w_d have the following relation:

$$w_a = w_d + 0.5$$

Thus, we can reformulate equation 4.7 as

$$p_i = \max_{w_a} \frac{(1-\theta)w_a}{(1-\theta)w_a + 2\theta(w_a + 0.5)}$$

Now we consider the equation of p_i a function with a single parameter w_a, and we denote it by $f(w_a)$. To get its maximum we compute the derivative of $f(w_a)$, and we get

$$\frac{df}{dw_a} = \frac{-(1-\theta)\cdot\theta}{[w_a(1+\theta)-\theta]^2}$$

We note that the function is decreasing for $\theta \in [0,1]$. Thus, the maximum occurs when w_a is minimum, i.e. when $w_a = 1$ (since $w_a \in [1, 1.5]$).

The same procedure can be applied to calculate p_p. We substitute w_d with $w_a + 0.5$ and compute the derivative (we denote by $g(w_a)$ the function of p_p).

$$\frac{dg}{dw_a} = \frac{-(1-\theta)\cdot 0.5\theta}{(w_a - 0.5\theta)^2}$$

We note that also this function is decreasing for $\theta \in [0,1]$, and thus, the maximum occurs when $w_a = 1$.

□

Example 4 *Consider the alphabetical order $\mathcal{O}_a$ of the token universe $\mathcal{U}$ and a set $\mathcal{R}$ of four records, each with tokens sorted in the order of $\mathcal{O}_a$. Consider the temporal model defined in section 3.3 and a temporal Jaccard similarity threshold of $\theta = 0.8$. We get the following p_p factor:*

$$p_p = \frac{1-0.8}{1-0.4} = \frac{1}{3}$$

Thus, prefix length for a record $x \in \mathcal{R}$ must be

$$\left\lfloor \frac{1}{3}\cdot|x|+1 \right\rfloor$$

Assume we have the following records in $\mathcal{R}$:

$$w = \{\underline{A}, \underline{B}, C, F\}$$
$$x = \{\underline{C}, \underline{D}, E, F, G\}$$
$$y = \{\underline{A}, \underline{C}, D, E\}$$
$$z = \{\underline{E}, \underline{G}, H\}$$

Tokens in the prefixes are indexed and are underlined. We probe prefixes and get the following set of candidate pairs:

$$\mathcal{S} = \{\langle w, y\rangle, \langle x, y\rangle\}$$

All other pairs are guaranteed to have a temporal Jaccard similarity $J_T < 0.8$ and can be safely pruned.

4.2.2 Indexing prefix

We have seen that lemma 2 states different prefix lengths for two records that are being compared: a short prefix length for the record with smaller size, and a longer prefix length for the record with larger size. This difference can be exploited to enhance the performance a prefix filtering-based algorithm to compute temporal similarity joins.

The enhancement is based on a conceptual division of prefixes. We can say that each record do not have one but two prefixes, namely an *indexing prefix* and a *probing prefix*. We define the *indexing prefix* of a record as the set of tokens that need to be indexed, i.e., tokens for which the record identifier has to be stored in the corresponding inverted index. We define the *probing prefix* of a record as the prefix that need to be probed; tokens in this prefix are probed by the algorithm but they are not necessarily indexed.

In addition, we sort records by their size and scan them in ascending size order. Then, every record x that is scanned has a size larger or equal to the size of every other record y that have been already scanned. This step enables the use of the new prefix length for building inverted indexes. For a given record x, this is equal to

$$\lfloor p_i \cdot |x| + 1 \rfloor \tag{4.13}$$

with p_i defined in equation 4.7. In order to avoid confusion, in later sections prefix in equation 4.13 is named *indexing prefix*, while prefix in equation 4.10 is named *probing prefix*.

If we consider temporal model defined in section 3.3, then, for a given record x, length of *indexing prefix* must be equal to

$$\lfloor (1-\theta) \cdot |x| + 1 \rfloor \tag{4.14}$$

And length of *probing prefix* must be equal to

$$\left\lfloor \frac{1-\theta}{1-0.5\theta} \cdot |x| + 1 \right\rfloor \tag{4.15}$$

Example 5 *Consider the set of records in example 4, as well as the same threshold $\theta = 0.8$ for temporal Jaccard similarity join. Then, for a given record x in the set,* indexing prefix *length must be*

$$\lfloor 0.2 \cdot |x| + 1 \rfloor$$

Therefore, the following tokens marked with an apostrophe must be indexed:

$$\begin{aligned} z &= \{\underline{E}', \underline{G}, H\} \\ w &= \{\underline{A}', \underline{B}, C, F\} \\ y &= \{\underline{A}', \underline{C}, D, E\} \\ x &= \{\underline{C}', \underline{D}', E, F, G\} \end{aligned}$$

In addition, note that tokens in the probing prefixes *of records are still underlined and that we sorted records by their size.*

4.2.3 Size filtering

Size filtering is a technique that can by itself avoid a large number of records comparisons. It enables the direct pruning of all those pairs containing records that do not have a minimum size relative to the size of the record being probed.

Size filtering is based on the following lemma:

Lemma 3 *Consider temporal Jaccard similarity $J_T(x, y)$ and overlap similarity $O(x, y)$ defined in section 3.2. Then,*

$$J_T(x, y) \geq \theta \Longrightarrow |y| \geq \frac{\theta w_d}{(1-\theta)w_a + \theta w_d} \cdot |x| \tag{4.16}$$

Proof: From equation, we know that

$$J_T(x, y) \geq \theta \Longleftrightarrow \frac{O(x, y)w_a}{O(x, y)w_a + (|x| + |y| - 2O(x, y))w_d} \geq \theta$$

Since $|y| \geq O(x, y)$, the following relation holds:

$$|x| - O(x, y) \leq |x| - O(x, y) + |y| - O(x, y)$$

Thus, also the following not strict inequality holds:

$$\frac{O(x, y)w_a}{O(x, y)w_a + (|x| - O(x, y))w_d} \geq \frac{O(x, y)w_a}{O(x, y)w_a + (|x| + |y| - 2O(x, y))w_d}$$

From equation we can infer the following relation:

$$\frac{O(x, y)w_a}{O(x, y)w_a + (|x| - O(x, y))w_d} \geq t$$

We solve for $O(x, y)$ and we get

$$O(x, y) \geq \frac{\theta w_d}{(1-\theta)w_a + \theta w_d} \cdot |x|$$

Finally, since $|y| \geq O(x, y)$, we have

$$|y| \geq \frac{\theta w_d}{(1-\theta)w_a + \theta w_d} \cdot |x|$$

□

Concisely, lemma 3 states that we can directly prune a pair of records when the size of the smaller record is not at least $\frac{\theta w_d}{(1-\theta)w_a+\theta w_d}$-times the size of the larger record.

Example 6 *Consider the set of records in example 4, as well as the same threshold $\theta = 0.8$ for temporal Jaccard similarity join. Assume that for the pair of records $\langle x, z \rangle$ we have $w_a = 1.1$*

and $w_d = 0.6$. Then, record z needs the following minimum size to be compared to x:

$$|z| \geq \frac{0.6 \cdot 0.8}{1.1 \cdot 0.2 + 0.6 \cdot 0.8} \cdot 5$$

I.e.,

$$|z| \geq 3.42$$

Since the size of z is only 3, the pair $\langle x, z \rangle$ can safely be pruned.

4.2.4 Algorithm

We now present our algorithm TPREFIX which exploits the prefix filtering technique to perform temporal similarity joins with a temporal Jaccard threshold.

The algorithm takes as input a collection of canonicalized records already sorted in ascending order of their sizes, the threshold for the temporal Jaccard similarity, and the two precomputed factors p_i and p_p, which are necessary for computing the prefix lengths of records during the algorithm. p_i and p_p are computed according to equations 4.7 and 4.8. For each record, the algorithm sequentially scans its probing prefix (Line 6), and for each token in the probing prefix, the corresponding inverted index is returned and iterated (Lines 7-8). For each record in the inverted index, agreement and disagreement weights are computed at Lines 9-10, and size filtering is performed at Line 11. If size filtering constraint is respected, the pair is then verified against the temporal Jaccard constraint and, if it is the case, inserted into the result set $\mathcal{S}$ (Lines 12-14). Finally, after the whole inverted index of current token u has been iterated, the record is itself inserted in the inverted index of u (Line 19). However, note that this happens only if current token u is in the indexing prefix of the record (Line 18).

Notice that inverted indexes are not built in a separate phase prior to the execution of the algorithm. Instead, they are built on-the-fly during the probing phase. This is possible because when a general record x is being scanned, the tokens in the prefixes of already scanned records y are already in the inverted indexes. In addition, all records z that will be scanned after record x will use inverted indexes that include x. Thus, no record will be excluded from the comparison with each other record. This method has the advantage of reducing inverted indexes lengths when compared to the building of all indexes at the beginning of the algorithm.

Algorithm 4.1 TPREFIX($\mathcal{R}$, θ, p_i, p_p)

Input: $\mathcal{R}$: set of records sorted by ascending order of their size; each record has been canonicalized by a global order $\mathcal{O}$.
θ: temporal Jaccard similarity threshold.
p_i: precomputed factor equal to $\max_{w_a} \frac{(1-\theta)w_a}{(1-\theta)w_a+2\theta(w_a+0.5)}$.
p_p: precomputed factor equal to $\max_{w_a,w_d} \frac{(1-\theta)w_a}{(1-\theta)w_a+\theta w_d}$.

Output: $\mathcal{S}$: set of records pairs $\langle x, y\rangle$, with $J_T(x, y) \geq \theta$.

1: $\mathcal{S} \leftarrow \emptyset$;
2: $I_u \leftarrow \emptyset$ $(\forall u \in \mathcal{U})$; //initialize the inverted index for each token in the universe
3: **for each** $x \in \mathcal{R}$ **do**
4: *iPrefix* $\leftarrow \lfloor p_i \cdot |x| + 1 \rfloor$; //indexing prefix
5: *pPrefix* $\leftarrow \lfloor p_p \cdot |x| + 1 \rfloor$; //probing prefix
6: **for** $i = 1$ to *pPrefix* **do**
7: $u \leftarrow x[i]$;
8: **for each** $y \in I_u$ such that $\langle x, y\rangle \notin \mathcal{S}$ **do**
9: $w_a \leftarrow$ CALCWAGR(x, y);
10: $w_d \leftarrow$ CALCWDIS(x, y);
11: **if** $|y| \geq \frac{\theta w_d}{(1-\theta)w_a+\theta w_d} \cdot |x|$ **then**
12: $J_T \leftarrow$ CALCSIM(x, y);
13: **if** $J_T \geq \theta$ **then**
14: insert $\langle x, y\rangle$ into $\mathcal{S}$;
15: **end if**
16: **end if**
17: **end for**
18: **if** $i \leq$ *iPrefix* **then**
19: insert x into I_u;
20: **end if**
21: **end for**
22: **end for**

4.3 Positional filtering

In this chapter we first review the positional filtering technique developed by Xiao et al. [16], and then we present our refinement in order to handle temporal Jaccard similarity.

4.3.1 Description

Positional filtering was originally devised by Xiao et al. [16]. They implemented the first algorithm that uses this technique, namely PPJOIN, which combines prefix filtering and positional filtering. They observed that, at that time, no algorithm fully exploited the global ordering of tokens which is necessary for prefix filtering. In particular, they noticed that the position of a token in a canonicalized record could be used to further reduce the candidate size.

Positional filtering is based on following lemma [16] (rephrased using notation presented in section 3.1):

Lemma 4 (Positional filtering principle) *Given an ordering $\mathcal{O}$ of the token universe $\mathcal{U}$ and a set of records, each with tokens sorted in the order of $\mathcal{O}$. Let $u = x_i$, u partitions the record into the left partition $x_l(u) = \{x_1..x_i\}$ and the right partition $x_r(u) = \{x_{i+1}..x_{|x|}\}$. If $O(x, y) \geq \alpha$, then for every token $u \in x \cap y$, $O(x_l(u), y_l(u)) + \min(|x_r(u)|, |y_r(u)|) \geq \alpha$.*

Following example should make clear the rationale behind positional filtering:

Example 7 *Consider the alphabetical order $\mathcal{O}_a$ of the token universe $\mathcal{U}$, and two records x, y, each with tokens sorted in the order of $\mathcal{O}_a$:*

$$x = \{\underline{A}, \underline{B}, C, E\}$$
$$y = \{\underline{B}, \underline{C}, D, E\}$$

Tokens in the prefixes are underlined. Assume that the constraint for similarity join is $\alpha = 4$. We note that the pair does not meet the constraint, since $O(x, y) = 3$. However, prefix filtering selects $\langle x, y \rangle$ as a candidate pair, because x and y share a common token in their prefixes.

Now, with the help of the positional filtering principle, we can compute a maximum possible overlap value between x and y. Indeed, if we look at the positions of the common token B we get

$$\max O(x, y) = 1 + \min(2, 3) = 3$$

And since this upper bound is smaller than α, we can safely prune pair $\langle x, y \rangle$.

4.3.2 Algorithm

In this section we present our algorithm TPOSITIONAL, which performs temporal similarity joins utilizing the positional filtering technique. The algorithm extends the TPREFIX algorithm seen in section 4.2.4 with positional filtering as implemented in PPJOIN [16]. The changes with respect to TPREFIX are the following:

- *Inverted indexes* (Lines 9 and 24):
  TPREFIX stores in the inverted indexes of a given token only records and nothing else. Instead, TPOSITIONAL stores also the position at which the token appears (Line 24). I.e., an entry in an inverted index has the form (record id, position). E.g., if token u is found at position 3 in record x, then entry $(x, 3)$ is inserted into I_u. Storing also this positional information is necessary to perform positional filtering (see next point).

- *Positional filtering* (Lines 4 and 13-20):
  The core of positional filtering is in Lines 13-20. At Line 13, the overlap constraint α is computed according to equation 4.1. Then the maximum possible overlap between x and y is computed and stored in variable ***uBound***. Notice that the already accumulated overlap between x and the other records is stored in map A and is taken into account in the ***uBound*** calculation (Lines 14-15). The keys of map A are record ids, while its values have the form (record reference, int, int); for a given record y, the map stores in $A[y.id]$ following three values: a reference to record y, the overlap constraint α between x and y, and the already accumulated overlap between x and y. Finally, at Line 16, ***uBound*** is checked against the overlap constraint; if ***uBound*** is greater or equal to α, the new accumulated overlap between x and y is put in map A, otherwise, the insertion of value $(y, 0, 0)$ prevents that y will be further considered as a candidate for x.

- *Verification* (Line 27):
  Verification is now performed in a dedicated function VERIFY() for every record x. We now describe how function VERIFY() works. We remark that the accumulated overlap between the probing prefix of x and the indexing prefix of other records y is stored in map A.

  First, when the accumulated overlap is zero, pair $\langle x, y \rangle$ can be pruned (Line 1), since this means that the threshold α is not met (see previous point). In addition, VERIFY() utilizes the positional filtering principle to further reduce the workload. At Line 6 it compares the last token in the probing prefix of x with the last token in the indexing prefix of y. Then, only the suffix of the record with the smaller token needs to be intersected with the entire other record. E.g., consider two records v and w. We denote by u_v last token in the prefix of v, and by u_w the last token in the prefix of w. If $u_v \prec u_w$, then the prefix of v will not intersect with the suffix of w, because the prefix of v contains only tokens that are smaller than u_v, while the suffix of w contains only tokens that are larger than u_v. In addition, the first O tokens in w (where O denotes the overlap between the prefixes of v and w and is originally stored in Line 4) can be also omitted from the calculation, since at least O tokens have intersected with v' prefix and therefore will not contribute to any overlap with v' suffix. This optimization is performed in Lines 6-17 of VERIFY().

Algorithm 4.2 TPOSITIONAL($\mathcal{R}$, θ, p_i, p_p)

Input: $\mathcal{R}$: set of records sorted by ascending order of their size; each record has been canonicalized by a global order $\mathcal{O}$.

θ: temporal Jaccard similarity threshold.

p_i: precomputed factor equal to $\max_{w_a} \frac{(1-\theta)w_a}{(1-\theta)w_a+2\theta(w_a+0.5)}$.

p_p: precomputed factor equal to $\max_{w_a,w_d} \frac{(1-\theta)w_a}{(1-\theta)w_a+\theta w_d}$.

Output: $\mathcal{S}$: set of records pairs $\langle x, y\rangle$, with $J_T(x, y) \geq \theta$.

$\mathcal{S} \leftarrow \emptyset$;
$I_u \leftarrow \emptyset$ $(\forall u \in \mathcal{U})$; //initialize the inverted index for each token in the universe
for each $x \in \mathcal{R}$ **do**
    $A \leftarrow$ empty map; *key*: (record id); *value*: (record reference, int, int);
    iPrefix $\leftarrow \lfloor p_i \cdot |x| + 1 \rfloor$; //indexing prefix
    pPrefix $\leftarrow \lfloor p_p \cdot |x| + 1 \rfloor$; //probing prefix
    for $i = 1$ to *pPrefix* **do**
        $u \leftarrow x[i]$;
        for each $(y, j) \in I_u$ **do**
            $w_a \leftarrow$ CALCWAGR(x, y);
            $w_d \leftarrow$ CALCWDIS(x, y);
            if $|y| \geq \frac{\theta w_d}{(1-\theta)w_a+\theta w_d} \cdot |x|$ **then**
                $\alpha \leftarrow \left\lceil \frac{\theta w_d}{(1-\theta)w_a+2\theta w_d} \cdot (|x| + |y|) \right\rceil$;
                $O \leftarrow A[y.id].O$; //get the overlap already accumulated with y
                uBound $\leftarrow O + 1 + \min(|x| - i, |y| - j)$;
                if *uBound* $\geq \alpha$ **then**
                    $A \leftarrow (y, \alpha, O + 1)$;
                else
                    $A \leftarrow (y, 0, 0)$;
                end if
            end if
        end for
        if $i \leq$ *iPrefix* **then**
            insert (x, i) into I_u;
        end if
    end for
    VERIFY(x, A, *pPrefix*, p_i);
end for

Algorithm 4.3 VERIFY(x, A, l_x, p_i)

Input: x: a record in $\mathcal{R}$.
A: map containing overlap of other records with x.
l_x: length of probing prefix of x.
p_i: precomputed factor equal to $\max_{w_a} \frac{(1-\theta)w_a}{(1-\theta)w_a+2\theta(w_a+0.5)}$.

1: **for each** y such that $A[y.id].O > 0$ **do**
2: $u_x \leftarrow$ the last token in the prefix of x;
3: $u_y \leftarrow$ the last token in the prefix of y;
4: $O \leftarrow A[y.id].O$; //get the overlap already accumulated with y
5: $\alpha \leftarrow A[y.id].\alpha$; //get the stored alpha constraint
6: **if** $u_x \prec u_y$ **then**
7: *uBound* $\leftarrow O + |x| - l_x$;
8: **if** *uBound* $\geq \alpha$ **then**
9: $O \leftarrow O + \left|\{x_{(l_x+1)}..x_{|x|}\} \cap \{y_{(O+1)}..y_{|y|}\}\right|$;
10: **end if**
11: **else**
12: $l_y \leftarrow \lfloor p_i \cdot |y| + 1 \rfloor$; //length of indexing prefix of y
13: *uBound* $\leftarrow O + |y| - l_y$;
14: **if** *uBound* $\geq \alpha$ **then**
15: $O \leftarrow O + \left|\{x_{(O+1)}..x_{|x|}\} \cap \{y_{(l_y+1)}..y_{|y|}\}\right|$;
16: **end if**
17: **end if**
18: **if** $O \geq \alpha$ **then**
19: insert $\langle x, y \rangle$ into $\mathcal{S}$;
20: **end if**
21: **end for**

4.4 Suffix filtering

Suffix filtering is a technique developed by Xiao et al. [16] and is used to further reduce the candidate size in similarity joins. Xiao et al. [16] apply suffix filtering in their algorithm PPJOIN+ as an additional filtering barrier after prefix and positional filtering. We now review how suffix filtering works and how we integrated it in our algorithm TSUFFIX, which extends TPOSITIONAL with suffix filtering.

Consider two records x, y, and assume they share a common token at position i of x and at position j of y. We denote by x_p the prefix of record x, i.e. tokens in x from position 1 to position i, and by y_p the prefix of record y, i.e. tokens in y from position 1 to position j. In addition, we denote by x_s the suffix of x, i.e. tokens in x from position $(i+1)$ to position $|x|$, and by y_s the suffix of y, i.e. tokens in y from position $(j+1)$ to position $|y|$. Essentially, suffix filtering works as follows:

1. A threshold Hamming distance value H_{max} is computed. This value is the maximum allowable Hamming distance between x_s and y_s that enables the pair to meet the overlap threshold α.

2. A minimum possible Hamming distance H between the suffixes of x and y is computed, and if $H > H_{max}$, pair $\langle x, y \rangle$ can be pruned.

Step 1 derives from the conversion of the overlap constraint α into an equivalent Hamming distance constraint. We know that (equation 4.2)

$$O(x,y) \geq \alpha \iff H(x,y) \leq |x| + |y| - 2\alpha$$

Since x and y share the same token at positions i and j, then

$$H(x_p, y_p) + H(x_s, y_s) = H(x,y)$$

In addition, assume that O is the overlap between x_p and y_p. Note that in algorithm 4.2 O can be always retrieved, since we store the accumulated overlap between records (Line 17). Then, the Hamming distance between the two prefixes is at least

$$H(x_p, y_p) \geq i + j - 2O$$

Now, we can derive the following Hamming distance constraint for x_s and y_s:

$$H(x_s, y_s) \leq H_{max} = |x| + |y| - 2\alpha - (i + j - 2O) \tag{4.17}$$

On the other hand, step 2 finds a lower bound (i.e., the minimum possible value) for Hamming distance between x_s and y_s in the following way. Consider an arbitrary token u in x_s. We now divide x_s into a left partition x_l and a right partition x_r. x_l contains all the tokens in x that precede u in the global ordering, while x_r contains u and all the tokens that succeed u in the global ordering. In addition, we also divide y into partitions y_l and y_r according to the

position of token u (although u could not occur in y). Then, we calculate a lower bound for Hamming distance between x_s and y_s as follows:

$$H(x_s, y_s) \geq \text{abs}(|x_l| - |y_l|) + \text{abs}(|x_r| - |y_r|) \tag{4.18}$$

This relation holds because x_l shares no common token with y_r, and at the same time y_l shares no common token with x_r.

Example 8 *Consider the alphabetical order $\mathcal{O}_a$ of the token universe $\mathcal{U}$, and two suffixes x_s, y_s, each with tokens sorted in the order of $\mathcal{O}_a$:*

$$x_s = \{C, D, E, \boldsymbol{F}, G, H, I\}$$
$$y_s = \{A, B, C, D, G, H, I\}$$

If we divide the suffixes by token F, we get the following partitions:

$$x_l = \{C, D, E\} \qquad x_r = \{F, G, H, I\}$$
$$y_l = \{A, B, C, D\} \qquad y_r = \{G, H, I\}$$

It is then clear that the lower bound for Hamming distance between x_s and y_s is

$$H = \text{abs}(3 - 4) + \text{abs}(4 - 3) = 2.$$

The token in x_s can be drawn at the center of x_s with a low cost for the algorithm, while the position at which y_s must be partitioned can be discovered using binary search. In addition, suffix filtering can be implemented in a recursive way as shown in example 9.

Example 9 *Consider example 8. If we further divide x_l and y_l by token D, we get the following partitions:*

$$x_{ll} = \{C\} \qquad x_{lr} = \{D, E\} \qquad x_r = \{F, G, H, I\}$$
$$y_{ll} = \{A, B, C\} \qquad y_{lr} = \{D\} \qquad y_r = \{G, H, I\}$$

Note that we denoted by x_{ll} (y_{ll}) the left partition of x_l (y_l) and by x_{lr} (y_{lr}) the right partition of x_l (y_l). Since a general partition x_a shares common tokens only with the respective partition y_a, we can calculate a new lower bound for Hamming distance, i.e.,

$$H = \text{abs}(1 - 3) + \text{abs}(2 - 1) + \text{abs}(4 - 3) = 4.$$

Finally, we extend TPOSITIONAL with the SUFFIXFILTER() function originally presented in [16] and we name the new algorithm TSUFFIX. We implement SUFFIXFILTER() function exactly as shown in algorithm 3 in [16]. However, Xiao et al. [16] decided to perform suffix filtering only once for each candidate pair on the first occasion that the pair is formed. Instead, we decided to perform suffix filtering in the verification phase for each candidate pair that survives positional filtering. The reason is that suffix filtering generates considerable overhead, since it requires binary search, and we noticed that when executing it in the verification phase,

i.e., on less pairs than at the beginning, our algorithm performed better. In order to perform suffix filtering in the verification phase, we need to insert the lines shown in algorithm 4 between Lines 5 and 6 of our function VERIFY() seen in section 4.3.2.

Algorithm 4.4 Insertion between Lines 5 and 6 of VERIFY()

1: $H_{max} \leftarrow |x| + |y| - 2\alpha - (i + j - 2O)$;
2: $H \leftarrow$ SUFFIXFILTER($\{x_{i+1}..x_{|x|}\}$, $\{y_{j+1}..y_{|y|}\}$, H_{max}, 1);
3: **if** $H > H_{max}$ **then**
4: continue; //go to next iteration
5: **end if**

5 Experimental evaluation

This chapter describes our experimental setup and results. We evaluated our algorithms on candidate size, running time, and scalability.

5.1 Experiment setup

5.1.1 Data set

For the evaluation we considered two subsets of the DBLP data set. DBLP is a publicly available data set[1], which contains bibliographic records of major computer science journals and proceedings. A record in DBLP defines a real world published article or paper, and it is usually described by attributes "title", "authors", and "year of publication". We ignored attributes title and main author (we considered as the main author the author listed first in the paper) and created a data set of records as in table 5.1. All records in example table 5.1 refer to papers written by the same main author "Khaled Gaaloul". We can notice that some coauthors appear in more than one records.

rID	co-authors	year
r_1	François Charoy, Claude Godart	2006
r_2	François Charoy, Andreas Schaad, Hannah Lee	2007
r_3	Andreas Schaad, Ulrich Flegel, François Charoy	2008
r_4	Ehtesham Zahoor, François Charoy, Claude Godart	2010
r_5	Erik Proper, François Charoy	2011

Table 5.1: Records in the experimental data set.

We tokenized the coauthors of each record using white spaces and punctuations, and then we built two different subsets of DBLP.

- **DBLP-Large**: This data set contains almost 0.6 millions of records and is used to evaluate filtering algorithms in detail. Given its size, it constitutes a considerable challenge for temporal similarity join algorithms. However, evaluating the brute force algorithm on this data set is almost an impossible task, since the algorithm would require a very long running time.
- **DBLP-Small**: This data set is smaller and contains only about 5,000 records. It is used to compare the filtering algorithms with the brute force algorithm. However, a

[1] http://dblp.uni-trier.de/

detailed evaluation of the performance of filtering algorithms on this data set is not possible, since running times of these algorithms on this data set differ only by some milliseconds.

Some important statistic about the data sets are listed in table 5.2.

Data set	**DBLP-Large**	**DBLP-Small**
Size	578,236	5,049
Avg. record length	9.2	9.7
Distinct tokens ($\|\mathcal{U}\|$)	284,294	16,958

Table 5.2: Statistics about the experimental data sets.

5.1.2 Algorithms

We implemented and used the following algorithms in the experiments:

- **TPREFIX**: Our proposed algorithm that exploits prefix filtering for computing temporal similarity joins.
- **TPOSITIONAL**: Our proposed algorithm that extends TPREFIX with positional filtering.
- **TSUFFIX**: Our proposed algorithm that further extends TPOSITIONAL with suffix filtering.
- **BRUTEFORCE**: An algorithm that computes temporal similarity join using the brute force approach.

All the algorithms use the temporal model described in section 3.3 to calculate temporal weights w_a and w_d. Recurrence rates for attribute co-authors were computed in advance according to procedure described in section 3.3. For this purpose we used a labeled subset of DBLP composed by 100 entities. We calculated recurrence rates for $1 \leq \Delta t \leq 20$. Results are shown in table 5.3.

We preprocessed the records sorting them in the ascending order of their sizes, and we used the document frequency ordering $\mathcal{O}_{df}$ as global ordering among tokens. Note that we did not include preprocessing time in the results of our experiment. In addition, we set parameter MAXDEPTH $= 2$, which is needed by function SUFFIXFILTER() in algorithm 4.4. This parameter defines the total number of recursions performed by suffix filtering (see [16, p. 136] for more informations).

Δt	$rec(\Delta t)$	Δt	$rec(\Delta t)$
1	0.31	11	0.02
2	0.22	12	0.01
3	0.17	13	0.00
4	0.13	14	0.01
5	0.10	15	0.00
6	0.09	16	0.00
7	0.06	17	0.00
8	0.05	18	0.00
9	0.03	19	0.00
10	0.02	20	0.00

Table 5.3: Recurrence rates for attribute "co-authors".

5.1.3 Implementation

All algorithms were implemented as single-threaded Java programs (Java version: jre 1.8.0_45), and we ran them inside of Eclipse (version 4.4.2). All experiments were carried out on a PC machine with Windows 7 OS, a 2.27 GHz Intel CPU, and 4GB of RAM.

5.2 Candidate size

We measured the size of candidate pairs generated by the algorithms on the two data sets with varying similarity thresholds from 0.8 to 0.95. Figure 5.1 shows the results on DBLP-Small (including BRUTEFORCE), while figure 5.2 shows the results on DBLP-Large (without BRUTEFORCE). In addition, the result set is also included in the graphs. The result set is composed by all pairs with an actual similarity value higher than the threshold. Note that in both figures the y-axis is in logarithmic scale.

From the results, we first notice that the size of the join result grows modestly when the similarity decrease. Second, we observe that all filtering algorithms generates a lot less candidate pairs than what BRUTEFORCE does (figure 5.1). We also observe that the three filtering algorithm generates more candidate pairs with the decrease of the similarity threshold. Specifically, TPREFIX grows the fastest, followed by TPOSITIONAL, while TSUFFIX generates candidate sets that have sizes very close to the size of the result.

Differences among the three filtering algorithms are observed more in detail on DBLP-Large. Here we can notice that TSUFFIX prunes a large number of pairs in addition to what TPREFIX and TPOSITIONAL do.

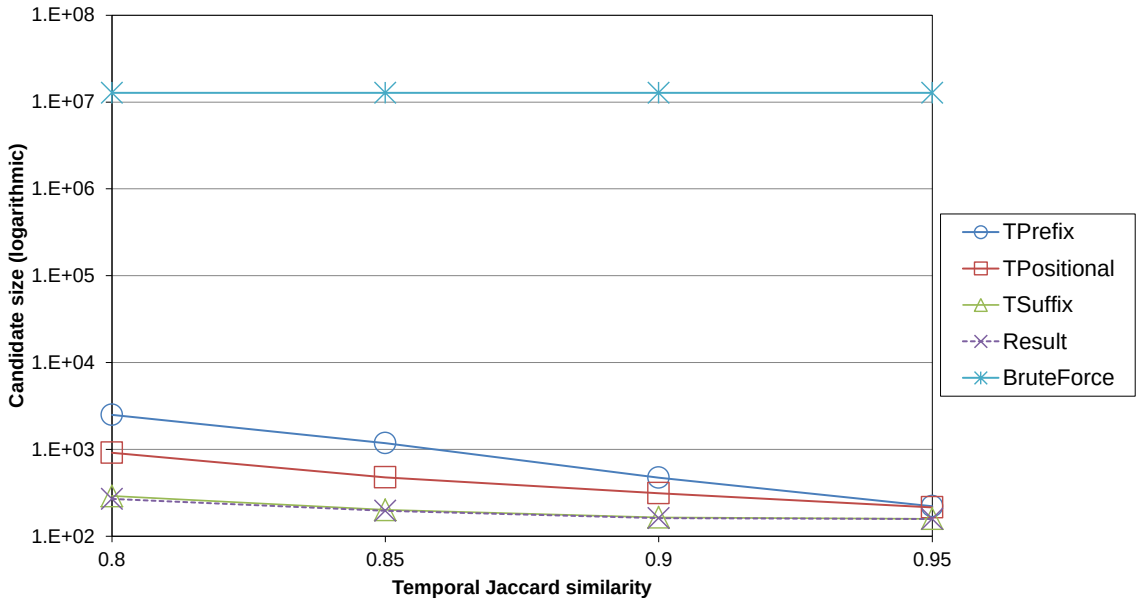

Figure 5.1: DBLP-Small, Candidate size.

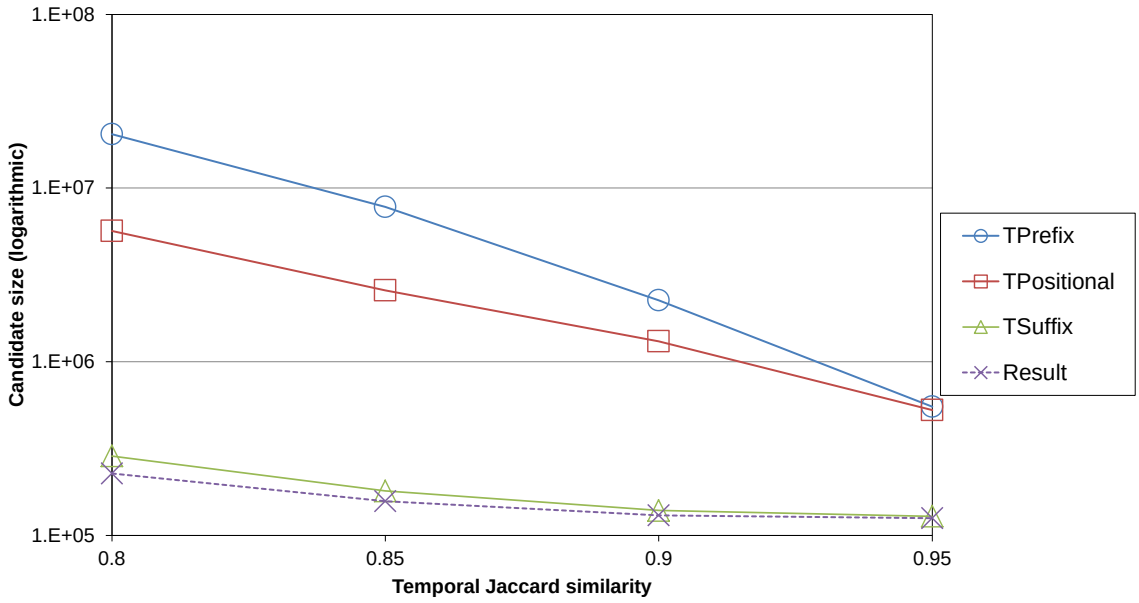

Figure 5.2: DBLP-Large, Candidate size.

5.3 Running time

Figures 5.3 and 5.4 show the running time of the algorithms on the two data sets. Note that, as in the previous experiment, DBLP-Large does not include BRUTEFORCE.

On DBLP-Small (figure 5.3), we observe that the increase in performance of all three filtering algorithms with respect to BRUTEFORCE is impressive. In particular, BRUTEFORCE needs ca. 40 seconds to perform the join on this data set, while at threshold 0.8 all filtering algorithms need only few milliseconds (between 38 and 46 milliseconds). This means approximately a 1000x speed-up. Note that the running time of BRUTEFORCE does not change for different similarity thresholds, since the algorithm always probe all the possible combinations of pairs.

To observe the differences among the three filtering algorithms, we need to look at the results on DBLP-Large (figure 5.4). Here we can make following observations. We first observe that TSUFFIX is the most efficient algorithm at thresholds 0.8, 0.85, 0.9. However, at threshold 0.95 TPREFIX has approximately the same running time as TSUFFIX, i.e., ca. 4.2 seconds.

We also observe a general trend related to this point: the speed-up of TSUFFIX against TPREFIX decreases with the increase of the similarity threshold. In particular, at threshold 0.8 TSUFFIX can achieve 2.5x speed-up against TPREFIX, at threshold 0.85 it achieves 2.4x speed-up, at threshold 0.9 only 1.6x, and, as we have seen, at threshold 0.95 it achieves no speed-up. This could be due to the following reasons [2]:

1. In TSUFFIX the overheads caused by positional and suffix filtering are greater with a high similarity threshold. This is because with a high similarity threshold the result is small and easy to compute also for TPREFIX.

2. With a low similarity threshold prefixes are longer, and, as a result, inverted lists grow very quickly. Therefore, since inverted lists are longer, the prefix filtering method generates a large quantity of candidate pairs. This slows down the TPREFIX algorithm, since prefix filtering is the only filtering mechanism of TPREFIX, while TSUFFIX can also benefit from positional and suffix filtering.

Another observation on DBLP-Large experimental results is that TPOSITIONAL runs always a few seconds slower than TSUFFIX. Anyway, TPOSITIONAL achieves also a noticeable speed-up against TPREFIX at thresholds 0.8, 0.85 and 0.9. On the other hand, at threshold 0.95 it needs ca. 5.6 seconds to perform the join and gets slower than TPREFIX.

Finally, we observe that also if TSUFFIX generates a considerable smaller number of candidate pairs than TPOSITIONAL (see section 5.2), the difference in running time between the two is minor. This is because suffix filtering produces a notable overhead for pruning candidates, in particular due to the use of binary search.

[2] These points were already made by Xiao et al. [16, p. 139] to explain the differences in running time between non-temporal prefix filtering (ALL-PAIRS) and suffix filtering (PPJOIN) algorithms.

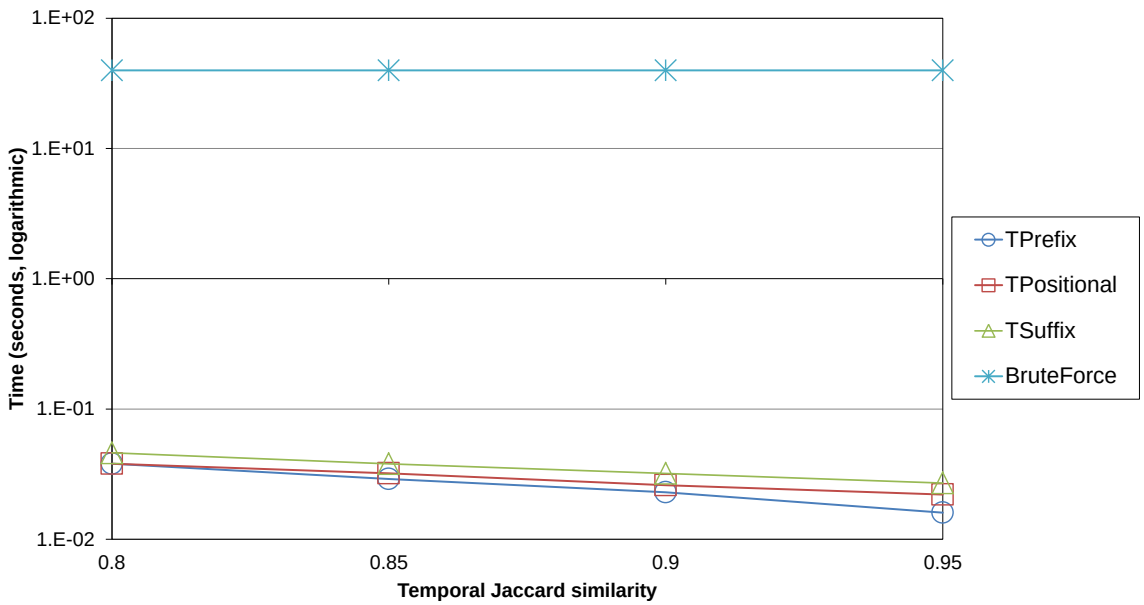

Figure 5.3: DBLP-Small, Time.

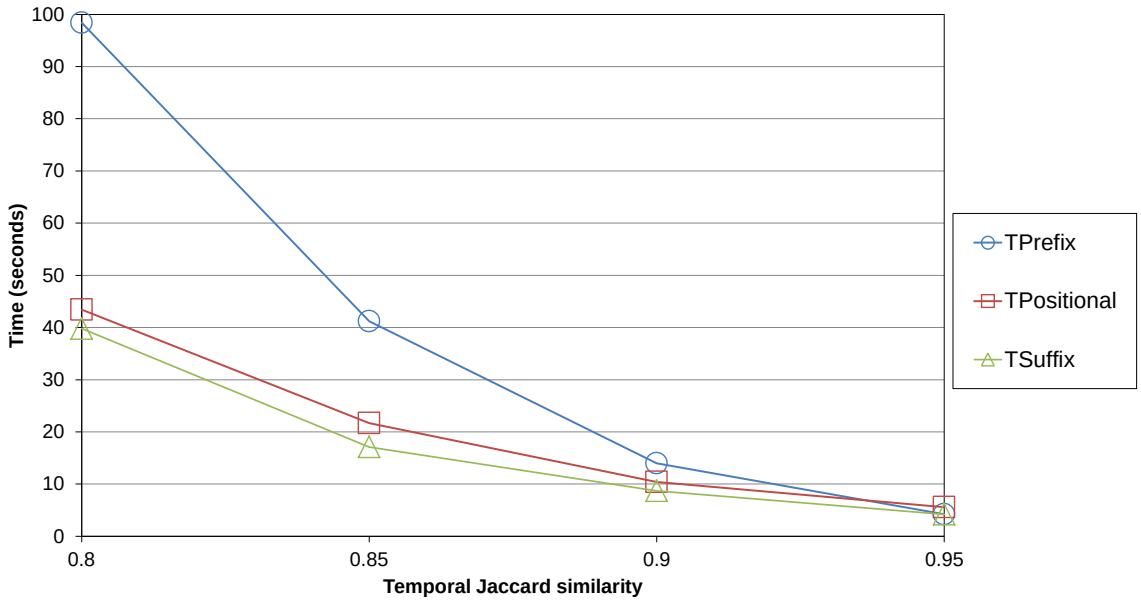

Figure 5.4: DBLP-Large, Time.

5.4 Scalability

In order to evaluate how the algorithms scale, we run the algorithms on growing subsets of the DBLP-Large data set keeping threshold fixed to 0.8, and we measured running times. Figure 5.5 shows the results. Measures start on a data set with 100,000 tokens, which is then increased by 100,000 for any following measure.

We observe that the running time of the three filtering algorithms grows quadratically. Among them, TSUFFIX is the one that has the slower growth rate. TPOSITIONAL has a growth rate which is only a little higher than the one of TSUFFIX, while the difference with TPREFIX is considerable.

In addition, table 5.4 shows the candidate sizes for different data set sizes and includes also BRUTEFORCE[3]. Also in this experiment threshold is kept fixed to 0.8. Note that all values in this table are in thousands, except column "Data set size".

Data set size	BRUTEFORCE	TPREFIX	TPOSITIONAL	TSUFFIX	Result
100k	$4 \cdot 10^6$	669	194	16	13
200k	$20 \cdot 10^6$	2,381	665	42	34
300k	$45 \cdot 10^6$	5,171	1,451	89	70
400k	$80 \cdot 10^6$	9,676	2,693	153	121
500k	$125 \cdot 10^6$	15,864	4,387	221	173

Table 5.4: Scalability, Candidate sizes, $\theta = 0.8$. All values are in thousands, except column "Data set size".

[3]The candidate size of BRUTEFORCE for a data set of size n can be computed with the following formula: $\frac{n!}{(n-2)!\cdot 2}$.

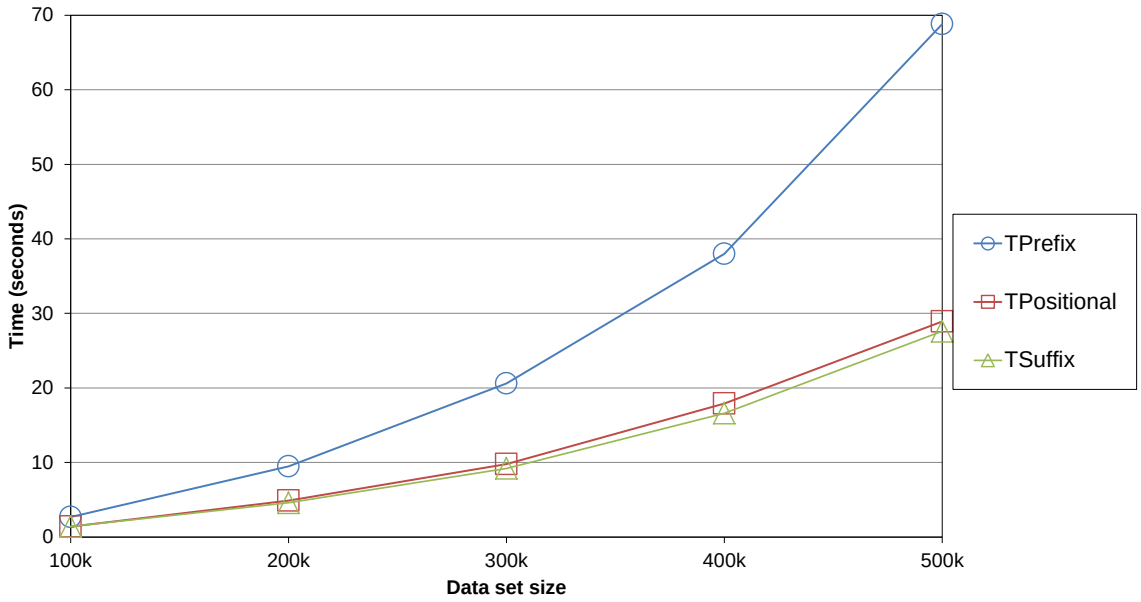

Figure 5.5: Scalability, Time, $\theta = 0.8$.

6 Conclusions and future work

In this thesis, we refined prefix filtering, positional filtering, and suffix filtering techniques to perform similarity joins utilizing also temporal information of records. Specifically, we implemented three algorithms that perform temporal similarity joins with a temporal Jaccard similarity threshold: TPREFIX is an algorithm that exploits prefix filtering for computing temporal similarity joins; TPOSITIONAL, which extends TPREFIX with positional filtering; and TSUFFIX, which further extends TPOSITIONAL with suffix filtering.

We showed that these algorithms improve considerably the performance of exact temporal similarity joins with temporal Jaccard when compared to the brute force approach. In particular, TSUFFIX proved to be the fastest among the three in our experiments, followed by TPOSITIONAL which proved to be only modestly slower.

Future work might address the implementation and test of the algorithms proposed in these thesis with different temporal models, and thus, with different ranges of the temporal weights w_a and w_d. The development of ad hoc temporal models to better exploit filtering techniques might also represent a research field. Finally, other similarity metrics for computing the join could be considered, as for instance temporal versions of overlap and cosine similarity functions.

Bibliography

[1] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling Up All Pairs Similarity Search. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 131–140, New York, NY, USA, 2007. ACM.

[2] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 5–, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Yueh-Hsuan Chiang, AnHai Doan, and Jeffrey F. Naughton. Modeling Entity Evolution for Temporal Record Matching. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1175–1186, New York, NY, USA, 2014. ACM.

[4] Yueh-Hsuan Chiang, AnHai Doan, and Jeffrey F. Naughton. Tracking Entities in the Dynamic World: A Fast Algorithm for Matching Temporal Records. *Proc. VLDB Endow.*, 7(6):469–480, February 2014.

[5] Edith Cohen and Martin Strauss. Maintaining Time-decaying Stream Aggregates. In *Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '03, pages 223–233, New York, NY, USA, 2003. ACM.

[6] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. Forward Decay: A Practical Time Decay Model for Streaming Systems. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 138–149, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate Record Detection: A Survey. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):1–16, January 2007.

[8] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String Similarity Joins: An Experimental Evaluation. *Proc. VLDB Endow.*, 7(8):625–636, April 2014.

[9] Pei Li, Xin Dong, Andrea Maurino, and Divesh Srivastava. Linking temporal records. *Proceedings of the VLDB Endowment*, 4(11):956–967, 2011.

[10] Gultekin Ozsoyoglu and Richard Thomas Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Trans. on Knowl. and Data Eng.*, 7(4):513–532, August 1995.

[11] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. Efficient Exact Edit Similarity Query Processing with the Asymmetric Signature Scheme. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1033–1044, New York, NY, USA, 2011. ACM.

[12] John F. Roddick and Myra Spiliopoulou. A Survey of Temporal Knowledge Discovery Paradigms and Methods. *IEEE Trans. on Knowl. and Data Eng.*, 14(4):750–767, July 2002.

[13] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can We Beat the Prefix Filtering?: An Adaptive Framework for Similarity Join and Search. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 85–96, New York, NY, USA, 2012. ACM.

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

[15] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-Join: An Efficient Algorithm for Similarity Joins with Edit Distance Constraints. *Proc. VLDB Endow.*, 1(1):933–944, August 2008.

[16] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient Similarity Joins for Near Duplicate Detection. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 131–140, New York, NY, USA, 2008. ACM.

[17] Mohamed Yakout, Ahmed K. Elmagarmid, Hazem Elmeleegy, Mourad Ouzzani, and Alan Qi. Behavior Based Record Linkage. *Proc. VLDB Endow.*, 3(1-2):439–448, September 2010.