Approximate String Processing

Article *in* Foundations and Trends in Databases · January 2011

DOI: 10.1561/1900000010 · Source: DBLP

Divesh Srivastava
AT&T
458 PUBLICATIONS **19,207** CITATIONS

Foundations and Trends® in
Databases
Vol. 2, No. 4 (2009) 267–402
© 2011 M. Hadjieleftheriou and D. Srivastava
DOI: 10.1561/1900000010

Approximate String Processing

By Marios Hadjieleftheriou and Divesh Srivastava

Contents

Foundations and Trends® in
Databases
Vol. 2, No. 4 (2009) 267–402
© 2011 M. Hadjieleftheriou and D. Srivastava
DOI: 10.1561/1900000010

Approximate String Processing

Marios Hadjieleftheriou[1] and Divesh Srivastava[2]

[1] *AT&T Labs - Research, 180 Park Ave, Florham Park, NJ, 07932, USA, marioh@research.att.com*
[2] *AT&T Labs - Research, 180 Park Ave, Florham Park, NJ, 07932, USA, divesh@research.att.com*

Abstract

One of the most important primitive data types in modern data processing is text. Text data are known to have a variety of inconsistencies (e.g., spelling mistakes and representational variations). For that reason, there exists a large body of literature related to approximate processing of text. This monograph focuses specifically on the problem of *approximate string matching*, where, given a set of strings S and a query string v, the goal is to find all strings $s \in S$ that have a user specified degree of similarity to v. Set S could be, for example, a corpus of documents, a set of web pages, or an attribute of a relational table. The similarity between strings is always defined with respect to a similarity function that is chosen based on the characteristics of the data and application at hand. This work presents a survey of indexing techniques and algorithms specifically designed for approximate string matching. We concentrate on inverted indexes, filtering techniques, and tree data structures that can be used to evaluate a variety of set based and edit based similarity functions. We focus on all-match and top-k flavors of selection and join queries, and discuss the applicability, advantages and disadvantages of each technique for every query type.

1

Introduction

Arguably, one of the most important primitive data types in modern data processing is strings. Short strings comprise the largest percentage of data in relational database systems, long strings are used to represent proteins and DNA sequences in biological applications, as well as HTML and XML documents on the Web. In fact this very monograph is safely stored in multiple formats (HTML, PDF, TeX, etc.) as a collection of very long strings. Searching through string datasets is a fundamental operation in almost every application domain. For example, in SQL query processing, information retrieval on the Web, genomic research on DNA sequences, product search in eCommerce applications, and local business search on online maps. Hence, a plethora of specialized indexes, algorithms, and techniques have been developed for searching through strings.

Due to the complexity of collecting, storing and managing strings, string datasets almost always contain representational inconsistencies, spelling mistakes, and a variety of other errors. For example, a representational inconsistency occurs when the query string is 'Doctors Without Borders' and the data entry is stored as 'Doctors w/o Borders'. A spelling mistake occurs when the user mistypes the query as 'Doctors

Witout Borders'. Even though exact string and substring processing have been studied extensively in the past and a variety of efficient string searching algorithms have been developed, it is clear that approximate string processing is fundamental for retrieving the most relevant results for a given query, and ultimately improving user satisfaction.

How many times have we posed a keyword query to our favorite search engine, only to be confronted by a search engine suggestion for a spelling mistake? In a sense, correcting spelling mistakes in the query is not a very hard problem. Most search engines use pre-built dictionaries and query logs in order to present users with meaningful suggestions. On the other hand though, even if the query is correct (or the search engine corrects the query) spelling mistakes and various other inconsistencies can still exist in the web pages we are searching for, hindering effective searching. Efficient processing of string similarity as a primitive operator has become an essential component of many successful applications dealing with processing of strings. Applications are not limited to the realm of information retrieval and *selection queries* only. A variety of other applications heavily depend on robust processing of *join queries*. Such applications include, but are not limited to, record linkage, entity resolution, data cleaning, data integration, and text analytics.

The fundamental *approximate text processing problem* is defined as follows:

Definition 1.1 (Approximate Text Matching). Given a text T and a query string v one desires to identify all substrings of T that have a user specified degree of similarity to v.

Here, the similarity of strings is defined with respect to a particular similarity function that is chosen based on specific characteristics of the data and application at hand. There exist a large number of similarity functions specifically designed for strings. All similarity functions fall under two main categories, *set based* and *edit based*. Set based similarity functions (e.g., Jaccard, Cosine) consider strings as sets of tokens (e.g., q-grams or words), and the similarity is evaluated with respect to the number, position and importance of common tokens. Edit based similarity functions (e.g., Edit Distance, Hamming) evaluate the similarity of strings as a function of the total number of edit operations that are necessary to convert one string into the other. Edit operations can be insertions, deletions, replacements, and transpositions of characters or tokens.

Approximate text processing has two flavors, online and offline. In the online version, the query can be pre-processed but the text cannot, and the query is answered without using an index. A survey on existing work for this problem was conducted by Navarro [54]. In the offline version of the problem the text is pre-processed and the query is answered using an index. A review of existing work for this problem was conducted by Chan et al. [16].

Here, we focus on a special case of the fundamental approximate text processing problem:

Definition 1.2 (Approximate String Matching). Given a set of strings S and a query string v, one desires to identify all strings $s \in S$ that have a user specified degree of similarity to v.

The approximate string matching problem (which is also referred to as the approximate dictionary matching problem in related literature) is inherently simpler than the text matching problem, since the former relates to retrieving strings that are similar to the query as a whole, while the latter relates to retrieving strings that *contain a substring* that is similar to the query. Clearly, a solution for the text matching problem will yield a solution for the string matching problem. Nevertheless, due to the simpler nature of approximate string matching, there is a variety of specialized algorithms for solving the problem that are faster, simpler, and with smaller space requirements than well-known solutions for text matching. The purpose of this work is to provide an overview of concepts, techniques and algorithms related specifically to the approximate string matching problem.

To date, the field of approximate string matching has been developing at a very fast pace. There now exists a gamut of specialized data structures and algorithms for a variety of string similarity functions and application domains that can scale to millions of strings and can provide answers at interactive speeds. Previous experience has shown that for most complex problems there is almost never a one size fits all solution. Given the importance of strings in a wide array of applications, it is safe to assume that different application domains will benefit from specialized solutions.

There are four fundamental primitives that characterize an indexing solution for approximate string matching:

- The similarity function: As already discussed, there are two types of similarity functions for strings, set based and edit based.
- String tokenization: Tokenization is the process of decomposing a string into a set of primitive components, called tokens. For example, in a particular application a primitive component might refer to a word, while in some other application a primitive component might refer to a whole sentence. There are two fundamental tokenization schemes, overlapping and non-overlapping tokenization.
- The query type: There are two fundamental query types, selections and joins. Selection queries retrieve strings similar to a given query string. Join queries retrieve all similar pairs of strings between two sets of strings. There are also two flavors of selection and join queries, all-match and top-k queries. All-match queries retrieve all strings (or pairs of strings) within a user specified similarity threshold. Top-k queries retrieve the k most similar strings (or pairs of strings).
- The underlying index structure: There are two fundamental indexing schemes, inverted indexes and trees. An inverted index consists of a set of lists, one list per token in the token universe produced by the tokenization scheme. A tree organizes strings into a hierarchical structure specifically designed to answer particular queries.

Every approximate string indexing technique falls within the space of the above parametrization. Different parameters can be used to solve a variety of problems, and the right choice of parameters — or combination thereof — is dependent only on the application at hand.

This work explains in detail the available choices for each primitive, in an effort to delineate the application space related to every choice.

For example, consider a relevant document retrieval application that uses cosine similarity and token frequency/inverse document frequency weights[1] to retrieve the most relevant documents to a keyword query. The application uses a set based similarity function, implying a word-based, non-overlapping tokenization for keyword identification, a clear focus on selection queries, and most probably an underlying inverted index on keywords. Notice that this particular application is not related to approximate matching of keywords. A misspelled keyword, either in the query or the documents, will miss relevant answers. Clearly, to support approximate matching of keywords, a relevant document retrieval engine will have to use a combination of primitives.

As another example, consider an application that produces query completion suggestions interactively, as the user is typing a query in a text box. Usually, query completion is based on the most popular queries present in the query logs. A simple way to enable query suggestions based on approximate matching of keywords as the user is typing (in order to account for spelling mistakes) is to use edit distance to match what the user has typed so far as an approximate substring of any string in the query logs. This application setting implies an edit based similarity, possibly overlapping tokenization for enabling identification of errors on a per keyword level, focus on selection queries, and either an inverted index structure built on string signatures tailored for edit distance, or specialized trie structures.

The monograph is organized into eight sections. In the first four sections we discuss in detail the fundamental primitives that characterize any approximate string matching indexing technique. Section 2 presents in detail some of the most widely used similarity functions for strings. Section 3 discusses string tokenization schemes. Section 4 gives a formal definition of the four primitive query types on strings. Finally, Section 5 discusses the two basic types of data structures used to answer approximate string matching queries. The next three sections are dedicated to specialized indexing techniques and algorithms for

[1] Token frequency is also referred to as term frequency.

approximate string matching. Section 6 discusses set based similarity algorithms using inverted indexes. Section 7 discusses set based similarity algorithms using filtering algorithms. Finally, Section 8 discusses edit based similarity algorithms using both inverted indexes and filtering algorithms. Section 9 concludes the monograph.

2

String Similarity Functions

There are two types of similarity functions for strings, set based and edit based. Set based similarity considers strings as sets of tokens and evaluates the similarity of strings with respect to the similarity of the corresponding sets. Edit based similarity considers strings as sequences of characters (or tokens) by assigning absolute positions to each character (or token) and evaluating string similarity with respect to the minimum number of edit operations needed to convert one sequence of characters (or tokens) into the other.

2.1 Edit Based Similarity

Edit based similarity is a very intuitive measure for strings. The similarity is determined according to the minimum number of edit operations needed to transform one string into another. Let Σ be an alphabet. Let string $s = \sigma_1 \cdots \sigma_\ell, \sigma_i \in \Sigma^*$. Primitive edit operations consist of insertions, deletions, and replacements of characters. An insertion $I(s,i,\sigma)$ of character $\sigma \in \Sigma$ at position i of string s results in a new string s' of length $\ell + 1, s' = \sigma_1 \cdots \sigma_{i-1} \sigma \sigma_i \cdots \sigma_\ell$. A deletion $D(s,i)$ removes character σ_i resulting in a new string s' of length $\ell - 1, s' = \sigma_1 \cdots \sigma_{i-1}\sigma_{i+1} \cdots \sigma_\ell$. Finally, a replacement $R(s,i,\sigma)$ replaces the character σ_i with character $\sigma \in \Sigma$ resulting in string $s' = \sigma_1 \cdots \sigma_{i-1}\sigma\sigma_{i+1} \cdots \sigma_\ell$, of the same length ℓ.

Definition 2.1 (Edit Distance). The edit distance $\mathcal{E}(s,r)$ between two strings s,r is the minimum number of primitive operations (i.e., insertions, deletions, and replacements) needed to transform s to r.

Edit distance is also known as Levenshtein distance.

For example, let $s =$‘1 Laptop per Child’ and $r =$ ‘One Laptop per Child’. Then $\mathcal{E}(s,r) = 3$ since the sequence of operations $I(s,1,‘n’), I(s,1,‘O’), R(s,3,’e’)$ results in string r, and there is no shorter sequence of operations that can transform s into r.

Clearly, edit distance is symmetric. The minimum edits needed to transform r into s is the inverse of that needed to transform s into $r(R(r,3,‘1’), D(r,1), D(r,1)$ in this example). In addition, edit distance satisfies the triangular inequality, i.e., $\mathcal{E}(s,r) \leq \mathcal{E}(s,t) + \mathcal{E}(t,r)$. This is easy to show using induction. Hence, edit distance is a metric.

The edit distance between two strings can be computed using dynamic programming in $O(|s|)$ space and $O(|s|^2)$ time for strings of length $|s|$, while the most efficient known algorithm requires $O(|s|)$ space and $O(|s|^2/log|s|)$ time instead [52]. Clearly computing the edit distance between strings is a very expensive operation. However, if the goal is to test whether the edit distance between two strings is within some threshold θ, then there exists a verification algorithm that runs in $O(\theta)$ space and $O(\theta|s|)$ time, which is very efficient for small thresholds θ. The idea is once more based on dynamic programming, but the verification algorithm tests only the entries with offset no more than θ from the diagonal in the dynamic programming matrix. The algorithm is shown as Algorithm 2.1.1, and a sample execution between the strings ‘1 Laptop’ and ‘One Laptop’ with $\theta = 2$ is shown in Table 2.1. The computation of the actual edit distance between the two strings (as opposed to simply verifying whether the edit distance is smaller than 2) could, in the worst case, require the full computation of the dynamic programming matrix, as opposed to the constrained verification algorithm.

Algorithm 2.1.1: EDIT DISTANCE VERIFICATION (s,r,θ)

```
**if** $||s| - |r|| > \theta$ : **return** false
Construct a table $T$ of 2 rows and $|r| + 1$ columns
**for** $j = 1$ **to** $\min(|r| + 1, 1 + \theta)$ : $T[1][j] = j - 1$
Set $m = \theta + 1$
**for** $i = 2$ **to** $|s| + 1$
  **for** $j = \min(1, i - \theta)$ **to** $\min(|r| + 1, i + \theta)$
    **do**
      $d_1 = (j < i + \theta)$ ? $T[1][j] + 1$ : $\infty$
      $d_2 = (j > 1)$ ? $T[2][j - 1] + 1$ : $\infty$
      $d_3 = (j > 1)$ ? $T[1][j - 1] +$
          $(s[i - 1] = r[j - 1])$ ? $0 : 1)$ : $\infty$
      $T[2][j] = \min(d_1, d_2, d_3)$
      $m = \min(m, T[2][j])$
  **if** $m > \theta$ : *return false*
  **for** $j = 0$ **to** $|r| + 1$ : $T[1][j] = T[2][j]$
**return** true
```

Table 2.1. Sample execution of the dynamic programming verification algorithm for edit distance with threshold $\theta = 2$. The algorithm returns false.

	∅	O	n	e		L	a	p	t	o	p
∅	0	1	2								
1	1	1	2	3							
	2	2	2	3	3						
L		3	3	3	4	3					
a			4	4	4	4	3				
p				5	5	5	4	3			
t					6	6	5	4	3		
o						7	6	5	4	3	
p							7	6	5	4	3

A plethora of extensions to the basic edit distance exist. For example, one can extend the primitive edit operations to include a transposition of two consecutive characters. This extended edit distance has been shown to cover most spelling mistakes made by humans in written text. One can also compute a normalized edit distance, defined as $\frac{\mathcal{E}(s,r)}{\max|s|,|r|}$, that takes into account the actual length of the strings, since it is more meaningful to allow larger edit distance thresholds for longer strings. Another extension is to compute a *weighted edit distance*, where each edit operation has its own weight. The weights can also depend on the particular character being inserted, deleted, or substituted. One can also compute a normalized version of weighted edit distance, which is defined as the minimum fraction $W(P)/|P|$, where P is a sequence of edit operations, $W(P)$ is the total weight of P, and $|P|$ is the length of P. More involved variations of edit distance allow whole substrings to be replaced or moved with a unit cost as a primitive operation. A similar idea is to allow the introduction of gaps with a unit cost dependent on the length of the gap, to enable alignment of strings. Finally, Hamming distance can be considered a special case of edit distance, where only replacement operations are allowed.

Each variation of edit distance is designed with a specific application in mind. For example, computing similarity of biological sequences, where mutations are more likely to appear as a variation of several consecutive, rather than individual, nucleotides, necessitates the introduction of primitive edit operations on tokens. Similarly, performing local alignment of DNA sequences where there might exist long sequences of low complexity or noise introduced by evolutionary mutation that should be ignored, necessitates the introduction of gaps as a primitive operation.

Edit distance has been shown to work very well for a variety of applications. Nevertheless, it does have some drawbacks that make it insufficient in certain scenarios. First, edit distance does not inherently discount mismatching substrings that might not be very important from a practical perspective, in some data domains. For example, consider very common words in the English language, like 'the'; the edit distance between 'Feed the Children' and 'Feed Children' is four, even though from a practical viewpoint the two strings are the same. The solution here is to use customized weighing schemes to alleviate these problems, but in this case the computation of edit distance becomes more expensive (efficient approximate solutions though do exist; for example see BLAST [1]). A second inherent drawback of edit distance is that, by construction, it is not amenable to word re-ordering. For example, the two strings 'One Laptop per Child' and 'One Child per Laptop' have edit distance twelve. Clearly, a better way to compare these two strings using edit distance is to first decompose them into individual words and compare each word separately first, then determine the similarity between the complete strings as a weighted function of matching words, their respective edit distance, and possibly their relative positions in the two strings. Of course, the cost of comparing the strings now becomes quadratic in the number of words in the strings.

2.2 Set Based Similarity

Set based similarity decomposes strings into sets using a variety of tokenization methods and evaluates similarity of strings with respect to the similarity of their sets. A variety of set similarity functions can be used for that purpose, all of which have a similar flavor: Each set element (or token) is assigned a weight and the similarity between the sets is computed as a weighted function of the tokens in the intersection and/or the complement of the intersection of the sets. The application characteristics heavily influence, first, the tokens to extract, second, the token weights, and third, the type of similarity function used.

Strings are modeled as sequences of characters, nevertheless a string can also be represented as a set or a multi-set of tokens, based on the tokenization function used. Let Λ be a token universe. Let $s = \lambda_1^s \lambda_2^s \cdots \lambda_m^s, \lambda_i^s \in \Lambda$ be a string consisting of a sequence of m tokens. This definition is a generalization of the definition of strings as a sequence of characters (recall that $s = \sigma_1 \cdots \sigma_\ell$). If the chosen tokenization returns each character in the string as an individual token, then the two definitions are the same, and $m = \ell = |s|$. In general, notice that $m \neq |s|$ (in other words, the number of tokens in s is not always equal to the number of characters in s). In the rest, for clarity we will always refer to the number of characters in s with $|s|$ and the number of tokens in s with $\|s\|_0$ (i.e., the L_0-norm of set s), irrespective of the context of the similarity function and the specific tokenization used.

Implicitly, each token in the sequence above is assigned a position in the string. A more explicit way of representing the string as a sequence of tokens is to use a set of token/position pairs $s = \{(\lambda_1^s, 1), \ldots, (\lambda_m^s, m)\}$. A weaker string representation is to sacrifice the positional information and only preserve the number of times each token appears in the string.

Definition 2.2 (Token Frequency). The token frequency $f_s(\lambda)$ is the number of occurrences of token λ in string s. When clear from context, we simply write $f(\lambda)$ to refer to the token frequency of λ in a certain string.

Using token frequencies a string can be represented as the set $s = \{(\lambda_1^s, f(\lambda_1^s)), \ldots, (\lambda_n^s, f(\lambda_n^s))\}$ (notice that $n \leq m$). Here the order of tokens is lost. We refer to this representation as a frequency-set of tokens. An even weaker representation is to discard the token frequency and consider strings as simple sets of tokens, i.e., $s = \{\lambda_1^s, \ldots, \lambda_n^s\}$. We differentiate between these three string representations as *sequences*, *frequency-sets*, and *sets*.

It should be stressed here that all three representations are explicitly defined to be sets of elements (as opposed to multi-sets or bags). Hence, in what follows, all intersection and union predicates operate on sets. Notice also that the most general representation of the three is sequences. When indexing sequences, one can easily disregard the positional information of the tokens and treat the strings either as frequency-sets or sets. Obviously, the particular interpretation of a string as a sequence, a frequency-set or a set has a *significant influence* on the semantics of the similarity between strings, with sequences being the most strict interpretation (strings not only have to agree on a large number of tokens being similar, but the tokens have to have similar positions within the string as well), and sets being the loosest (the similarity of strings depends only on the number of tokens shared, rather than the position or the multiplicity of those tokens).

Let $W : \Lambda \to \mathbb{R}^+$ be a function that assigns a positive real value as a weight of each token in Λ. The simplest function for evaluating the similarity between two strings is the weighted intersection of the token sequences.

Definition 2.3 (Weighted Intersection on Sequences). Let $s = \{(\lambda_1^s, 1), \ldots, (\lambda_m^s, m)\}$, $r = \{(\lambda_1^r, 1), \ldots, (\lambda_n^r, n)\}$, $\lambda_i^s, \lambda_i^r \in \Lambda$, be two sequences of tokens. The weighted intersection of s and r is defined as

$$\mathcal{I}(s,r) = \sum_{(\lambda,p) \in s \cap r} W(\lambda).$$

The intuition here is simple. If two strings share enough heavy tokens/position pairs or a large number of light token/position pairs then the strings are potentially very similar. Clearly, intersection is a symmetric similarity measure. This definition uses the absolute position of tokens within the sequences; if the common token does not appear in exactly the same position within the two strings then it is not included in the intersection. For example, the two sequences 'The Bill & Melinda Gates Foundation' and 'The Melinda & Bill Gates Foundation' have only four token/position pairs in common. One can extend the definition to consider edit based similarity, where the distance in the position of a common token between two strings is allowed to be within user specified bounds, instead of requiring it to be exactly the same.

Depending on application characteristics, it might be important to consider similarity of strings without regard for token positions. For example, when tokens are words and word order is not important (consider 'The Bill & Melinda Gates Foundation' and 'The Melinda & Bill Gates Foundation'), or when one is interested in evaluating similarity on substrings (consider 'American Red Cross' and 'Red Cross'). For that purpose weighted intersection can be defined on frequency-sets.

Definition 2.4 (Weighted Intersection on Frequency-sets). Let $s = (\lambda_1^s, f(\lambda_1^s)), \ldots, (\lambda_m^s, f(\lambda_m^s)), r = (\lambda_1^r, f(\lambda_1^r)), \ldots, (\lambda_n^r, f(\lambda_n^r)), \lambda_i^s, \lambda_i^r \in \Lambda$, be two frequency-sets of tokens. The weighted intersection of s and r is defined as

$$\mathcal{I}(s,r) = \sum_{\lambda \in s \cap r} \min(f_s(\lambda), f_r(\lambda)) W(\lambda).$$

Notice that this definition implicitly treats frequency-set intersection as intersection on bags of tokens, by virtue of the min operation. In other words, two frequency-sets do not have to agree both on the token and the exact frequency of that token, for the token to be considered in the intersection.

Finally, the definition can be modified to disregard token positions and multiplicity.

Definition 2.5 (Weighted Intersection on Sets). Let $s = \{\lambda_1^s, \ldots, \lambda_m^s\}, r = \{\lambda_1^r, \ldots, \lambda_n^r\}, \lambda_i^s, \lambda_i^r \in \Lambda$, be two sets of tokens. The weighted intersection of s and r is defined as

$$\mathcal{I}(s,r) = \sum_{\lambda \in s \cap r} W(\lambda).$$

Representing strings as sequences, frequency-sets or sets of tokens is application dependent and is applicable for all similarity functions introduced below. In the rest we refer to strings as sequences, which is the most general case. Extending the definitions to frequency-sets and sets is straightforward.

Notice that the definitions above do not take into account the weight or number of tokens that the two strings *do not have* in common (i.e., in the complement of their intersection). In certain applications, it is required for strings to have similar lengths (either similar number of tokens or similar total token weight). One could use various forms of normalization to address this issue. A simple technique is to divide the weighted intersection by the maximum sequence weight.

Definition 2.6 (Normalized Weighted Intersection). Let $s = \lambda_1^s \cdots \lambda_m^s, r = \lambda_1^r \cdots \lambda_n^r$ be two sequences of tokens. The normalized weighted intersection of s and r is defined as

$$\mathcal{N}(s,r) = \frac{\|s \cap r\|_1}{\max(\|s\|_1, \|r\|_1)},$$

where $\|s\|_1 = \sum_{i=1}^{\|s\|_0} W(\lambda_i^s)$ (i.e., the L_1-norm of token sequence s).

One could also simply normalize by the length of the strings $\max(|s|,|r|)$ or even by the number of tokens in the strings $\max(n,m)$.

A formulation that normalizes by the weight of tokens in the union of the two strings is the Jaccard similarity.

Definition 2.7 (Jaccard Similarity). Let $s = \lambda_1^s \cdots \lambda_m^s$, $r = \lambda_1^r \cdots \lambda_n^r$ be two sequences of tokens. The Jaccard similarity of s and r is defined as

$$\mathcal{J}(s,r) = \frac{\|s \cap r\|_1}{\|s \cup r\|_1} = \frac{\|s \cap r\|_1}{\|s\|_1 + \|r\|_1 - \|s \cap r\|_1}.$$

Here, the similarity between two strings is normalized by the total weight of the union of their token sets. The larger the weight of the tokens that the two strings do not have in common is, the smaller the similarity becomes. The similarity is maximized (i.e., becomes equal to one) only if the two sequences are the same. Jaccard similarity is a metric.

One can also define a non-symmetric notion of Jaccard similarity, commonly referred to as Jaccard containment.

Definition 2.8 (Jaccard Containment). Let $s = \lambda_1^s \cdots \lambda_m^s, r = \lambda_1^r \cdots \lambda_n^r$ be two sequences of tokens. The Jaccard containment of s and r is defined as

$$\mathcal{J}_c(s,r) = \frac{\|s \cap r\|_1}{\|s\|_1}.$$

The Jaccard containment quantifies the containment of set s in set r. Jaccard containment is maximized if and only if $s \subseteq r$.

A related set similarity function is the Dice similarity.

Definition 2.9 (Dice Similarity). Let $s = \lambda_1^s \cdots \lambda_m^s, r = \lambda_1^r \cdots \lambda_n^r$ be two sequences of tokens. The Dice similarity of s and r is defined as

$$\mathcal{D}(s,r) = \frac{2\|s \cap r\|_1}{\|s\|_1 + \|r\|_1}.$$

Dice is maximized if and only if the two sequences are the same.

Another extension of weighted intersection that takes into account the total weight of each token sequence is the cosine similarity. Cosine similarity is the inner product between two vectors. Each token sequence can be represented conceptually as a vector in the high dimensional space defined by the token universe Λ (or the cross-product $\Lambda \times \mathbb{N}$ for token/position pairs). For example, set s can be represented as a vector of dimensionality $|\Lambda|$, where each vector coordinate λ is $W(\lambda)$ if $\lambda \in s$ and zero otherwise. This representation is commonly referred to as the *vector space model*.

Definition 2.10 (Cosine Similarity). Let $s = \lambda_1^s \cdots \lambda_m^s, r = \lambda_1^r \cdots \lambda_n^r$ be two sequences of tokens. The cosine similarity of s and r is defined as

$$\mathcal{C}(s,r) = \frac{(\|s \cap r\|_2)^2}{\|s\|_2 \|r\|_2},$$

where $\|s\|_2 = \sqrt{\sum_{i=1}^{\|s\|_0} W(\lambda_i^s)^2}$ (i.e., the L_2-norm of token sequence s).

Cosine similarity is maximized if and only if the two sequences are the same.

Clearly, Normalized Weighted Intersection, Jaccard, Dice, and Cosine similarity are strongly related in a sense that they normalize the similarity with respect to the weight of the token sequences. Hence, there is no functional difference between those similarity functions. The only difference is semantic and which function works best depends heavily on application and data characteristics.

An important consideration for weighted similarity functions is the token weighing scheme used. The simplest weighing scheme is to use unit weights for all tokens. A more meaningful weighing scheme though, should assign large weights to tokens that carry larger information content. As usual, the information content of a token is application and data dependent. For example, a specific sequence of characters might be a very rare word in the English language but a very popular occurrence in non-coding DNA sequences, or a common sequence of phonemes in Greek. Hence, a variety of weighing schemes have been designed, with a variety of application domains in mind.

The most commonly used weighing scheme for text processing is based on inverse document frequency weights. Given a set of strings S and a universe of tokens Λ, the document frequency $df(\lambda), \lambda \in \Lambda$ is the number of strings $s \in S$ that have at least one occurrence of λ. The inverse document frequency weight $idf(\lambda)$ is

Definition 2.11 (Inverse Document Frequency Weight). Let S denote a collection of strings and $df(\lambda), \lambda \in \Lambda$, the number of strings $s \in S$ with at least one occurrence of λ. The inverse document frequency weight of λ is defined as:

$$idf(\lambda) = \log\left(1 + \frac{|S|}{df(\lambda)}\right).$$

Alternative definitions of *idf* weights are also possible. Nevertheless, they all have a similar flavor. The *idf* weight is related to the likelihood that a given token λ appears in a random string $s \in S$. Very frequent tokens have a high likelihood of appearing in every string, hence they are assigned small weights. On the other hand, very infrequent tokens have a very small likelihood of appearing in any string, hence they are assigned very large weights. The intuition is that two strings that share a few infrequent tokens must have a large degree of similarity.

Custom weighing schemes are more appropriate in other applications. A good example is the availability of expert knowledge regarding the importance of specific tokens (e.g., in biological sequences). Another example is deriving weights according to various language models.

A problem with using set based similarity functions is that by evaluating the similarity between sets of tokens we lose the ability to identify spelling mistakes and inconsistencies on a sub-token level. Very similar tokens belonging to different strings are always considered as a mismatch by all aforementioned similarity functions. One way to alleviate this problem is to tokenize strings into overlapping tokens, as will be discussed in more detail in Section 3. To alleviate some of the problems associated with edit based and set based similarity functions, hybrid similarity functions based on combinations thereof have also been considered. A combination similarity function is derived by defining the edit distance between sets of tokens (rather than sequences of characters). In this scenario, the goal is to compute the sum of weighted primitive operations needed to convert one set of tokens into another. A primitive operation can be a deletion, an insertion, or a replacement of a token. The weight of each operation might depend on the actual weight of the token being inserted or deleted, or the weight and the actual edit distance between the token being replaced and the token replacing it. For example consider the two strings 'Feed the Children' and 'Food Child'. From a practical viewpoint, the two strings are very similar. From a set based similarity perspective the two strings do not have any words in common (if stemming is not performed). From an edit distance perspective the edit distance between the two strings is very large. A combination distance would consider inserting the popular word 'the' with a very small cost (e.g., proportional to the *idf* weight of the word), and replacing words 'Food' and 'Child', that are within edit distance 2 and 3 from the respective 'Feed' and 'Children' using a cost proportional to the actual edit distance.

2.3 Related Work

The edit distance concept was originally introduced by Levenshtein [47]. Wagner and Fischer [70] introduced the first algorithm for computing edit distance with time and space complexity of $O(|s||r|)$, where $|s|$ and $|r|$ are the lengths of the two strings. Cormen et al. [25] presented a space-efficient algorithm with space complexity $O(\max\{|s|,|r|\})$. The fastest edit distance algorithm, proposed by Masek and Patterson [52], requires $O(|s|^2/\log|s|)$ time using a split-and-merge strategy to partition the problem. The edit distance verification algorithm was proposed by Ukkonen [66]. There is a very rich literature of algorithms for computing variations of the basic edit distance measure. One of the first dynamic programming algorithms for aligning biological sequences was proposed by Needleman and Wunsch [55]. A variation of this algorithm that uses a substitution matrix and gap-scoring was proposed by Smith and Waterman [61]. Edit distance with block moves was proposed by Tichy [65]. Edit distance with reversals was discussed by Kececioglu and Sankoff [41]. Edit distance allowing swaps has been studied by Amir et al. [2] and Lee et al. [42]. More recently, Cormode and Muthukrishnan [26] introduced a variation of edit distance allowing sub-strings to be moved with smaller cost. Chaudhuri et al. [18] introduced a weighted edit distance for sets of tokens, based on the *idf* weights of tokens. Approximating edit distance has also been well studied in the literature of computer algorithms. Bar-Yossef et al. [9] showed that an $O(n^{3/7})$ approximation of edit distance can be obtained in linear time. Ostrovsky and Rabani [56] proved that edit distance can be probabilistically well preserved by Hamming distance after projecting the strings onto randomly selected subspaces. Andoni and Onak [3] extended this idea by reducing both the space and time complexities. The problem of approximate string matching using set based similarity functions has received wide attention from the information retrieval and core database communities. A detailed analysis and history of set based similarity in information retrieval was conducted by Baeza-Yates and Ribeiro-Neto [8].

3

String Tokenization

The type of string tokenization is a crucial design choice behind any string processing framework. There are two fundamental tokenization approaches: Non-overlapping and overlapping tokens. Non-overlapping tokens are better for capturing similarity between short/long query strings and long data strings (or documents) from a *relevant document retrieval* perspective. Overlapping tokens are better at capturing similarity of strings in the presence of spelling mistakes and inconsistencies on a sub-token level.

3.1 Non-overlapping Tokens

The most basic instantiation of non-overlapping tokenization is to consider tokens on word boundaries, sentences or any other natural boundary depending on the particular domain of strings. For example, the string $s =$ 'Doctors Without Borders' would be decomposed into the token set $T = \{`Doctors', `Without', `Borders'\}$.

The similarity between two strings is evaluated according to the similarity of their respective token sets. Notice that in this example minor inconsistencies or spelling mistakes on a word level will significantly affect the similarity of strings. For instance, the Jaccard similarity between strings ‘Doctors Without Borders’ and ‘Doctor Witout Border’ is zero, even though the two strings are practically the same. In practice, the most common solution to such problems is, first, to use stemming on the queries and the data, and second, to correct spelling mistakes using a dictionary search. Neither of these solutions might be applicable in certain situations. For example, for mathematical equations or source code, neither stemming nor a dictionary might always make sense; for business names, stemming might result in unwanted side effects (e.g., ‘Feed the Children’ becomes ‘food child’ and can result in many irrelevant answers).

Non-overlapping tokenization results in token sets with storage requirements equal or smaller than the storage size of the string itself. The token set can be stored explicitly or, in some cases, hashing can be used to represent tokens as integers.

3.2 Overlapping Tokens

In overlapping tokenization the idea is to extract all possible substrings of a given length q of string s. The resulting substrings are more commonly referred to as q-grams.

Definition 3.1 (Set of positional q-grams). Let $s = \sigma_1 \cdots \sigma_\ell$, $\sigma_i \in \Sigma$, be a string consisting of ℓ characters. The set of positional q-grams $G_q(s)$ consists of all q-length substrings of s:

$$G_q(s) = \{(\sigma_1 \cdots \sigma_q, 1), (\sigma_2 \cdots \sigma_{q+1}, 2), \ldots, (\sigma_{\ell-q+1} \cdots \sigma_\ell, \ell - q + 1)\}.$$

For example, let string $s =$ ‘UNICEF’ and $q = 3$. The resulting set of 3-grams is $G_3(s) =${(‘UNI’, 1), (‘NIC’, 2), (‘ICE’, 3), (‘CEF’, 4)}.

Notice that for strings with length smaller than q the q-gram set is empty. For simplicity of implementation, when working with q-grams, the q-grams are extracted over strings extended with a special beginning and ending character $\# \notin \Sigma$. For example, for $q = 4$ the string ‘WWF’ is extended to ‘###WWF###’ and G_4(‘WWF’) ={(‘###W’, 1), (‘##WW’, 2), (‘#WWF’, 3), ('WWF#', 4), ('WF##', 5), ('F###', 6)}. This representation also has the advantage that the beginning and ending characters are represented explicitly using their own unique q-grams, hence capturing more accurately the beginning and ending of the string.

One can also define generalizations to sets of grams of various sizes (as opposed to fixed length q-grams); for example, variable length grams, or all grams of length one up to length q. The bigger the gram set is, the larger the information captured by the grams becomes. The similarity of gram sets can be assessed using any set similarity function. The advantage of gram sets is that they can be used to evaluate similarity of strings on a substring level, where small inconsistencies or spelling mistakes do not affect the similarity of the gram sets significantly.

On the other hand, since representing strings as gram sets increases the representation size of the string, gram sets have the drawback of significantly larger space requirements, even when hashing is used. In addition, given the increased representation size, the evaluation of the respective similarity functions becomes expensive, especially for long strings.

4

Query Types

There are two fundamental query types in string processing: Selections and Joins. There are two fundamental query strategies: All-matches and Top-k matches.

4.1 Selection Queries

All-match selection queries return all data strings whose similarity with the query string is larger than or equal to a user specified threshold.

Definition 4.1 (All-Match Selection Query). Given a string similarity function Θ, a set of strings S, a query string v, and a positive threshold θ, identify the answer set

$$A = \{s \in S : \Theta(v,s) \geq \theta\}.$$

Top-k selection queries return, among all strings in the data, the k strings with the largest similarity to the query.

Definition 4.2 (Top-*k* Selection Query). Given a string similarity function Θ, a set of strings S, a query string v, and a positive integer k, identify the answer set A, s.t. $|A| = k$ and $\forall s \in A, s' \in S \setminus A : \Theta(v,s) \geq \Theta(v,s')$.

Top-k queries are very useful in practice since in many applications it is difficult to decide in advance a meaningful threshold θ for running an all-match query.

Clearly, all-match queries are easier to evaluate than top-k queries, given that a cutoff similarity threshold for top-k queries cannot be decided in advance, making initial pruning of strings difficult. Nevertheless, once k good answers have been identified (good in a sense that the k-th answer has similarity sufficiently close to the correct k-th answer) top-k queries essentially degenerate to all-match queries. Query answering strategies typically try to identify k good answers as fast as possible and subsequently revert to all-match query strategies.

4.2 Join Queries

Given two sets of strings and a user specified threshold, all-match join queries return all pairs of strings in the cross product of the two sets, with similarity larger than or equal to the threshold.

Definition 4.3 (All-Match Join Query). Given a string similarity function Θ, two sets of strings S, R, and a positive threshold θ, identify the answer set

$$A = \{(s,r) \in S \times R : \Theta(s,r) \geq \theta\}.$$

Top-k join queries return the k pairs with the largest similarity among all pairs in the cross product.

Definition 4.4 (Top-*k* Join Query). Given a string similarity function Θ, two sets of strings S, R, and a positive integer k, identify the answer set A s.t. $|A| = k$ and $\forall(s,r) \in A$ and $\forall(s',r') \in (S \times R) \setminus A : \Theta(s,r) \geq \Theta(s',r')$.

5

Index Structures

There are three major data structures used for indexing strings in order to answer similarity queries: the Inverted Index, Tries, and B-trees. Most algorithms are based on these three data structures that are explained in detail next.

5.1 Inverted Indexes

Consider a collection of strings. Each string is represented as a sequence of tokens. An example is shown in Table 5.1. Clearly, one can easily invert this representation by constructing one set per token (instead of one set per string), where each token set consists of all the strings containing this token at a specified position.

Definition 5.1 (Inverted List). Given a token $\lambda \in \Lambda$ the inverted list $L(\lambda)$ is the set of all strings $s \in S$ s.t. $\lambda \in s$.

An inverted index is a collection of inverted lists, one list per token in Λ. Storing the actual strings in the lists results in increased space requirements, especially for long strings, since each string is duplicated as many times as the tokens it consists of. In most practical cases, duplicating the strings is infeasible, and for that reason the lists usually contain only the string identifiers. The only advantage of storing the actual strings in the lists is that it eliminates the need of at least one extra read operation for retrieving the actual string when it is needed for further processing (e.g., reporting it to the user or for computing a ranking weight). In applications with critical performance requirements, duplicating strings might be acceptable (this is a typical performance/space tradeoff). In what follows we assume that only the string identifiers are contained in the lists, and, in order to avoid confusion, strings are denoted with lowercase characters (e.g., s) and string identifiers using double dots (e.g., $\ddot{s}$).

Notice that an inverted index on frequency-sets forfeits the positional information by replacing it with frequency information (and as a result reducing the size of the inverted index by collapsing multiple entries of a given string identifier within an inverted list into one entry). Finally, an inverted index on sets forfeits the frequency information as well. Furthermore, one can extend the information stored with each inverted list entry as well. For example, one can include the position of the string within a document or the column and table containing the string in a relational database. The *addressing granularity* stored along with each inverted list entry comes at the cost of additional storage space per list. Alternatively, one can retrieve this information by using the string identifier to query a separate index for retrieving the actual string and other associated information, on demand.

The inverted representation of the strings in Table 5.1 is shown in Table 5.2 after stemming has been performed on the tokens and

Table 5.1. A collection of strings represented as a collection of sets of tokens.

	Tokens			
String	1	2	3	4
s_1	‘Children’s’	‘Food’	‘Fund’	
s_2	‘One’	‘Laptop’	‘per’	‘Child’
s_3	‘Feed’	‘the’	‘Children’	
s_4	‘International’	‘Children’s’	‘Found’	
s_5	‘UNICEF’			

Table 5.2. An inverted representation of the strings in Table 5.1 as inverted lists. It is assumed that common stop words and special characters have been removed, stemming of words has occurred, all tokens have been converted to lowercase, and strings are considered as sequences of tokens (resulting in string-identifier/token-position pairs).

Token	Inverted List
child	$(\ddot{s}_1,1),(\ddot{s}_2,4),(\ddot{s}_3,3),(\ddot{s}_4,2)$
food	$(\ddot{s}_1,2),(\ddot{s}_3,1)$
found	$(\ddot{s}_4,3)$
fund	$(\ddot{s}_1,3)$
international	$(\ddot{s}_4,1)$
one	$(\ddot{s}_2,1)$
laptop	$(\ddot{s}_2,2)$
unicef	$(\ddot{s}_5,1)$

common stop words have been eliminated. The two representations are equivalent. With the first representation one can efficiently determine all tokens contained in a given string (by simply scanning the set corresponding to that string), while with the second representation one can efficiently identify all strings containing a specific token (by simply scanning the set corresponding to that token).

The advantage of the inverted index is that it can be used to answer union and intersection queries very efficiently. Given a query string, represented as a set of tokens, one can find all the strings containing at least one of the query tokens by simply taking the union of inverted lists corresponding to these tokens. For example, given query $q = \{$‘Children’, ‘International’$\}$, the union of lists ‘child’ and ‘international’ results in string identifiers $\ddot{s}_1, \ddot{s}_2, \ddot{s}_3, \ddot{s}_4$. To find all the strings containing all of the query tokens one has to take the intersection of the lists corresponding to the query tokens (and hence identify all data strings that are super sets of the query string). For example the intersection of lists ‘child’ and ‘international’ results is string identifier $\ddot{s}_4$. Hybrid list intersection strategies can be used to evaluate more involved string similarity functions, as explained in detail in Section 6.

5.2 Trees

5.2.1 Tries

A *trie* is a multi-way tree specifically designed for indexing strings in main memory. Given a set of strings S, the trie consists of one path per distinct string $s \in S$, of length $|s|$ where each node in the path corresponds to each character in s. Strings with the same prefixes share the same paths. The leaf nodes of the paths store pointers to the actual data (string identifiers, pointers to the location of the strings in a text document, the table/column/row in a relational database, etc.). For example, part of the trie corresponding to the strings of Table 5.1 is shown in Figure 5.1. For static data, it is possible to compress a trie by merging multiple nodes of the trie into a single node (using multiple characters as the label of that node). This results in a Patricia trie (an example is shown in Figure 5.2).

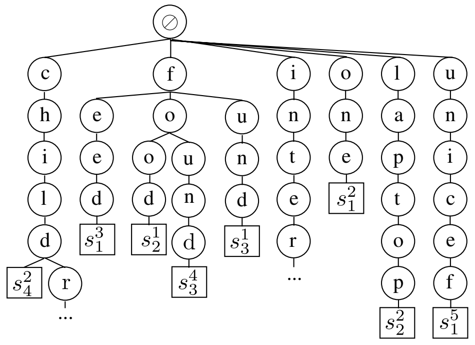


Fig. 5.1 The trie for the strings in Table 5.1.

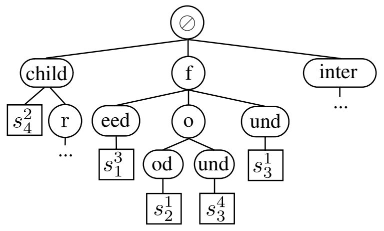


Fig. 5.2 The Patricia trie corresponding to part of the trie of Figure 5.1.

Searching the trie for an exact match is easy. Given a query string $v = \sigma_1 \cdots \sigma_\ell$ the trie is traversed from the root toward the leaves by sequentially following the nodes labeled σ_1 through σ_ℓ. If the search stops before all ℓ characters have been examined, then v does not exist in the data. Clearly, the trie has excellent performance (linear in the length of the query) for finding exact matches or prefixes of the query. The trie cannot be used efficiently for substring matching (i.e., for finding data strings that have v as a substring). Notice that the fanout at every level of the trie can, in the worst case, be equal to $|\Sigma|$.

Several algorithms have been proposed for extending tries to answer string similarity queries using edit distance, as will be discussed in Section 8.

5.2.2 Suffix Trees

A *suffix tree* is a data structure based on tries that indexes all the suffixes of all the strings $s \in S$. A string $s = \sigma_1 \cdots \sigma_m$ has a total of m suffixes $\sigma_1 \cdots \sigma_m, \sigma_2 \cdots \sigma_m, \ldots, \sigma_m$. By building a trie over all suffixes in S a suffix trie is produced.

The advantage of the suffix trie is that it can be used to find substring matches very efficiently. Given a query v, a traversal of the suffix trie will produce all strings $s \in S$ that contain v as a substring. The disadvantage is that the suffix trie consumes significantly more space than a simple trie. Moreover, the construction algorithm of the suffix trie has complexity $O(N^2)$, where N is the total length of the strings in S (i.e., $N = \sum_{s \in S} |s|$), which is prohibitively large for long strings.

The size of the suffix trie can be reduced by collapsing all paths consisting of nodes with only one child into a single node, similar to a Patricia trie. The resulting Patricia suffix trie is more commonly called a suffix tree. An example is shown in Figure 5.3.

The suffix tree has two advantages. First, the overall size of the structure can be reduced to $O(N)$, or linear in the length of the strings to be indexed. Second, the suffix tree construction algorithm also requires $O(N)$ time and space, provided that the tree fits entirely in main memory. (Suffix tries are more expensive to construct since one node for every character of every suffix of every string has to be

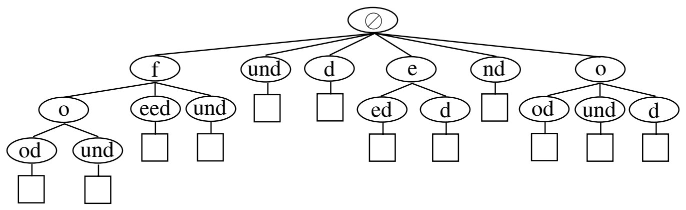


Fig. 5.3 The suffix tree for the strings ‘food’, ‘feed’, ‘fund’, and ‘found’.

instantiated in the worst case, as opposed to one node for every string in the suffix tree in the worst case.)

The size of the suffix tree can be reduced even further by constructing what is called a suffix array. The strings in S are concatenated into one large string (using a special terminating character $\# \notin \Sigma$ for every string) and the position of every suffix is sorted in an array according to the lexicographic ordering of the suffix. The resulting sorted array of positions can be searched using binary search on the suffix pointed to by each element in the array. The size of the suffix array (since it contains only suffix positions) is very small. The drawback of the suffix array is that the data strings need to be readily available for comparison purposes, resulting in accesses with poor locality of reference. Advanced algorithms for improving the efficiency of suffix arrays have been proposed, but the structure eventually resembles an inverted index built over all suffixes in S.

Suffix tries, suffix trees, and suffix arrays were designed to be used primarily for main memory processing. Fairly recently, a variety of alternatives have been specifically designed for external memory processing. Cache-conscious alternatives have also been proposed.

5.2.3 B-trees

B-trees are used for indexing one-dimensional data in secondary storage. A B-tree is a multiway tree that contains n values $k_1 \leq \cdots \leq k_n$ that act as comparison keys and $n+1$ pointers $p_1, \ldots, p_{n+1}$, per node. Pointer p_i points to a child node that is the root of a sub-tree containing all data values that lie in the interval $(k_{i-1}, k_i]$. Data values

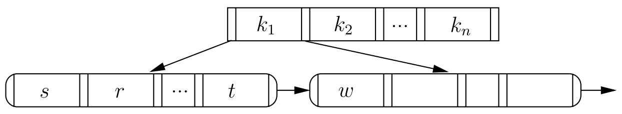


Fig. 5.4 A B-tree data structure for strings.

(or pointers to the actual data values) are stored in the leaves of the tree. A simple B-tree structure is shown in Figure 5.4.

Assume that data values and keys are strings sorted in lexicographic order. Given a query string v an exact string match query can be answered by starting at the root of the B-tree and following the pointer to the next level corresponding to the largest key s s.t. $v \leq s$. We continue iteratively until we reach a leaf node. A simple scan of the leaf node reveals whether v exists or not. An optimized specialization of the B-tree specifically designed to index strings, called the string B-tree, organizes each index node as a trie built over the keys of the node (instead of a sorted list of keys), to reduce the computational cost. Specialized algorithms for answering approximate string matching queries using B-trees will be covered in Section 8.

5.3 Related Work

Witten et al. [71], Baeza-Yates and Ribeiro-Neto [8], and Zobel and Moffat [79] present very detailed discussions on inverted indexes. Baeza-Yates and Ribeiro-Neto [8] also discuss suffix trees and suffix arrays. Puglisi et al. [59] discuss all varieties of suffix array construction algorithms. Comer [24] gives a detailed explanation of B-trees. The string B-tree was proposed by Ferragina and Grossi [30].

6

Algorithms for Set Based Similarity Using Inverted Indexes

In this section we will cover evaluation of set based similarity functions. We will cover algorithms for selection and join queries. The brute-force approach for evaluating selection queries has linear cost (computes all pairwise similarities between the query string and the data strings), while the brute-force approach for join queries has quadratic cost (computes all pairwise similarities in the cross product of the two datasets). Straightforwardly, big savings in computation cost can be achieved by using an index structure (resulting in a typical computation cost versus storage cost tradeoff).

6.1 All-Match Selection Queries

An all-match selection query returns all the strings in the dataset with similarity to the query string that is larger than or equal to a user specified threshold θ.

All of the set based similarity functions introduced in Section 2.2 can be computed easily using an inverted index. First, the data strings are tokenized and an inverted index is build on the resulting tokens, one list per token (an example using word tokens is shown in Tables 5.1

and 5.2). If the strings are treated as sequences, then each list element is a string-identifier/token-position pair. If the strings are treated as frequency-sets each list element is a string-identifier/token-frequency pair. Finally, if the strings are treated as sets, each list element is simply a string identifier. In what follows we consider algorithms for sequences only, since sequences are the most general representation of strings. The concepts and algorithms can be straightforwardly modified to work on frequency-set and sets.

Depending on the algorithm used to evaluate the chosen similarity function, the lists can be stored sequentially as a flat file, using a B-tree or a hash table. They can also be sorted according to string identifiers or any other information stored in the lists (e.g., the L_p-norm of the strings as will become clear shortly).

All-match selection queries can be answered quite efficiently using an inverted index. To answer the query, first the query string is tokenized using exactly the same tokenization scheme that was used for the data strings. The data strings satisfying the similarity threshold need to have at least one token in common with the query, hence only the token lists corresponding to query tokens need to be involved in the search. This results in significant pruning compared to the brute-force algorithm.

6.1.1 Lists Sorted in String Identifier Order

Let S be a set of strings over which we build an inverted index. Let $v = \lambda_1^v \cdots \lambda_m^v, \lambda_i^v \in \Lambda$ be a query string and $L_v = \{L(\lambda_1^v), \dots, L(\lambda_m^v)\}$ be the m lists corresponding to the query tokens. By construction, each list $L(\lambda_i^v)$ contains all string identifiers of strings $s \in S$ s.t. $\lambda_i^v \in s$. The simplest algorithm for evaluating the similarity between the query and all the strings in lists L_v, is to perform a `multiway merge` of the string identifiers in L_v, to compute all intersections $v \cap s$. This will directly yield the *Weighted Intersection* similarity. Whether we compute the weighted intersection on sequences, frequency-sets or sets depends on whether the initial construction of the inverted index was done on sequences, frequency-sets, or sets. Clearly, the `multiway merge` computation can be performed very efficiently if the lists are already sorted in increasing (or decreasing) order of string identifiers. Moreover, the algorithm has to exhaustively scan all lists in L_v in order to identify all strings with similarity exceeding threshold θ. Merging of sorted lists has been studied extensively with respect to external sorting algorithms (where the goal is to merge multiple sorted runs of files), and the algorithms used for our purposes are exactly the same. The `multiway merge` procedure for sequences is shown in Algorithm 6.1.1. The algorithm assumes that inverted lists are constructed for sequences and that each list is sorted primarily by token positions and secondarily by string identifiers. With this simple optimization the algorithm can directly seek to the first element in each list with token position equal to the token position in the query, and stop reading from a particular list once all elements with token position equal to that of the query have been

Algorithm 6.1.1: MULTIWAY MERGE(v,θ)

```
Tokenize the query: $v = \{(\lambda_1^v,1),\ldots,(\lambda_m^v,m)\}$
$M$ is a heap of $(\ddot{s},\mathcal{I}_s,i)$ tuples, sorted on $\ddot{s}$
$S$ is a stack of indexes $i$
$M \leftarrow \emptyset, S \leftarrow \{1,\ldots,m\}$
for $i \leftarrow 1$ to $m$
  do Seek to first element of $L(\lambda_i^v)$ with token position $p = i$
while $M$ is not empty or $S$ is not empty
  do while $S$ is not empty
       do $i = S$.pop
          $(\ddot{s},p) = L(\lambda_i^v)$.next
          if $p > i$ then continue
          if $\ddot{s} \in M$
            then $(\ddot{s},\mathcal{I}_s,j) \leftarrow M$.find$(\ddot{s})$
                 $\mathcal{I}_s{+}= W(\lambda_i^v)$ and $S \leftarrow i$
            else $M \leftarrow (\ddot{s},W(\lambda_i^v),i)$
     $(\ddot{r},\mathcal{I}_r,j) \leftarrow M$.pop
     if $\mathcal{I}_r \geq \theta$ then report $(\ddot{r},\mathcal{I}_r)$
     $S \leftarrow j$
```

examined. This optimization is irrelevant when indexing frequency-sets or sets. In addition, one could allow token positions to vary within some bounds of the corresponding token position in the query, in order to evaluate sequence based similarity measures that allow tokens to have slightly different positions in the query and the resulting data strings (e.g., recall the example 'The Bill & Melinda Gates Foundations' and 'The Melinda & Bill Gates Foundation'). An important detail that is missing from Algorithm 6.1.1 for simplicity, is that whenever a given token λ appears multiple times in different positions, the algorithm should scan list $L(\lambda)$ once, simultaneously for all positions. We omit this detail in all subsequent algorithms as well.

Computing *Normalized Weighted Intersection* is also straightforward, provided that the L_1-norm of every data string is stored in the token lists (see Definition 2.6). Hence, the lists store tuples string-identifier/token-position/L_1-norm per string for sequences, string-identifier/token-frequency/L_1-norm for frequency-sets, and string-identifier/L_1-norm for sets. Once again, if lists are already sorted in increasing order of string identifiers, a `multiway merge` algorithm can evaluate the normalized weighted intersection between the query and all relevant strings very efficiently. The algorithm for computing normalized weighted intersection essentially boils down to computing Weighted Intersection and hence is very similar to Algorithm 6.1.1. Similar arguments hold for Jaccard, Dice and Cosine similarity.

A number of optimizations are possible in case of *unit weights* (i.e., when $\forall \lambda \in \Lambda, W(\lambda) = 1$). For unit weights, the problem of identifying strings with similarity exceeding the query threshold θ is equivalent to the problem of identifying strings that appear in at least θ lists (since all the lists have the same weight, it does not matter which list a string appears in). In this case, in every iteration of the `multiway merge` algorithm, first we pop all occurrences of the top element of the heap. If that element occurs at least θ times we report it. Otherwise, assume that it appears a total of x times. We pop from the heap another $\theta - 1 - x$ elements, identify the new top element of the heap, let it be $\ddot{r}$, and directly skip to the first element $\ddot{s} \geq \ddot{r}$ in every token list (we can identify such elements using binary search on the respective lists). Finally, we insert the new top elements from every list for which a skip actually occurred to the heap, and repeat. An important observation here is that an element that was popped from the heap might end up subsequently being re-inserted to the heap, if it happens to be equal to the top element of the heap after the $\theta - 1$ total removals. This is possible given that a skip operation might result in skipping backwards on some of the lists. The correctness of the algorithm follows from the fact that if an element appears in at least θ lists, eventually it will have to appear at the top of the heap θ times (after potentially multiple reinsertions to the heap). The advantage of the algorithm is that it will result in many list elements that appear in fewer than θ lists to be instantly skipped. Skipping might be beneficial for main memory processing or lists stored on solid state drives, but not so for disk resident inverted lists due to the cost of the random seeks required in order to skip over elements.

A generalization of this algorithm is to utilize the fact that some token lists are short and others are long. We divide the lists into two groups, one with the τ longest lists and one with the rest of the lists. We use the aforementioned optimized `multiway merge` algorithm on the short lists to identify candidate strings that appear at least $\theta - \tau$ times. Then, we probe the τ long lists to verify the candidates. Depending on the query and the data at hand we can find an appropriate value τ that will result in a good tradeoff between the cost of running the `multiway merge` algorithm on the short lists and the subsequent probing of the long lists. An alternative approach that can be beneficial in certain scenarios is to avoid probing the long lists altogether, and simply retrieve all candidate strings produced by the `multiway merge` algorithm on the short lists and evaluate their similarity with the query directly. This algorithm can result in faster processing whenever the total number of candidates produced is sufficiently small.

The optimized `multiway merge` algorithm can be generalized for all similarity functions with unit token weights. It cannot be generalized for arbitrary weights.

Nevertheless, there is another simple optimization for skipping list elements in the case of arbitrary token weights. Given k elements in the heap and the fact that each list has a fixed weight, if we keep track of the list from which each element has been popped off from, we can directly determine the best possible score of the 1st, 2nd, ..., $k-1$-th element in the heap. If the best score of the i-th element is smaller than the threshold, we can immediately pop the first i elements from the heap and seek directly beyond element i in each affected list. Clearly, this optimization also leads to an early termination condition. When a given list becomes inactive, the best possible score of any unseen candidate reduces by the weight of that list. When enough lists become inactive such that the best possible weight of all unseen candidates falls below the query threshold, the algorithm can safely terminate. We refer to this algorithm as the `optimized multiway merge` algorithm.

6.1.2 Lists Sorted in L_p-norm Order

6.1.2.1 Normalized Weighted Intersection

For Normalized Weighted Intersection similarity we make the following observation:

Lemma 6.1 (Normalized Weighted Intersection L_1-norm Filter). Given sets s, r and Normalized Weighted Intersection threshold $0 < \theta \leq 1$ the following holds:

$$\mathcal{N}(s,r) \geq \theta \Leftrightarrow \theta \|r\|_1 \leq \|s\|_1 \leq \frac{\|r\|_1}{\theta}.$$

Proof. For the lower bound, let $s \subseteq r$. Hence,

$$\mathcal{N}(s,r) = \frac{\|s \cap r\|_1}{\max(\|s\|_1, \|r\|_1)} \geq \theta \Rightarrow \frac{\|s\|_1}{\|r\|_1} \geq \theta \Rightarrow \theta\|r\|_1 \leq \|s\|_1.$$

For the upper bound, let $r \subseteq s$. Hence,

$$\mathcal{N}(s,r) = \frac{\|s \cap r\|_1}{\max(\|s\|_1, \|r\|_1)} \geq \theta \Rightarrow \frac{\|r\|_1}{\|s\|_1} \geq \theta \Rightarrow \|s\|_1 \leq \frac{\|r\|_1}{\theta}. \qquad \square$$

We can use the L_1-norm filter to possibly prune strings appearing in any list in L_v without the need to compute the actual similarity with the query.

For that purpose, we sort token lists in increasing (or decreasing) order of the L_1-norms of strings, rather than in string identifier order. (Without loss of generality, in the rest we assume that the inverted lists are always sorted in increasing order of L_p-norms.) Using Lemma 6.1, we can directly skip over all candidate strings with L_1-norm $\|s\|_1 < \theta\|v\|_1$ (where $\|v\|_1$ is the L_1-norm of the query) and stop scanning a list whenever we encounter the first candidate string with L_1-norm $\|s\|_1 > \frac{\|v\|_1}{\theta}$.

Given that the lists are not sorted in increasing string identifier order, the obvious choice is to use the classic `threshold` based algorithms to compute the similarity of each string. `Threshold` algorithms utilize special terminating conditions that enable processing to stop before exhaustively scanning all token lists. This is easy to show if the similarity function is a monotone aggregate function. Let $\alpha_i(s,v) = \frac{W(\lambda_i^v)}{\max(\|s\|_1,\|v\|_1)}$ be the partial similarity of data string s and query token λ_i^v. Then, $\mathcal{N}(s,v) = \sum_{\lambda_i^v \in s \cap v} \alpha_i(s,v)$. It can be shown that $\mathcal{N}(s,v)$ is a monotone aggregate function, i.e., $\mathcal{N}(s,v) \le \mathcal{N}(s',v)$ if $\forall i \in [1,m]$: $\alpha_i(s,v) \le \alpha_i(s',v)$. The proof is straightforward.

There are three `threshold` algorithm flavors. The first scans lists sequentially and in a round robin fashion and computes the similarity of strings incrementally. The second uses random accesses to compute similarity aggressively every time a new string identifier is encountered in one of the lists. The third uses combination strategies of both sequential and random accesses.

Since the sequential round robin algorithm computes similarity incrementally, it has to temporarily store in main memory information about strings whose similarity has only been partially computed. Hence, the algorithm has a high book-keeping cost, given that the candidate set needs to be maintained as a hash table organized by string identifiers. The aggressive random access based algorithm computes the similarity of strings in one step and hence does not need to store any information in main memory. Hence it has a very small book-keeping cost, but on the other hand it has to perform a large number of random accesses to achieve this. This could be a drawback on traditional storage devices (like hard drives) but unimportant in modern solid state drives. Combination strategies try to strike a balance between low book-keeping and a small number of random accesses. A simple round robin strategy, called `nra` (for No Random Access algorithm) is shown in Algorithm 6.1.2.

The algorithm keeps a candidate set M containing tuples consisting of a string identifier, a partial similarity score, and a bit vector containing zeros for all query tokens that have not matched with the particular string identifier yet, and ones for those that a match has been found. The candidate set is organized as a hash table on string identifiers for efficiently determining whether a given string has already been encountered or not. Lemma 6.1 states that for each list we only need to scan elements within a narrow L_1-norm interval. We skip directly to the first element in every list with L_1-norm $\|s\|_1 \geq \theta\|v\|_1$ (assuming that we are indexing sequences and that lists are primarily sorted by token positions, we also skip to the first element with token position equal to, or within some bounds from, the token position in the query; for simplicity in what follows we ignore any positional information). The algorithm proceeds by reading one element per list in every iteration, from each active list. According to Lemma 6.1, if the next element read from list $L(\lambda_i^v)$ has L_1-norm larger than $\frac{\|v\|_1}{\theta}$ the algorithm flags $L(\lambda_i^v)$ as inactive. Otherwise, there are two cases to consider:

- If the string identifier read is already contained in the candidate set M, its entry is updated to reflect the new partial similarity score. In addition, the bit vector is updated to reflect the fact that the candidate string contains the particular query token. Also, the algorithm checks whether any other lists have already become inactive, which implies one of two things: If the candidate string contains the token corresponding to that list, then the bit vector has already been set to one from a previous iteration and the partial similarity score has been updated to reflect that fact; or the candidate string does not contain that token, hence the bit vector can be set to one without updating the partial similarity score (essentially adding zero to the similarity score).

Algorithm 6.1.2: NRA(v,θ)

```
Tokenize the query: $v = \{(\lambda_1^v, 1), \dots, (\lambda_m^v, m)\}$
$M$ is an empty map of $(\ddot{s}, \mathcal{N}_s, B_1, \dots B_m)$ tuples ($B_i$ are bits)
for $i \leftarrow 1$ to $m$ do Seek to first element of $L(\lambda_i^v)$ s.t. $\|s\|_1 \geq \theta\|v\|_1$
while there exists an active $L(\lambda_i^v)$
    $f = \infty, w = 0$
    for $i \leftarrow 1$ to $m$
      do
        if $L(\lambda_i^v)$ is inactive then continue
        $(\ddot{s}, \|s\|_1) \leftarrow L(\lambda_i^v)$.next
        if $\|s\|_1 > \frac{\|v\|_1}{\theta}$ then make $L(\lambda_i^v)$ inactive and continue
        $f = \min(f, \|s\|_1), w = w + W(\lambda_i^v)$
        $(\ddot{s}, \mathcal{N}_s, B_1, \dots, B_m) \leftarrow M$.find$(\ddot{s})$
        for $j \leftarrow 1$ to $m$
          if $j = i$ or $L(\lambda_j^v)$ is inactive then $B_j = 1$
        if $\ddot{s} \in M$
          then
            if $B_1 = 1, \dots, B_m = 1$
              then
                if $(\mathcal{N}_s + W(\lambda_i^v))/\max(\|s\|_1, \|v\|_1) \geq \theta$
                  then report $\ddot{s}, (\mathcal{N}_s + W(\lambda_i^v))/$
                    $\max(\|s\|_1, \|v\|_1)$
                Remove from $M$
              else $M \leftarrow (\ddot{s}, \mathcal{N}_s + W(\lambda_i^v), B_1, \dots, B_m)$
          else $M \leftarrow (\ddot{s}, W(\lambda_i^v), B_1, \dots, B_m)$
    for all $(\ddot{s}, \mathcal{N}_s, B_1, \dots, B_m) \in M$
      do
        for $j \leftarrow 1$ to $m$
          if $L(\lambda_j^v)$ is inactive then $B_j = 1$
        if $B_1 = 1, \dots, B_m = 1$
          then
            if $\mathcal{N}_s/\max(\|s\|_1, \|v\|_1) \geq \theta$ then report $\ddot{s}, \mathcal{N}_s/$
              $\max(\|s\|_1, \|v\|_1)$
            Remove from $M$
    if $\frac{w}{\max(f, \|v\|_1)} < \theta$ and $M$ is empty then break
```

- If the new string read is not already contained in M, a new entry is created and reported as an answer or inserted in M, using reasoning similar to the previous step.

After one round robin iteration, the bit vector of each candidate in the candidate set is updated, and candidates with fully set bit vectors are reported as answers or evicted from the candidate set accordingly.

The algorithm can terminate early if two conditions are met. First the candidate set is empty, which means no more viable candidates whose similarity has not been completely computed yet exist. Second, the maximum possible score of a conceptual frontier string that appears at the current position in all lists cannot exceed the query threshold.

Lemma 6.2. Let $L_a \subseteq L_v$ be the set of active lists. The terminating condition

$$\mathcal{N}^f = \frac{\sum_{L(\lambda_i^v) \in L_a} W(\lambda_i^v)}{\max(\min_{L(\lambda_i^v) \in L_a} \|f_i\|_1, \|v\|_1)} < \theta,$$

does not lead to any false dismissals.

Proof. The terminating condition leads to no false dismissals if and only if the maximum possible normalized weighted intersection of any unseen element is smaller than θ. To maximize $\mathcal{N}(s,v)$ we need to maximize the numerator and minimize the denominator.

After each round of the `nra` algorithm, let the frontier element of each list (i.e., the last element read on that list) be $f_i, 1 \leq i \leq m$. Each frontier element corresponds to a string f_i with L_1-norm $\|f_i\|_1$. Let $L_a \subseteq L_v$ be the set of active lists (i.e., lists that have not been fully traversed yet).

A conceptual frontier element that has not been seen yet has a possible maximum weighted intersection of $\sum_{L(\lambda_i^v) \in L_a} W(\lambda_i^v)$. To minimize the denominator, notice that $\mathcal{N}(s,v)$ is monotone decreasing in $\|s\|_1$. Hence, for strings s, r s.t. $\|s\|_1 < \|r\|_1$ and $\|s \cap v\|_1 = \|r \cap v\|_1$ it holds that $\mathcal{N}(s,v) \geq \mathcal{N}(r,v)$. Thus, the maximum possible similarity of an unseen candidate is

$$\mathcal{N}^f = \frac{\sum_{L(\lambda_i^v)\in L_a} W(\lambda_i^v)}{\max(\min_{L(\lambda_i^v)\in L_a} \|f_i\|_1, \|v\|_1)}.$$

Hence, $\mathcal{N}^f < \theta$ is a sufficient stopping condition that leads to no false dismissals. □

It is easy to see that a tighter bound for $\mathcal{N}^f$ exists. This can be achieved by examining the L_1-norm of all frontier elements simultaneously and is based on the observation that

Observation 6.1. Given a string s with L_1-norm $\|s\|_1$ and the L_1-norms of all frontier elements $\|f_i\|_1$, we can immediately deduce whether s potentially appears in list $L(\lambda_i^v)$ or not by a simple comparison: If $\|s\|_1 < \|f_i\|_1$ and s has not been encountered in list $L(\lambda_i^v)$ yet, then s does not appear in $L(\lambda_i^v)$ (given that lists are sorted in increasing order of L_1-norms).

Let $L(\lambda_j^v)$ be a list s.t. $\|f_j\|_1 = \min_{L(\lambda_i^v)\in L_a} \|f_i\|_1$. Based on Observation 6.1, a conceptual string s with L_1-norm $\|s\|_1 = \|f_j\|_1$ that has not been encountered yet, can appear only in list $L(\lambda_j^v)$. Thus

Lemma 6.3. Let $L_a \subseteq L_v$ be the set of active lists. The terminating condition

$$\mathcal{N}^f = \max_{L(\lambda_i^v)\in L_a} \frac{\sum_{L(\lambda_j^v)\in L_a : \|f_j\|_1 \le \|f_i\|_1} W(\lambda_j^v)}{\max(\|f_i\|_1, \|v\|_1)} < \theta,$$

does not lead to any false dismissals.

Proof. The proof is based on the proof of Lemma 6.2 and a simple enumeration argument with at most m possible cases. Without loss of generality, let $L_a = \{L(\lambda_1^v), \ldots, L(\lambda_m^v)\}$ be the set of active lists and $\|f_1\|_1 < \ldots < \|f_m\|_1$. Then, if $\mathcal{N}^f < \theta$ is true, we have the following possible cases:

1. An unseen string s appears only in $L(\lambda_1^v)$. By construction $\|s\|_1 \ge \|f_1\|_1 \Rightarrow \mathcal{N}(s,v) \le \mathcal{N}(f_1,v) \Rightarrow \mathcal{N}(s,v) < \theta$.

2. An unseen string s appears only in $L(\lambda_1^v), L(\lambda_2^v)$. By construction $\|s\|_1 \geq \|f_2\|_1 \Rightarrow \mathcal{N}(s,v) < \theta$.
3. ...

m. An unseen string s appears only in $L(\lambda_1^v), \ldots, L(\lambda_m^v)$. By construction $\|s\|_1 > \|f_m\|_1 \Rightarrow \mathcal{N}(s,v) < \theta$. □

We can also use Observation 6.1 to improve the pruning power of the `nra` algorithm by computing a best-case similarity for all candidate strings before and after they have been inserted in the candidate set. Strings whose best-case similarity is below the threshold can be pruned immediately, reducing the memory requirements and the book-keeping cost of the algorithm. The best-case similarity of a new candidate uses Observation 6.1 to identify lists that do not contain that candidate.

Lemma 6.4. Given query v, candidate string s, and a subset of lists $L_{v'} \subseteq L_v$ potentially containing s (based on Observation 6.1 and the frontier elements f_i), the best-case similarity score for s is

$$\mathcal{N}^b(s,v) = \frac{\|v'\|_1}{\max(\|s\|_1, \|v\|_1)}.$$

It is important to note here that the above reasoning is correct if and only if each string entry appears only once in every token list. This is indeed the case for the frequency set and set representations, but not always true for sequences, *when the positional information is disregarded* (e.g., for computing the similarity of strings without regard for positions). In that case, in order to compute the termination condition correctly, we need to know for an arbitrary frontier element the maximum number of times that the element might be contained in a given list. The maximum frequency of an arbitrary entry per inverted list can be computed at list construction time. Alternatively, we can assume that a given token appears in a data string at most as many times as it appears in the query, as a worst case scenario, and as a result overestimate the similarity of the frontier elements.

An alternative implementation of the `nra` algorithm can postpone updating candidate bit vectors and purging candidates from M until after all lists have been examined, in order to reduce the book-keeping cost. The drawback of this implementation is that the candidate set might become prohibitively large. A compromise between these two extreme strategies is to postpone purging the candidate set until its size becomes large enough. Which one is the best strategy depends, in practice, on many variables, including the query token distribution. For example, some token distributions might favor aggressive candidate pruning, while others might render the purging step ineffective and hence unnecessary after each round robin iteration.

An alternative *threshold* algorithm, based on random rather than sequential accesses of lists, is to assume that token lists are sorted in increasing L_p-norms but that there also exists one string identifier index (e.g., a B-tree) per inverted list. Then, a random access only algorithm can scan token lists sequentially and for every element probe the remaining lists using the string identifier index to immediately compute the exact similarity of the string. This algorithm, called `ta` (for Threshold Algorithm), has similar terminating conditions as those used for `nra`.

The last threshold based strategy is to use a combination of `nra` and random accesses. We run the `nra` algorithm as usual but after each iteration we do a linear pass over the candidate set and use random accesses to compute the exact similarity of the strings and empty the candidate set. Another strategy is to run the `nra` algorithm until Lemma 6.3 is satisfied, and then revert to random accesses to compute the actual similarity of all candidates remaining in the candidate set.

Threshold algorithms are not only complex in terms of implementation but also result in very high book-keeping costs. An alternative strategy is to use, once more, the `optimized multiway merge` algorithm to compute the scores. In order to achieve this we sort lists primarily by norms and secondarily by string identifiers. Then, we can simply run the `optimized multiway merge` algorithm on the lists, and also take into account the norm filter. The algorithm is similar to Algorithm 6.1.1, but we first skip directly to the smallest element with norm larger than or equal to the norm filter lower bound, and stop processing a list once we encounter an element with norm larger than the upper bound. Notice that the `optimized multiway merge` algorithm implicitly guarantees that when a new element is inserted in the heap with L_1-norm larger than the smallest L_1-norm currently in the heap, the list corresponding to that element effectively becomes inactive until all other elements with L_1-norms smaller than the norm of that element are evicted from the heap. Essentially, the merging process operates on a small range of L_1-norms until enough lists become "inactive" to force all elements within this range to be evicted from the heap and bring about a new iteration of merging on the next range of L_1-norms (or early termination). As with `nra`, we use Lemma 6.4 to compute the best case scores of elements when evicting entries from the heap and Lemma 6.3 as the terminating condition. The algorithm is shown as Algorithm 6.1.3 (for simplicity, the algorithm assumes that there does not exist a single list whose weight is such that an element appearing only in that list can satisfy the threshold).

6.1.2.2 Dice

Exactly the same norm based algorithms can be used for Dice similarity. It is easy to show that Dice similarity is a monotone aggregate function with partial similarity defined as $\alpha_i(s,v) = \frac{W(\lambda_i^v)}{\|s\|_1+\|v\|_1}$. Hence, all algorithms presented above can be adapted to Dice. In addition, it is easy to prove the following lemmas.

Lemma 6.5 (Dice L_1-norm Filter). Given sets s, r and Dice similarity threshold $0 < \theta \leq 1$ the following holds:

$$\mathcal{D}(s,r) \geq \theta \Leftrightarrow \frac{\theta}{2-\theta}\|r\|_1 \leq \|s\|_1 \leq \frac{2-\theta}{\theta}\|r\|_1.$$

Proof. For the lower bound:

It holds that $\|s \cap r\|_1 \leq \|s\|_1$. Hence,

$$\mathcal{D}(s,r) \geq \theta \Rightarrow \frac{2\|s \cap r\|_1}{\|s\|_1 + \|r\|_1} \geq \theta \Rightarrow \frac{2\|s\|_1}{\|s\|_1 + \|r\|_1} \geq \theta \Rightarrow \frac{\theta}{2-\theta}\|r\|_1 \leq \|s\|_1.$$

For the upper bound:

It holds that $\|s \cap r\|_1 \leq \|r\|_1$. Hence,

$$\mathcal{D}(s,r) \geq \theta \Rightarrow \frac{2\|s \cap r\|_1}{\|s\|_1 + \|r\|_1} \geq \theta \Rightarrow \frac{2\|r\|_1}{\|s\|_1 + \|r\|_1} \geq \theta \Rightarrow \|s\|_1 \leq \frac{2-\theta}{\theta}\|r\|_1.$$

□

Algorithm 6.1.3: Optimized Multiway Merge(v,θ)

```
Tokenize the query: $v = \{(\lambda_1^v, 1), \ldots, (\lambda_m^v, m)\}$
$M$ is an empty heap of $(\ddot{s}, \|s\|_1, \mathcal{I}_s, i)$ tuples, sorted on $\|s\|_1, \ddot{s}$
$S$ is a stack of indexes $i$
$(\ddot{f}_1, \|f_1\|_1) \ldots, (\ddot{f}_m, \|f_m\|_1)$ are the frontier elements
for $i \leftarrow 1$ to $m$ do $\ddot{f}_i = 0, \|f_i\|_1 = \theta\|v\|_1$ and $S \leftarrow i$
while $M$ is not empty or $S$ is not empty
do
    if $M$ is empty and enough lists are inactive then break
    while $S$ is not empty
    do
        $i = S$.pop
        Seek to first element of $L(\lambda_i^v)$ s.t. $\|s\|_1 \geq \|f_i\|_1$
        $(\ddot{s}, \|s\|_1) = L(\lambda_i^v)$.next and $\ddot{f}_i = \ddot{s}, \|f_i\|_1 = \|s\|_1$
        if $\|s\|_1 > \frac{\|v\|_1}{\theta}$ then make $L(\lambda_i^v)$ inactive and
            continue
        if $\ddot{s} \in M$
          then
            $(\ddot{s}, \|s\|_1, \mathcal{I}_s, j) \leftarrow M$.find$(\ddot{s})$
            $\mathcal{I}_s += W(\lambda_i^v)$ and $S \leftarrow i$
          else $M \leftarrow (\ddot{s}, \|s\|_1, W(\lambda_i^v), i)$
    while $M$ is not empty
    do
        $(\ddot{r}, \|r\|_1, \mathcal{I}_r, j) \leftarrow M$.peek
        if $\mathcal{I}_r / \max(\|r\|_1, \|v\|_1) \geq \theta$
          then
            Report $(\ddot{r}, \mathcal{I}_r / \max(\|r\|_1, \|v\|_1))$
            $M$.pop, $S \leftarrow j$, break
          else
            $\tau = 0$
            for $i \leftarrow 1$ to $m$
              if $i \neq j$ and $\|f_i\|_1 \leq \|r\|_1$ and $\ddot{f}_i \leq \ddot{r}$
                then $\tau += W(\lambda_i^v)$
            if $(\mathcal{I}_r + \tau) / \max(\|r\|_1, \|v\|_1) \geq \theta$
              then break
              else $M$.pop, $S \leftarrow j$
    $(\ddot{r}, \|r\|_1, \mathcal{I}_r, j) \leftarrow M$.peek
    for all $i \in S$ do $(\|f_i\|_1 < \|r\|_1)$ ? $\|f_i\|_1 = \|r\|_1, \ddot{f}_i = \ddot{r}$
```

Lemma 6.6. Let $L_a \subseteq L_v$ be the set of active lists. The terminating condition

$$\mathcal{D}^f = \max_{L(\lambda_i^v) \in L_a} \frac{\sum_{L(\lambda_j^v) \in L_a : \|f_j\|_1 \leq \|f_i\|_1} 2W(\lambda_j^v)}{\|f_i\|_1 + \|v\|_1} < \theta,$$

does not lead to any false dismissals.

Lemma 6.7. Given query v, candidate string s, and a subset of list $L_{v'} \subseteq L_v$ potentially containing s, the best-case similarity score for s is

$$\mathcal{D}^b(s,v) = \frac{2\|v'\|_1}{\|s\|_1 + \|v\|_1}.$$

6.1.2.3 Cosine

Recall that cosine similarity is computed based on the L_2-norm of the strings. Once again, an inverted index is built where this time each list element contains tuples (string-identifier/token-position/L_2-norm) for sequences, (string-identifier/token-frequency/L_2-norm) for frequency-sets and (string-identifier/L_2-norm) for sets. Cosine similarity is a monotone aggregate function with partial similarity $\alpha_i(s,v) = \frac{W(\lambda_i^v)^2}{\|s\|_2\|v\|_2}$. An L_2-norm filter also holds.

Lemma 6.8 (Cosine similarity L_2-norm Filter). Given sets s, r and Cosine similarity threshold $0 < \theta \leq 1$ the following holds:

$$\mathcal{C}(s,r) \geq \theta \Leftrightarrow \theta\|r\|_2 \leq \|s\|_2 \leq \frac{\|r\|_2}{\theta}.$$

Proof. For the lower bound:

It holds that $\|s \cap r\|_2 \leq \|s\|_2$. Hence,

$$\mathcal{C}(s,r) \geq \theta \Rightarrow \frac{(\|s \cap r\|_2)^2}{\|s\|_2\|r\|_2} \geq \theta \Rightarrow \frac{(\|s\|_2)^2}{\|s\|_2\|r\|_2} \geq \theta \Rightarrow \theta\|r\|_2 \leq \|s\|_2.$$

For the upper bound:

It holds that $\|s \cap r\|_2 \leq \|r\|_2$. Hence,

$$\mathcal{C}(s,r) \geq \theta \Rightarrow \frac{(\|s \cap r\|_2)^2}{\|s\|_2\|r\|_2} \geq \theta \Rightarrow \frac{(\|r\|_2)^2}{\|s\|_2\|r\|_2} \geq \theta \Rightarrow \|s\|_2 \leq \frac{\|r\|_1}{\theta}. \qquad \square$$

In addition

Observation 6.2. Given a string s with L_2-norm $\|s\|_2$ and the L_2-norms of all frontier elements $\|f_i\|_2$, we can immediately deduce whether s potentially appears in list $L(\lambda_i^v)$ or not by a simple comparison: If $\|s\|_2 < \|f_i\|_2$ and s has not been encountered in list $L(\lambda_i^v)$ yet, then s does not appear in $L(\lambda_i^v)$ (given that lists are sorted in increasing order of L_2-norms).

Hence

Lemma 6.9. Let $L_a \subseteq L_v$ be the set of active lists. The terminating condition

$$\mathcal{C}^f = \max_{L(\lambda_i^v) \in L_a} \frac{\sum_{L(\lambda_j^v) \in L_a : \|f_j\|_2 \leq \|f_i\|_2} W(\lambda_j^v)^2}{\|f_i\|_2\|v\|_2} < \theta,$$

does not lead to any false dismissals.

Lemma 6.10. Given query v, candidate string s, and a subset of lists $L_{v'} \subseteq L_v$ potentially containing s, the best-case similarity score for s is

$$\mathcal{C}^b(s,v) = \frac{(\|v'\|_2)^2}{\|s\|_2\|v\|_2}.$$

The actual algorithms in principle remain the same.

6.1.2.4 Jaccard

Jaccard similarity presents some difficulties. Notice that Jaccard is a monotone aggregate function with partial similarity $\alpha_i(s,v) = \frac{W(\lambda_i^v)}{\|s \cup v\|_1}$.

Nevertheless, we cannot use this fact for designing termination conditions for the simple reason that α_i's cannot be evaluated on a per token list basis since $\|s \cup v\|_1$ is not known in advance (knowledge of $\|s \cup v\|_1$ implies knowledge of the whole string s and hence knowledge of $\|s \cap v\|_1$ which is equivalent to directly computing the similarity). Recall that an alternative expression for Jaccard is $\mathcal{J}(s,v) = \frac{\|s \cap v\|_1}{\|s\|_1 + \|v\|_1 - \|s \cap v\|_1}$. This expression cannot be decomposed into aggregate parts on a per token basis, and hence is not useful either. Nevertheless, we can still prove various properties of Jaccard that enable us to use all `threshold` algorithms and the `optimized multiway merge` algorithm. In particular

---

Lemma 6.11 (Jaccard L_1-norm Filter). Given sets s, r and Jaccard similarity threshold $0 < \theta \leq 1$ the following holds:

$$\mathcal{J}(s,r) \geq \theta \Leftrightarrow \theta\|r\|_1 \leq \|s\|_1 \leq \frac{\|r\|_1}{\theta}.$$

Proof. For the lower bound:

It holds that $\|s \cup r\|_1 \geq \|r\|_1$ and $\|s \cap r\|_1 \leq \|s\|_1$. Hence,

$$\mathcal{J}(s,r) \geq \theta \Rightarrow \frac{\|s \cap r\|_1}{\|s \cup r\|_1} \geq \theta \Rightarrow \frac{\|s\|_1}{\|r\|_1} \geq \theta \Rightarrow \theta\|r\|_1 \leq \|s\|_1.$$

For the upper bound:

It holds that $\|s \cup r\|_1 \geq \|s\|_1$ and $\|s \cap r\|_1 \leq \|r\|_1$. Hence,

$$\mathcal{J}(s,r) \geq \theta \Rightarrow \frac{\|s \cap r\|_1}{\|s \cup r\|_1} \geq \theta \Rightarrow \frac{\|r\|_1}{\|s\|_1} \geq \theta \Rightarrow \|s\|_1 \leq \frac{\|r\|_1}{\theta}. \qquad \square$$

Lemma 6.12. Let $L_a \subseteq L_v$ be the set of active lists. Let $I_i = \sum_{L(\lambda_j^v) \in L_a : \|f_j\|_1 \leq \|f_i\|_1} W(\lambda_j^v)$. The terminating condition

$$\mathcal{J}^f = \max_{L(\lambda_i^v) \in L_a} \frac{I_i}{\|f_i\|_1 + \|v\|_1 - I_i} < \theta,$$

does not lead to any false dismissals.

Proof. Let $x = \sum_{\lambda \in s \cap v} W(\lambda)$ be the weight of tokens in the intersection of strings s and v. Let $y = \sum_{\lambda \in s \setminus v} W(\lambda)$ be the weight of tokens contained only in s and $z = \sum_{\lambda \in v \setminus s} W(\lambda)$ be the weight of tokens contained only in v. Then

$$\mathcal{J}(s,v) = \frac{x}{x+y+z}.$$

It suffices to show that the function $f(x,y,z) = \frac{x}{x+y+z}$ is monotone increasing in x and monotone decreasing in y, for positive x, y, z. Then, the proof is similar to that of Lemma 6.2 and 6.3. □

Lemma 6.13. Function $f(x,y,z) = \frac{x}{x+y+z}$ is monotone increasing in x and monotone decreasing in y for all positive x, y, z.

Proof. Consider the function $g(x) = 1/f(x)$. Function $f(x)$ is monotone increasing iff $g(x)$ is monotone decreasing.

$$g(x) = \frac{x+y+z}{x} = 1 + \frac{y}{x} + \frac{z}{x},$$

which is clearly monotone decreasing in x for positive x, y, z. Also $f(y)$ is monotone decreasing iff $g(y) = 1/f(y)$ is monotone increasing, which is clearly true for positive x, y, z. □

Lemma 6.14. Given query v, candidate string s, and a subset of list $L_{v'} \subseteq L_v$ potentially containing s, the best-case similarity score for s is

$$\mathcal{J}^b(s,v) = \frac{\|v'\|_1}{\|s\|_1 + \|v\|_1 - \|v'\|_1}.$$

6.1.2.5 Weighted Intersection

Consider now Weighted Intersection similarity (i.e., without normalization). We have seen how to evaluate weighted similarity using the `optimized multiway merge` strategy. We now explore if the `threshold` algorithms can be used effectively for the same purpose.

Notice that for weighted intersection the L_1-norm does not play any role in the similarity of two strings, hence we cannot design an L_1-norm filter to prune strings. Nevertheless, assume that we use the `nra` algorithm to merge token lists. Weighted intersection is a monotone aggregate function with partial weights $\alpha_i(s,v) = W(\lambda_i^v)$. Notice that in this case the partial weights of all strings in a given token list are constant (i.e., $\alpha_i(s,v) = \alpha_i(s',v), \forall s,s' \in L(\lambda_i^v)$). An immediate conclusion is that the terminating condition $\mathcal{I}^f = \sum_{i=1}^{l} W(\lambda_i^v) < \theta$ of the `nra` algorithm will never be true (for $0 < \theta \leq \sum_{i=1}^{l} W(\lambda_i^v)$). Hence, given that neither an L_1-norm filter nor a termination condition can be applied, the `nra` algorithm will have to exhaustively scan all token lists.

6.1.3 Prioritization Across Lists

This section focuses on strategies that take advantage of specific access priorities across inverted lists. Notice that for the `multiway merge` and `threshold` based algorithms the order in which inverted lists are processed is not important (e.g., the order of the round robin iterations over the inverted lists for the `nra` algorithm). Nevertheless, the distribution of tokens in the data strings can help determine a more meaningful ordering in which inverted lists are processed, that can yield significant performance benefits.

Let the lists in L_v be sorted according to their respective token weights, from heaviest to lightest. Without loss of generality let this ordering be $L_v = \{L(\lambda_1^v), \ldots, L(\lambda_m^v)\}$ s.t. $W(\lambda_1^v) \geq \cdots \geq W(\lambda_m^v)$. Considering the partial token weights of a given similarity function, since $\alpha_1(s,v) \geq \cdots \geq \alpha_m(s,v)$, the most promising candidates appear in list $L(\lambda_1^v)$; the second most promising appear in $L(\lambda_2^v)$; and so on. This leads to an alternative sequential algorithm, which we refer to here as `heaviest first`. The algorithm exhaustively reads the heaviest token list first, and stores all strings in a candidate set. The second to heaviest list is scanned next. The similarity of strings that have already been encountered in the previous list is updated, and new candidates are added in the candidate set, until all lists have been scanned and the similarities of all candidates have been computed. While a new list is being traversed the algorithm prunes the candidates in the candidate set whose best-case similarity is below the query threshold. The best-case similarity for every candidate is computed by taking the partial similarity score already computed for each candidate after list $L(\lambda_i^v)$ has been scanned, and assuming that the candidate exists in all subsequent lists $L(\lambda_{i+1}^v), \ldots, L(\lambda_m^v)$.

The important observation here is that we can compute a tighter L_1-norm filtering bound each time a new list is scanned. The idea is to treat the remaining lists as a new query $v' = \lambda_{i+1}^v \ldots \lambda_m^v$ and recompute the bounds using Lemma 6.1. The intuition is that the most promising new candidates in lists $L(\lambda_{i+1}^v), \ldots, L(\lambda_m^v)$, are the ones containing all tokens $\lambda_{i+1}^v, \ldots, \lambda_m^v$ and hence potentially satisfying $s = v'$.

Lemma 6.15 (Norm Tightening). Without loss of generality, given query string $v = \lambda_1^v \cdots \lambda_m^v$ s.t. $W(\lambda_1^v) \geq \cdots \geq W(\lambda_m^v)$ and query threshold $0 < \theta \leq 1$, for any viable candidate s s.t. $s \notin L(\lambda_1^v), \ldots L(\lambda_i^v)$ the following holds:

- Normalized Weighted Intersection:

$$\mathcal{N}(v,s) \geq \theta \Rightarrow \theta \|v\|_1 \leq \|s\|_1 \leq \frac{\|\lambda_{i+1}^v \cdots \lambda_m^v\|_1}{\theta}.$$

- Jaccard:

$$\mathcal{J}(v,s) \geq \theta \Rightarrow \theta \|v\|_1 \leq \|s\|_1 \leq \frac{1+\theta}{\theta} \|\lambda_{i+1}^v \cdots \lambda_m^v\|_1 - \|v\|_1.$$

- Dice:

$$\mathcal{D}(v,s) \geq \theta \Rightarrow \frac{\theta}{2-\theta} \|v\|_1 \leq \|s\|_1 \leq \frac{2}{\theta} \|\lambda_{i+1}^v \cdots \lambda_m^v\|_1 - \|v\|_1.$$

- Cosine:

$$\mathcal{C}(v,s) \geq \theta \Rightarrow \theta \|v\|_2 \leq \|s\|_2 \leq \frac{(\|\lambda_{i+1}^v \cdots \lambda_m^v\|_2)^2}{\theta \|v\|_2}.$$

Care needs to be taken though in order to accurately complete the partial similarity scores of all candidates already inserted in the candidate set from previous lists, which can have L_1-norms that do not satisfy the recomputed bounds for v'. For that purpose the algorithm identifies the largest L_1-norm in the candidate set and scans subsequent lists using that bound. If a string already appears in the candidate set its similarity is updated. If a string does not appear in the candidate set (this is either a new string or an already pruned string) then the string is inserted in the candidate set if and only if its L_1-norm satisfies the recomputed bounds based on v'.

To reduce the book-keeping cost of the candidate set, the algorithm stores the set as a linked list, sorted primarily in increasing order of L_1-norm and secondarily in increasing order of string identifiers. Then the algorithm can do a merge join of any token list and the candidate list very efficiently in one pass. The complete algorithm is shown as Algorithm 6.1.4.

The biggest advantage of the `heaviest first` algorithm is that it can benefit from long sequential accesses, one list at a time. Notice that both the `nra` and the `multiway merge` strategies have to access lists in parallel. For traditional disk based storage devices, as the query size increases and the number of lists that need to be accessed in parallel increases, a larger buffer is required in order to prefetch entries from all the lists and the seek time increases, moving from one list to the next (this is not an issue for solid state drives or for main memory processing).

Notice that the `heaviest first` strategy has another advantage when token weights are assigned according to token popularity, as is the case for *idf* weights. In that case, the heaviest tokens are rare tokens that correspond to the shortest lists and the `heaviest first` strategy is equivalent to scanning the lists in order of their length, shortest list first. The advantage is that this keeps the size of the candidate set to a minimum, since as the algorithm advances to lighter tokens, the probability of newly encountered strings (strings that do not share any of the heavy tokens with the query) will satisfy the similarity threshold significantly decreases. Such strings are therefore immediately discarded. In addition, the algorithm terminates with high probability before long lists have to be examined exhaustively.

This observation leads to an alternative strategy that uses a combination of sequential accesses for short lists and random accesses for long lists. The alternative algorithm is similar to the `heaviest`

Algorithm 6.1.4: HEAVIEST FIRST(v,θ)

```
Tokenize query: $v = \{(\lambda_1^v,1),\dots,(\lambda_m^v,m)\}$ s.t. $W(\lambda_i^v) \geq W(\lambda_j^v), i < j$
$M$ is a list of $(\ddot{s},\mathcal{N}_s,\|s\|_1)$ sorted in $(\|s\|_1,\ddot{s})$ order
$M \leftarrow \emptyset$
$\tau = \|v\|_1$
for $i \leftarrow 1$ to $m$
  do
    Seek to first element of $L(\lambda_i^v)$ with $\|s\|_1 \geq \theta\|v\|_1$
    $(\ddot{r},\mathcal{N}_r,\|r\|_1) \leftarrow M$.last
    $\mu = \max(\|r\|_1, \frac{\tau}{\theta})$
    $\tau = \tau - W(\lambda_i^v)$
    $(\ddot{r},\mathcal{N}_r,\|r\|_1) \leftarrow M$.first
    while not at end of $L(\lambda_i^v)$
      do
        $(\ddot{s},\|s\|_1) \leftarrow L(\lambda_i^v)$.next
        if $\|s\|_1 > \mu$ then break
        while $\|r\|_1 \leq \|s\|_1$ and $\ddot{r} \neq \ddot{s}$
          do
            if $\mathcal{N}_r + \tau \leq \theta\max(\|r|_1,\|v\|_1)$ then
              remove $\ddot{r}$ from $M$
            $(\ddot{r},\mathcal{N}_r,\|r\|_1) = M$.next
        if $\ddot{r} = \ddot{s}$
          then $\{M \leftarrow (\ddot{r},\mathcal{N}_r + W(\lambda_i^v),\|r\|_1)$
          else if $W(\lambda_i^v) + \tau \geq \theta\max(\|s\|_1,\|v\|_1)$
          then Insert $(\ddot{s},W(\lambda_i^v),\|s\|_1)$ at current position in M
        while not at end of $M$
          do
            $(\ddot{r},\mathcal{N}_r,\|r\|_1) = M$.next
            if $\mathcal{N}_r + \tau < \theta\max(\|r\|_1,\|v\|_1)$ then
              remove $\ddot{r}$ from $M$
    Report $(\ddot{s},\mathcal{N}_s,\|s\|_1) \in M$ s.t. $\mathcal{N}_s \geq \theta\max(\|s\|_1,\|v\|_1)$
```

`first` algorithm with the only difference being that once the potential score of an unseen candidate (a candidate that might be included in all remaining lists) is smaller than the query threshold, the algorithm reverts to random access mode that probes the remaining lists to complete the scores of all candidates left in the candidate set. If an index on string identifiers is available on each of the lists in the suffix, the algorithm needs to perform, on average, one random access per candidate per remaining list. If an index is not available, then assuming that lists are sorted in increasing norm order, the algorithm keeps track of the smallest and largest norm in the candidate set, seeks to the first list element with norm larger equal to the smallest norm and scans each list as deep as the largest norm. Alternatively, binary search can be performed. This strategy will help potentially prune a long head and tail from each list in the suffix. Alternatively, if the size of the remaining candidate set is small after the heaviest first strategy on the shortest lists terminates, the algorithm can simply retrieve the strings and compute the actual similarity.

Notice that in case of uniform token weights, it is meaningless to order lists according to weights. In this special case lists are simply sorted in increasing order of their lengths and all the algorithms discussed so far will work without any modifications, with similar benefits. This results in a shortest-first prioritization of lists, which is of independent interest, as was already mentioned above. In addition, all L_p-norm filters are still applicable: L_1-norm reduces to the L_0-norm and L_2-norm reduces to the square root of the L_0-norm.

Surprisingly, the `heaviest first` strategy can also help prune candidates for Weighted Intersection similarity, based on the observation that strings containing the heaviest query tokens are more likely to exceed the threshold. Let query string $v = \lambda_1^v \cdots \lambda_m^v$ and threshold $0 < \theta \leq \sum_{i=1}^m W(\lambda_i^v)$, and assume without loss of generality that lists are sorted in decreasing order of token weights (i.e., $W(\lambda_1^v) \geq \ldots \geq W(\lambda_m^v)$). Assume that there exists a string s that is contained only in a suffix $L(\lambda_k^v), \ldots, L(\lambda_m^v)$ of token lists, whose aggregate weight $\sum_{i=k}^m W(\lambda_i^v) < \theta$. Clearly, such a string cannot exceed the query threshold. An immediate conclusion is that

---

Lemma 6.16. Let query $v = \lambda_1^v \cdots \lambda_m^v$ s.t. $W(\lambda_1^v) \geq \cdots \geq W(\lambda_m^v)$, and threshold $0 < \theta \leq \sum_{i=1}^m W(\lambda_i^v)$. Let

$$\pi = \arg \max_{1 \leq \pi \leq m} \sum_{i=\pi}^{|v|} W(\lambda_i^v) \geq \theta.$$

Let $\mathcal{P}_\theta(v) = \{\lambda_1^v, \ldots, \lambda_\pi^v\}$ be the prefix of v and $\mathcal{S}_\theta(v) = \{\lambda_{\pi+1}^v, \ldots, \lambda_m^v\}$ be the suffix of v. Then, for any string s s.t. $s \cap \mathcal{P}_\theta(v) = \emptyset, \mathcal{I}(s,v) < \theta$.

Proof. The proof appears in Section 7.1. □

Hence, the only viable candidates have to appear in at least one of the lists in the prefix $L(\lambda_1^v), \ldots, L(\lambda_\pi^v)$. The `heaviest first` algorithm exhaustively scans the prefix lists to find all viable candidates and then probes the suffix lists to complete their scores using random access.

6.2 Top-k Selection Queries

A top-k selection query has to return the k data strings most similar to the query (assuming no ties; if ties exist then algorithms either return k strings by resolving ties arbitrarily, or return all ties).

6.2.1 Lists Sorted in String Identifier Order

It is easy to see that the `multiway merge` algorithm cannot be used to answer top-k queries efficiently. This is mainly because the algorithm identifies answers in order of their string identifier rather than in order of their similarity with the query. One could use the `multiway merge` algorithm in combination with a heap data structure to keep track of the current top-k answers, but in order to guarantee that the true top-k answers are returned, all token lists corresponding to query tokens have to be exhaustively examined. In other words, there is no guarantee that the answer with the largest similarity is not also the one with the largest string identifier.

Nevertheless, one can still use the `optimized multiway merge` algorithm described in Section 6.1.1 in order to potentially skip a large number of entries from every list, during the list merging step.

It is easy to see that the algorithms designed for all-match selection queries can be adapted for top-k queries, based on the observation that top-k queries are essentially all-match selection queries with an adaptively increasing similarity threshold θ. An efficient top-k algorithm has to identify a good approximation of the similarity of the k-th most similar string as fast as possible, in order to converge to the correct top-k answers as efficiently as possible. Clearly, given a query, it is impossible to know a priori what the k-th similarity will be. Hence, we can only resort to heuristics. A simple idea is to start merging lists until the similarity of k elements has been computed. Then, we populate a heap using those k candidates and use the k-th similarity as the query threshold. The algorithm proceeds by maintaining the top k elements in the heap as the similarity of new candidates is computed. The current, always increasing, k-th similarity can be used by the `optimized multiway merge` algorithm as a threshold for skipping elements from the lists and terminating early.

6.2.2 Lists Sorted in L_p-norm Order

As we argued for all-match selection queries, Weighted Intersection similarity cannot benefit from L_p-norm ordering, since the function itself does not depend on any string norm.

Consider Normalized Weighted Intersection. Let token lists be sorted in increasing order of L_1-norms. We modify the `nra` algorithm (see Algorithm 6.1.2) to maintain a heap H containing the k strings with the largest partial similarity computed so far from all strings appearing in candidate set M. The partial similarity of a string is clearly a *lower bound* on the true similarity (not to be confused with the best-case similarity computed using Lemma 6.4, which is an *upper bound* on the similarity). Recall that `nra` maintains a best-case similarity score $\mathcal{N}^f$ for a conceptual frontier string. Clearly $\mathcal{N}^f$ is an upper bound on the similarity of any string that has not been encountered in any of the lists yet. Let $\mathcal{N}^k$ be the k-th smallest similarity score in heap H. Straightforwardly, when $\mathcal{N}^f < \mathcal{N}^k$ there are no strings $s \notin M$ s.t. $\mathcal{N}(s, v) \geq \mathcal{N}^k$, hence the algorithm can stop inserting new strings in M. The algorithm can also evict from M strings whose best-case similarity is smaller than $\mathcal{N}^k$, as both $\mathcal{N}^k$ and the best-case similarity of strings in M keep getting tighter. After the frontier condition has been met, the algorithm needs to simply complete the partial similarity scores of strings already in M in order to determine the final top-k answers.

The important question is how to seed the algorithm with a good set of k initial candidates. The following observation is essential.

Observation 6.3. The answers that are potentially more similar to the query have L_1-norm equal to $\|v\|_1$. Strings with $\|s\|_1 = \|v\|_1$ potentially yield the maximum similarity which is equal to one. Intuitively, the potential similarity of candidate strings decreases as the L_1-norm of strings diverges from that of the query in either direction.

Notice that we refer to *potential* similarity given that $\|s\|_1 = \|v\|_1$ does not necessarily imply that $s = v$ since s and v might differ in a number of tokens that happen to have equal weights resulting in equal L_1-norms.

Based on Observation 6.3, the most promising candidates are the strings with L_1-norms clustered around the L_1-norm of the query. We can exploit this intuition in order to find a better k-th similarity approximation faster. In essence, instead of starting the round robin iterations of the `nra` algorithm from the beginning of the lists, we start from the first string s within each list s.t. $\|s\|_1 = \|v\|_1$. Then, we perform round robin iterations in both directions, once towards the end of the lists and once towards the beginning. When the algorithm has identified k strings, the k-th smallest partial similarity becomes the query threshold θ, and the algorithm can revert to normal all-match selection processing by scanning lists within the L_1-norm intervals given by Lemma 6.1. Notice that the algorithm is a heuristic. There is no guarantee regarding the quality of the k-th partial similarity discovered using this approach. In the best case the algorithm might immediately identify k strings equal to the query and stop. In the worst case the algorithm might be worse than vanilla `nra` (e.g., if it happens that the k-th most similar string is either the first or the last element in all lists).

Another heuristic is to use an optimistic approach that assumes that there are at least k data strings with similarity equal to one. Then, we can run an all-match selection query using $\theta = 1$. If the result contains fewer than k answers, we reduce θ deterministically and restart the previous query from where it stopped. We continue until k answers have been produced. Once again, depending on the data and the particular query this algorithm can either produce all answers within one iteration or might have to reduce the threshold to a very small value before identifying the k-th answer.

Similar arguments hold for the `optimized mutliway merge` algorithm based on sorting lists primarily by norms and secondarily by string identifiers.

6.2.3 Prioritization Across Lists

Consider weighted intersection similarity first. As was shown for all-match selection queries, the `threshold` based algorithms do not help prune any strings when considering simple weighted intersection, hence they do not work for top-k queries either. An alternative is to use the `heaviest first` algorithm based on prefix and suffix lists. Assume that lists are sorted in increasing order of string identifiers and that the algorithm processes lists in decreasing order of the corresponding token weights. Notice that for top-k queries there is no query threshold specified, which is necessary for determining the lists in the prefix/suffix sets. Nevertheless, we can assume that the initial threshold is zero (which implies that all query lists belong to the prefix set) and start scanning the first list. Each time an element is read, we probe the remaining lists and complete its similarity. Given that lists are sorted in increasing order of string identifiers we can use binary search to locate the strings in these lists (alternatively, an index on string identifiers can be built for each token list). Once we have retrieved k strings, we can use the k-th smallest similarity as a new threshold based on which the algorithm determines a new set of prefix/suffix lists. As the k-th similarity keeps improving, lists move from the prefix to the suffix set. The algorithm terminates once the similarity of all viable candidates in the prefix set has been computed, and the top-k answers have been found. Notice that once the k-th string has been retrieved the algorithm can stop doing random accesses to compute the actual similarity of strings, and it can revert to sequential accesses only. The algorithm terminates once all lists in the current prefix set have been fully traversed and the scores of all valid candidates have been computed. A better alternative here might be to perform random accesses for completing the score of all remaining viable candidates once the prefix lists have been fully traversed, or to simply retrieve the candidate strings and compute the similarity directly, in case that a small number of candidates remains. A version of the algorithm that performs random accesses only for the first k candidates is shown in Algorithm 6.2.1.

We can straightforwardly adapt the `heaviest first` top-k algorithm *for all normalized similarity measures*. The benefit of the `heaviest first` algorithm over the `nra` algorithms is the fact that, for disk based storage devices, it takes advantage of sequential accesses, one list at a time.

6.3 All-Match Join Queries

An all-match join query between two datasets S, R returns all pairs s, r in the cross-product $S \times R$ with similarity larger than a user specified threshold θ.

Join queries can be answered using a block nested loop join algorithm. Given two string datasets S and R, the algorithm scans S until main memory is full and subsequently scans R and identifies all pairs of strings s, r with similarity exceeding the query threshold. The obvious drawback of this algorithm is that it needs to repeatedly scan R as many times as the number of main memory partitions of S produced.

In certain cases it is more efficient to use inverted indexes to perform the join. If neither dataset is indexed, we can index either dataset and subsequently scan the un-indexed dataset and perform one selection query per string therein. If both inverted indexes can fit in main memory, the largest set is indexed. If the inverted index of only one dataset can fit in main memory, that set is indexed. If neither can fit in main memory, the largest set is partitioned such that the inverted index of each partition can fit in main memory. Then, the un-indexed dataset is scanned as many times as the number of partitions created. Notice that in order to create the inverted indexes, one of the datasets needs to be sorted first (depending on the type of index used, records will have to be sorted either in string identifier order or L_p-norm order).

A simple partitioning strategy is based on sorting both datasets S and R in increasing order of L_p-norms, and using norm filtering to reduce the number of strings examined from R in each iteration.

Algorithm 6.2.1: Heaviest First top-k(v,k)

```
Tokenize query: $v = \{(\lambda_1^v,1),\ldots,(\lambda_m^v,m)\}$ s.t. $W(\lambda_i^v) \geq W(\lambda_j^v), i < j$
$M$ is a list of $(\ddot{s},\mathcal{I}_s)$ pairs sorted in $\ddot{s}$ order
$H$ is a min-heap of at most $k$ pairs $(\ddot{s},\mathcal{I}_s),$ sorted in $\mathcal{I}_s$ order
$M \leftarrow \emptyset, \quad H \leftarrow \emptyset$
$\tau = \|v\|_1, \quad \theta = 0$
for $i \leftarrow 1$ to $m$
  if $\tau < \theta$ and $M$ is empty then break
  $\tau = \tau - W(\lambda_i^v)$
  $(\ddot{r},\mathcal{I}_r) \leftarrow M$.first
  while not at end of $L(\lambda_i^v)$
  do
    $(\ddot{s}) \leftarrow L(\lambda_i^v)$.next
    while $\ddot{r} \leq \ddot{s}$
    do
      if $\mathcal{I}_r + \tau \leq \theta$ then remove $\ddot{r}$ from $M$
      $(\ddot{r},\mathcal{I}_r) \leftarrow M$.next
    if $\ddot{r} = \ddot{s}$
    then
      $M \leftarrow (\ddot{r},\mathcal{I}_r + W(\lambda_i^v))$
      if $|H| < k$ then $H \leftarrow (\ddot{r},\mathcal{I}_r + W(\lambda_i^v))$
        else if $\ddot{r} \in H$ then $H$.pop$(\ddot{r}), H$
            $\leftarrow (\ddot{r},\mathcal{I}_r + W(\lambda_i^v))$
        else if $\mathcal{I}_r + W(\lambda_i^v) > \theta$ then $H$
            $\leftarrow (\ddot{r},\mathcal{I}_r + W(\lambda_i^v)), H$.pop
    else if $W(\lambda_i^v) + \tau > \theta$
    then
      Insert $(\ddot{s},W(\lambda_i^v))$ at current position in $M$
      if $|H| < k$ then $H \leftarrow (\ddot{s},W(\lambda_i^v))$
        else if $W(\lambda_i^v) > \theta$ then $H \leftarrow (\ddot{s},W(\lambda_i^v)), H$.pop
    while not at end of $M$
    do
      $(\ddot{r},\mathcal{N}_r,\|r\|_1) = M$.next
      if $\mathcal{N}_r + \tau < \theta \max(\|r\|_1,\|v\|_1)$ then
        remove $\ddot{r}$ from $M$
  Report $(\ddot{s},\mathcal{I}_s) \in H$
```

Algorithm 6.3.1: SORTED BLOCK NESTED LOOP JOIN(S,R,θ)

```
Sort $S, R$ in decreasing $L_1$-norm order
$I$ is an empty inverted index of string tokens
$s_f \leftarrow$ first string in $S$
for each $s \in S$
    if memory is available
        then $I \leftarrow s$
             $s_l \leftarrow s$
        else for each $r \in R$ s.t. $\theta\|s_f\|_1 \leq \|r\|_1 \leq \frac{\|s_l\|_1}{\theta}$
                 do $I$.Heaviest First($r$,$\theta$)
             $I \leftarrow \emptyset$,  $I \leftarrow s$
             $s_f \leftarrow s$
```

Once both datasets are sorted, we simply index the strings in S until no more memory is available and keep a pointer to the first and last strings indexed s_f, s_l. Then we switch to a probe only phase and process the strings in R in sorted order, within the appropriate norm intervals only (e.g., assuming Jaccard similarity, within $[\theta\|s_f\|_1, \|s_l\|_1/\theta]$). After the probe phase is complete, we discard the partial index of S and seek back to element s_l to continue indexing from where we left off. The drawback of this algorithm is the extra cost of having to sort R. The algorithm is shown in Algorithm 6.3.1.

More involved partitioning strategies that try to group strings into clusters of similar strings can also be designed. In general, the partitioning strategy used, significantly affects the performance of join processing. Notice that the hashing based partitioning approaches that are used for equality joins cannot be applied for similarity joins because this would typically imply that each record would need to be assigned to multiple partitions. A good partitioning strategy should minimize the number of partitions a record is assigned to, without making any given partition large.

The only previously proposed partitioning algorithm is tailored for weighted intersection and works in two phases. Let S and R be the datasets to be joined, and let an inverted index tailored for all-match selection queries for weighted intersection similarity for S and R have size W_s and W_r respectively. Assume that the largest inverted index that can fit in main memory has size M, where $M < W_s < W_r$. The first phase builds a compressed inverted index for R that can fit in main memory. The compressed inverted index is used to identify qualifying clusters for each string in S. Subsequently, a fine-grained inverted index is built on the qualifying clusters and used to answer the join query.

The compressed inverted index is built by grouping together strings with several overlapping tokens into clusters. The idea is to use pointers to the clusters of strings in the inverted index (instead of pointers to the strings themselves), thereby decreasing the length of the inverted lists. Let C be a cluster of strings. The cluster is represented by the union of tokens in all strings $s \in C$. Let Λ_C denote the union of tokens of cluster C. All inverted lists of the compressed inverted index corresponding to tokens in Λ_C contain a pointer to C. To answer a query, first the compressed inverted index is scanned as usual to find all qualifying clusters; a qualifying cluster needs to have weighted intersection similarity with the query string larger than the threshold θ (i.e., $\mathcal{I}(v,C) \geq \theta$). Then a fine-grained query is executed on the strings of each qualifying cluster.

Initially, we need to build a good set of string clusters $\mathcal{C}$ such that each cluster in $\mathcal{C}$ contains similar strings, and all clusters have approximately equal sizes. The algorithm first determines the maximum number of clusters and the average number of strings per cluster that the compressed inverted index can hold in main memory. The algorithm estimates the number of clusters as $N = \frac{|R| \times M}{W_l}$, and the average number of strings per cluster as $A = N$ under the assumption that $M \geq \sqrt{W_l}$ (if this assumption is violated, the algorithm can do recursive partitioning).

Next, the clusters are constructed as follows. The algorithm performs a linear scan of R, and for each string $r \in R$, it chooses a cluster C, which can be either an existing cluster with enough overlap with r or a new cluster if $|\mathcal{C}| < N$. Finding a cluster for each string $r \in R$ is based on a similarity function between r and that cluster. A simple definition for the similarity of string r and cluster C is their weighted intersection similarity $\mathcal{I}(r, C)$. An interesting observation here is that we can use the partially built compressed inverted index to identify which existing cluster is the most similar to r. This is accomplished by simply running a top-k selection query on the compressed inverted index with query string r. In this case the top-k algorithm needs to identify a cluster that contains fewer than A strings. We can identify the top-k matches incrementally until a suitable cluster C is found. In addition, if $\mathcal{I}(r, C) < \phi$, for a given threshold ϕ and $|\mathcal{C}| < N$ a new cluster is created instead. Threshold ϕ can be computed as a function of the average number of strings per cluster and the average number of tokens per string in $S \cup R$. Otherwise, r is assigned to cluster C. A pointer to C is inserted in all the inverted lists of the compressed inverted index corresponding to tokens in r that are not already contained in Λ_C. The algorithm also stores one record $(\ddot{r}, C)$ per string r in a hash table, which is needed in the querying phase. Finally, the algorithm scans all strings $s \in S$ and for each s queries the compressed inverted index to find all qualifying clusters C such that $\mathcal{I}(s, C) \geq \theta$. These are the clusters containing strings that need to be joined with s. The algorithm stores one record $(C, \ddot{s})$ per qualifying cluster in a hash table. A detail here is that instead of computing the weighted intersection between a string and the clusters, we can use Jaccard similarity without affecting the correctness of the algorithm. Jaccard similarity will prevent large clusters from getting too large too fast.

In the second phase the algorithm creates one fine-grained inverted index per cluster and computes the join results of each string $s \in S$ that needs to join with this cluster, by using the string/cluster and qualifying-cluster/string hash tables built in phase one. The compressed inverted index can be discarded at this stage. The algorithm uses the available memory M to build an inverted index for a set of clusters $C \in \mathcal{C}$ that fits in main memory. Each set of clusters is treated as one join partition.

Notice that the algorithm described above cannot be extended for Jaccard, Dice, or Cosine similarity. The main reason is that the idea is based on a monotonicity property: If $\mathcal{I}(v, C) < \theta$ then there does not exist $s \in C$ s.t. $\mathcal{I}(v, s) \geq \theta$. It is easy to see that this property is not true for Jaccard, Dice, or Cosine similarity.

6.4 All-Match Self-join Queries

An all-match self-join query for a dataset S returns all pairs s, s' in the cross-product $S \times S$ with similarity larger than a user specified threshold θ.

In case of self-join queries, if the dataset is not already indexed, an alternative approach is based on indexing the dataset incrementally. The procedure is shown in Algorithm 6.4.1.

The algorithm works for all similarity measures discussed in Section 2.2. In addition, the `multiway merge` algorithm can be replaced with any other selection algorithm (like `nra`, `optimized multiway merge`, and `heaviest first`). Notice that for Algorithm 6.4.1 to work we have to process the strings in a well defined order. For the `multiway merge` algorithm the strings have to be processed in increasing or decreasing order of string identifiers. For `nra`, `optimized multiway merge`, and `heaviest first`, they have to be processed in increasing order of L_1-norms for Jaccard and Dice, and L_2-norm for Cosine similarity.

Certain optimizations can be applied when the `nra`, `optimized multiway merge`, or `heaviest first` algorithms are used. Concentrating on Jaccard, assume that strings are processed in increasing order of their L_1-norm. As a new string s is processed, we know that all subsequent strings will have L_1-norm larger than or equal to the norm of s. Hence, by using Lemma 6.11, we can safely remove from the head of all inverted lists all entries corresponding to strings with L_1-norm smaller than $\theta\|s\|_1$. Clearly, none of the strings corresponding to these entries will be candidates for any subsequent string. This helps save space by decreasing the size of the inverted lists. Similar observations hold for Dice and Cosine similarity.

Algorithm 6.4.1: SELF-JOIN(S,θ)

```
$I$ is an empty inverted index of string tokens
for each $s \in S$
    $I$.Multiway Merge($s$,$\theta$)
    $I \leftarrow s$
```

In the rare case that the dataset is too large for the inverted index to fit in main memory even with the index size reduction optimization mentioned above (this could happen when a very large number of strings have equal norms), a simple modification of the algorithm is to index strings until no more main memory is available and keep a pointer to the last string indexed, similarly to Algorithm 6.3.1. Then the algorithm switches to a probe only phase and scans the rest of the dataset within the appropriate norm bounds, producing all matching pairs. Finally, the algorithm discards the existing index and continues indexing strings from where it left off. The drawback of this algorithm is that it might have to scan parts of the dataset multiple times. One can also modify the partitioning strategy described in Section 6.3 to do the self-join (with some possible optimizations) in the case of weighted intersection similarity.

6.5 Top-k Join and Self-join Queries

A top-k join query between two datasets S, R returns the k pairs s, r with similarity larger than any other pair in the cross-product $S \times R$. A top-k self-join query for a dataset S returns the k pairs s, s' (s.t. $\ddot{s} \neq \ddot{s}'$) with similarity larger than any other pair in the cross-product $S \times S$.

The difficulty with top-k queries is that the similarity of the k-th answer is not known in advance. A simple strategy for evaluating top-k join (self-join) queries is to quickly identify k candidates and use the k-th similarity as a threshold θ in order to answer the query using an all-match join (self-join) algorithm. As the algorithm proceeds and the similarity of the current k-th candidate converges toward the similarity of the k-th most similar string, the search becomes more effective. All the algorithms discussed for all-match join and self-join queries can be used, with slight modifications, to answer top-k queries.

6.6 Index Construction

Inverted index construction might require several linear scans of the data in some scenarios. The first step for constructing the inverted index is to sort the data either in increasing/decreasing order of string identifiers or increasing/decreasing order of L_p-norms, depending on the index being built.

Assume that the input datasets are too large to fit in memory. If the sort order is according to string identifiers then an external sort algorithm can be used to sort the data directly. Then, the token lists of the inverted index are simply populated incrementally, as new strings are processed in string identifier order.

On the other hand, if the sort order is according to L_p-norms, then the algorithm has to compute the norms of each string first. Depending on the nature of token weights used, computing the norms might require at least one additional scan of the data. Consider for example *idf* based weights. In order to compute *idf* based L_p-norms, the *idf* of each token has to be computed first. Computing *idf*s requires scanning the data to compute the document frequency of each token. Once the *idf*s have been computed, an external sort algorithm can be used to sort the data.

An alternative approach is to avoid sorting the data initially. Instead, we can incrementally populate the token lists as strings are processed and tokenized out of order. After all strings are processed we can sort each token list independently. The drawback of this approach is that, provided that the cardinality of the token universe Λ is very large, we might have to actively manage a very large number of token lists, which can be very expensive. On the other hand, if the cardinality of Λ is expected to be small then this simplistic approach can be faster than external sorting of the input dataset. Notice also that if token lists are maintained as B-trees, out of order processing of strings will immediately result in properly sorted token lists. Nevertheless, in this case every single insertion of a string entry in any token list results in a random access. A better approach is, once again, to use external sorting to sort the input dataset first, and bulk-load the B-trees corresponding to the token lists as a final step.

6.7 Index Updates

Typically, inverted indexes are used for mostly static data. More often than not, token lists are represented either by simple arrays in main memory, or flat files in secondary storage, since these representations are space efficient and also enable very fast sequential processing of list elements. On the other hand, inserting or deleting elements in the token lists becomes expensive.

In situations where fast updates are important, token lists can be represented by linked lists or binary trees in main memory, or B-trees in external memory. The drawback is the increased storage cost both for main memory and external memory structures, and also the fact that external memory data structures do not support sequential accesses over the complete token lists as efficiently as flat files. On the other hand, the advantage is that insertions and deletions can be performed very efficiently.

A subtle but very important point regarding updates arises when considering token lists that contain L_p-norms. Recall that L_p-norms are computed based on token weights. For certain token weighing schemes, like *idf* weights, token weights depend on the total number of strings overall and the total number of strings containing the given token. As strings get inserted, deleted, or updated, both the total number of strings and the number of strings containing a given token might change, and as a result the weight of tokens can change as well as the L_p-norms of strings that are not directly related with a given insertion, deletion, or modification.

Consider for example *idf* weights, where the weight of a token is a function of the total number of strings $|S|$ (see Definition 2.11). A single string insertion or deletion will affect the *idf*s of all tokens, and hence render the L_p-norms of all entries in the inverted lists in need of updating. For example, if we insert a string that contains token λ, the *idf* weight $W(\lambda)$ with change, and as a consequence, the L_p-norm of all other strings already appearing in the index that happen to contain λ will change. Hence, a single string update can result in a cascading update effect on the token lists of the inverted index that could be very expensive.

To alleviate the cost of updates in such scenarios a technique based on delayed propagation of updates has been proposed. The idea is to keep stale information in the inverted lists as long as certain guarantees on the answers returned can be provided. Once the computed L_p-norms have diverged far enough from the true values such that the guarantees no longer hold, the pending updates are applied in batch.

The propagation algorithm essentially computes lower and upper bounds between which the weight of individual tokens can vary, such that a selection query with a well defined reduced threshold $\theta' < \theta$ will return all true answers (i.e., will not result in any false negatives). The additional cost is an increased number of false positives. The reduced threshold θ' is a function of θ, the particular weighing scheme, and the relaxation bounds. The tighter the relaxation bounds chosen, the smaller the number of false positives becomes.

6.8 Discussion and Related Issues

It is very difficult to determine whether any of the aforementioned strategies performs better than the rest, across the board. The relative cost savings between sorting according to string identifiers, sorting according to L_p-norms, or prioritizing lists, heavily depend on a variety of factors most important of which are: the algorithm used, the weighing scheme used, the specific query (whether it contains tokens with very long lists for which the L_p-norm filtering will yield significant pruning), the overall token distribution of the data strings, the storage medium used to store the lists (e.g., disk drive, main memory, solid state drive), and the type of compression used for the lists, if any. Choosing the right algorithm depends on specific application and data characteristics.

It is important to state here that most list compression algorithms are designed for compressing string identifiers, which are natural numbers. The idea being that natural numbers in sorted order can easily be represented by delta coding. In delta coding only the first number is stored, and each consecutive number is represented by its distance to the previous number. This representation decreases the magnitude of the numbers stored (since deltas are expected to have small magnitude when the numbers are in sorted order) and enables very efficient compression. A significant problem with all normalized similarity functions is that, given arbitrary token weights, token lists have to store real valued L_1-norms (or L_2-norms for Cosine similarity). Lists containing real valued attributes cannot be compressed as efficiently as lists containing only natural numbers. Of course one could still use well-known compression techniques (e.g., Golomb coding and run length encoding).

Another list compression approach is to completely discard certain inverted lists based on the size of lists, their contribution to overall string similarity given the weight of the tokens they correspond to, the effect of discarded lists on list merging efficiency given a query workload, etc. Another intuitive idea is to combine token lists that are very similar to each other, using a single inverted list to store the union of the initial lists. Finally, one can use variable length grams (as opposed to fixed length q-grams) to better capture the distribution of short substrings within the data strings, which can potentially yield significant space and performance improvements, by eliminating very frequent, short grams and replacing them with longer, less frequent grams.

6.9 Related Work

Baeza-Yates and Ribeiro-Neto [8] and Witten et al. [71] discuss selection queries for various similarity measures using `multiway merge` and inverted indexes. The authors also discuss various compression algorithms for inverted lists sorted according to string identifiers. Behm et al. [11] discuss specialized lossy compression algorithms, while Li et al. [49] and Yang et al. [76] present compression algorithms based on alternative string tokenizations that take advantage of variable length grams. Li et al. [48] introduced the optimized version of the `multiway merge` algorithm for the special case of unit token weights. Linderman [50] proposed the optimized version and early termination condition of the `multiway merge` algorithm for general weights. Knuth [43] gives a more general discussion of the `multiway merge` algorithm in the context of external sorting. Linderman [50] also proposed the optimized version of the `multiway merge` algorithm based on sorting both by norm and string identifiers.

Koudas et al. [44] introduced an SQL based framework for identifying similar strings based on Cosine similarity and L_p-norms. Various strategies for evaluating set intersection queries based on sorting sets according to their L_p-norms have been made by Bayardo et al. [10].

Hadjieleftheriou et al. [33] introduced further optimizations for the special case of Cosine similarity. L_p-norm based filtering has also been used by Li et al. [48] and Xiao et al. [75].

A detailed analysis of threshold based algorithms is conducted by Fagin et al. [29]. Improved termination conditions for these algorithms are discussed by Sarawagi and Kirpal [60], Bayardo et al. [10] and Hadjieleftheriou et al. [33]. The `heaviest first` algorithm for weighted intersection based on prefix and suffix lists is based on ideas introduced by Sarawagi and Kirpal [60] and Chaudhuri et al. [19]. The same algorithm, assuming unit token weights, was extended for arbitrary prefix lengths by Li et al. [48]. The `heaviest first` algorithm for arbitrary token weights was introduced by Hadjieleftheriou et al. [33].

The partitioning strategy for all-match join queries with memory constraints was proposed by Sarawagi and Kirpal [60]. The incremental indexing for self-join queries was first proposed by Sarawagi and Kirpal [60]. The improved algorithm for Jaccard, Dice, and Cosine similarity based on deleting elements from the top of token lists was proposed by Bayardo et al. [10]. The block nested loop self-join algorithm in case of memory constraints was also proposed by Bayardo et al. [10]. Various techniques for answering top-k queries using inverted indexes and the `multiway merge` strategy were discussed by Vernica and Li [69].

Efficient online updates for inverted indexes have been studied extensively by Lester et al. [46]. Propagating updates for inverted indexes stored in a relational database were addressed by Koudas et al. [45]. Index construction and update related issues with regard to L_p-norm computation is discussed in detail by Hadjieleftheriou et al. [34].

7

Algorithms for Set Based Similarity Using Filtering Techniques

So far we have discussed several algorithms based on building inverted indexes on all the tokens contained in the data strings. A fundamentally different indexing approach is based on the observation that for a variety of similarity measures one can easily derive upper bounds on the similarity by examining only a small number of tokens from each string. We call algorithms for computing upper bounds on the similarity *filtering techniques*. Given a query string, once a candidate set of answers has been produced using a filtering technique, a refinement step is performed to compute the exact similarity and return the correct answers. This *filter and refine* approach, as it is commonly called, builds an inverted index of substantially reduced size over a small subset of tokens per string. The advantage of filtering techniques is that they can evaluate join queries very efficiently. The drawback is that the filtering phase might produce a large number of false positives that will have to be pruned by verification during the refinement step. Verification might be expensive, depending on the similarity function used. Here, we focus only on filtering techniques that do not result in any false negatives. We do not cover synopses data structures and probabilistic algorithms, like min-wise independent permutations and locality sensitive hashing that can discard valid answers.

7.1 The Prefix Filter

Let $\prec$ be a total ordering of the token universe Λ, such that for $\lambda_1, \lambda_2 \in \Lambda, \lambda_1 \prec \lambda_2 \Leftrightarrow W(\lambda_1) \geq W(\lambda_2)$ (i.e., tokens are sorted in non-increasing order of their weights). Consider Weighted Intersection similarity first and assume that we are treating strings as sets (disregarding token frequency and position) for simplicity of presentation. The prefix signature of a string is defined as

Definition 7.1 (Weighted Intersection Prefix). Let $s = \{\lambda_1^s, \ldots, \lambda_m^s\}$ represented as a set of tokens, where without loss of generality tokens are sorted in increasing $\prec$ order (i.e., $\lambda_1^s \prec \cdots \prec \lambda_m^s$). Let θ be a pre-determined minimum possible query threshold for Weighted Intersection similarity. Let λ_π^s be the token in s s.t.

$$\pi = \arg \max_{1 \leq \pi \leq m} \sum_{i=\pi}^{m} W(\lambda_i^s) \geq \theta.$$

The prefix $\mathcal{P}_\theta^{\mathcal{I}}(s)$ of string s is defined as $\mathcal{P}_\theta^{\mathcal{I}}(s) = \{\lambda_1^s, \ldots, \lambda_\pi^s\}$, and the suffix $\mathcal{S}_\theta^{\mathcal{I}}(s)$ is defined as $\mathcal{S}_\theta^{\mathcal{I}}(s) = \{\lambda_{\pi+1}^s, \ldots, \lambda_m^s\}$.

For example, the prefix and suffix of string $s = \{\lambda_1^s, \ldots, \lambda_m^s\}$ are shown in Figure 7.1.

It holds that

Lemma 7.1. Given two string prefixes $\mathcal{P}_\theta^{\mathcal{I}}(s), \mathcal{P}_\theta^{\mathcal{I}}(r)$

$$\mathcal{I}(s,r) \geq \theta \Rightarrow \mathcal{P}_\theta^{\mathcal{I}}(s) \cap \mathcal{P}_\theta^{\mathcal{I}}(r) \neq \emptyset.$$

Proof. The intuition behind the proof is shown in Figure 7.2. The proof is by contradiction. Let $\mathcal{I}(s,r) \geq \theta$ and $\mathcal{P}_\theta^{\mathcal{I}}(s) \cap \mathcal{P}_\theta^{\mathcal{I}}(r) = \emptyset$. Notice that

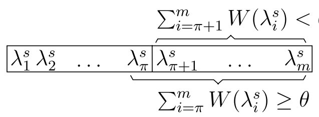

$\sum_{i=\pi+1}^{m} W(\lambda_i^s) < \theta$
$\lambda_1^s\ \lambda_2^s\ \ldots\ \lambda_\pi^s$
$\lambda_{\pi+1}^s\ \ldots\ \lambda_m^s$
$\sum_{i=\pi}^{m} W(\lambda_i^s) \geq \theta$

Fig. 7.1 The prefix and suffix of string $s = \{\lambda_1^s, \ldots, \lambda_m^s\}$. Without loss of generality, it is assumed that $W(\lambda_1^s) \geq \ldots \geq W(\lambda_m^s)$.

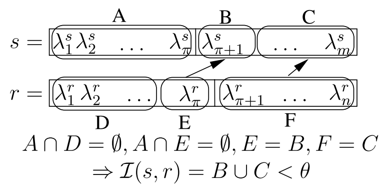

A
B
C
$s =$
$\lambda_1^s \lambda_2^s \ldots \lambda_\pi^s$
$\lambda_{\pi+1}^s$
$\ldots \lambda_m^s$
$r =$
$\lambda_1^r \lambda_2^r \ldots$
λ_π^r
$\lambda_{\pi+1}^r \ldots \lambda_n^r$
D
E
F
$A \cap D = \emptyset, A \cap E = \emptyset, E = B, F = C$
$\Rightarrow \mathcal{I}(s, r) = B \cup C < \theta$

Fig. 7.2 If two prefixes have empty intersection then, at best, even if a suffix matches completely, by construction the total intersection weight cannot be larger than θ.

since tokens are sorted in decreasing order of weights one of the following holds:

(1) If $\mathcal{P}_\theta^{\mathcal{I}}(s) \cap \mathcal{S}_\theta^{\mathcal{I}}(r) \neq \emptyset$, then $\mathcal{P}_\theta^{\mathcal{I}}(r) \cap \mathcal{S}_\theta^{\mathcal{I}}(s) = \emptyset$. By definition, $\|\mathcal{S}_\theta^{\mathcal{I}}(r)\|_1 = \sum_{\lambda_i^r \in \mathcal{S}_\theta^{\mathcal{I}}(r)} W(\lambda_i^r) < \theta$. Hence,

$$\begin{aligned}\mathcal{I}(s,r) &= \|\mathcal{P}_\theta^{\mathcal{I}}(r) \cap \mathcal{P}_\theta^{\mathcal{I}}(s)\|_1 + \|\mathcal{P}_\theta^{\mathcal{I}}(r) \cap \mathcal{S}_\theta^{\mathcal{I}}(s)\|_1 \\ &\quad + \|\mathcal{S}_\theta^{\mathcal{I}}(r) \cap \mathcal{P}_\theta^{\mathcal{I}}(s)\|_1 + \|\mathcal{S}_\theta^{\mathcal{I}}(r) \cap \mathcal{S}_\theta^{\mathcal{I}}(s)\|_1 \\ &= \|\mathcal{S}_\theta^{\mathcal{I}}(r) \cap \mathcal{P}_\theta^{\mathcal{I}}(s)\|_1 + \|\mathcal{S}_\theta^{\mathcal{I}}(r) \cap \mathcal{S}_\theta^{\mathcal{I}}(s)\|_1 \\ &\leq \|\mathcal{S}_\theta^{\mathcal{I}}(r)\|_1 < \theta.\end{aligned}$$

(2) If $\mathcal{P}_\theta^{\mathcal{I}}(r) \cap \mathcal{S}_\theta^{\mathcal{I}}(s) \neq \emptyset$, then $\mathcal{P}_\theta^{\mathcal{I}}(s) \cap \mathcal{S}_\theta^{\mathcal{I}}(r) = \emptyset$. Hence, similarly

$$\mathcal{I}(s,r) \leq \|\mathcal{S}_\theta^{\mathcal{I}}(s)\|_1 < \theta.$$

Both cases lead to a contradiction. □

The prefix filter reduces the problem of computing the intersection between the token sets of two strings to that of computing whether the prefixes of those strings have a non-empty intersection.

Notice that the prefix signature was defined in terms of strings represented as sets of tokens. The definition for sequences and frequency-sets is similar. For sequences, each token/position pair is considered as one element of the prefix. For frequency-sets, each token/frequency pair is considered as many times as the frequency of the token in the prefix (i.e., the prefix becomes a bag of tokens with a compressed token/frequency pair representation). It is important to consider each token multiple times in order to compute the minimum prefix (even though when computing the intersection of two prefixes multiplicity is not important), since considering each token only once, irrespective of its frequency, would allow extra tokens in the prefix which might increase the number of false positives at query time.

Extending the prefix filter algorithm for Normalized Weighted Intersection, Jaccard, Dice, and Cosine similarity is straightforward. Consider Jaccard similarity first. The prefix signature of a string is defined as

---

Definition 7.2 (Jaccard Prefix). The prefix $\mathcal{P}_\theta^{\mathcal{J}}(s)$ for Jaccard similarity is defined with respect to:

$$\pi = \arg\max_{1 \leq \pi \leq l} \sum_{i=\pi}^{l} W(\lambda_i^s) \geq \theta \|s\|_1.$$

The following is true:

Lemma 7.2. Given two string prefixes $\mathcal{P}_\theta^{\mathcal{J}}(s), \mathcal{P}_\theta^{\mathcal{J}}(r)$

$$\mathcal{J}(s,r) \geq \theta \Rightarrow \mathcal{P}_\theta^{\mathcal{J}}(s) \cap \mathcal{P}_\theta^{\mathcal{J}}(r) \neq \emptyset.$$

Proof. The proof is based on the fact that Jaccard similarity can be expressed as a weighted intersection constraint. Given two strings s, r

$$\mathcal{J}(s,r) \geq \theta \Rightarrow$$

$$\frac{\|s \cap r\|_1}{\|s\|_1 + \|r\|_1 - \|s \cap r\|_1} \geq \theta \Rightarrow$$

$$\|s \cap r\|_1 \geq \frac{\theta}{1+\theta}(\|s\|_1 + \|r\|_1).$$

From Lemma 6.11, $\mathcal{J}(s,r) \geq \theta \Rightarrow \|r\|_1 \geq \theta \|s\|_1$. Hence,

$$\mathcal{J}(s,r) \geq \theta \Rightarrow \|s \cap r\|_1 \geq \theta \|s\|_1.$$

The proof follows directly from the proof of Lemma 7.1 where the threshold for the weighted intersection similarity becomes $\theta\|s\|_1$ instead of θ. □

Careful observation reveals that given two strings s,r the condition $\mathcal{P}_\theta^{\mathcal{J}}(s) \cap \mathcal{P}_\theta^{\mathcal{J}}(r) \neq \emptyset$ for Jaccard similarity allows false positives that could be pruned by computing a tighter bound on the weighted intersection between two strings from the complete information provided in their prefixes, as opposed to using only their intersection.

Claim 7.3. Given two strings s,r

$$\|s \cap r\|_1 \leq \|\mathcal{P}_\theta^{\mathcal{J}}(s) \cap \mathcal{P}_\theta^{\mathcal{J}}(r)\|_1 + \max(\|\mathcal{S}_\theta^{\mathcal{J}}(s)\|_1, \|\mathcal{S}_\theta^{\mathcal{J}}(r)\|_1). \qquad (7.1)$$

Proof. Let $\mathcal{P}_\theta^{\mathcal{J}}(r) \cap \mathcal{S}_\theta^{\mathcal{J}}(s) = \emptyset$ (the case $\mathcal{P}_\theta^{\mathcal{J}}(s) \cap \mathcal{S}_\theta^{\mathcal{J}}(r) = \emptyset$ is symmetric). Then,

$$\begin{aligned}\|s \cap r\|_1 &= \|\mathcal{P}_\theta^{\mathcal{J}}(r) \cap \mathcal{P}_\theta^{\mathcal{J}}(s)\|_1 + \|\mathcal{P}_\theta^{\mathcal{J}}(r) \cap \mathcal{S}_\theta^{\mathcal{J}}(s)\|_1 \\ &\quad + \|\mathcal{S}_\theta^{\mathcal{J}}(r) \cap \mathcal{P}_\theta^{\mathcal{J}}(s)\|_1 + \|\mathcal{S}_\theta^{\mathcal{J}}(r) \cap \mathcal{S}_\theta^{\mathcal{J}}(s)\|_1 \\ &< \|\mathcal{P}_\theta^{\mathcal{J}}(r) \cap \mathcal{P}_\theta^{\mathcal{J}}(s)\|_1 + \|\mathcal{S}_\theta^{\mathcal{J}}(r)\|_1.\end{aligned}$$

Since either $\mathcal{P}_\theta^{\mathcal{J}}(r) \cap \mathcal{S}_\theta^{\mathcal{J}}(s) = \emptyset$ or $\mathcal{P}_\theta^{\mathcal{J}}(s) \cap \mathcal{S}_\theta^{\mathcal{J}}(r) = \emptyset$ has to be true (given that tokens are sorted in decreasing order of weights) the claim follows. □

Hence

Lemma 7.4. Given two string prefixes $\mathcal{P}_\theta^{\mathcal{J}}(s), \mathcal{P}_\theta^{\mathcal{J}}(r)$

$$\begin{aligned}\mathcal{J}(s,r) \geq \theta \Rightarrow \; &\|\mathcal{P}_\theta^{\mathcal{J}}(s) \cap \mathcal{P}_\theta^{\mathcal{J}}(r)\|_1 + \max(\|\mathcal{S}_\theta^{\mathcal{J}}(s)\|_1, \|\mathcal{S}_\theta^{\mathcal{J}}(r)\|_1) \\ &\geq \frac{\theta}{1+\theta}(\|s\|_1 + \|r\|_1).\end{aligned}$$

It is easy to see that similar conditions hold for Weighted Intersection, Normalized Weighted Intersection, Dice, and Cosine similarity. One drawback of using the tighter pruning conditions is that in order to evaluate the condition we need to know both the norm of the prefix of each data string and the norm of the suffix (or alternatively, the norm of the suffix can be computed as the norm of the whole string minus the norm of the prefix). This implies that we would have to store two norm values per entry, per inverted list, which increases the size of the inverted index significantly.

The prefix signature of a string for the other similarity measures follows.

Definition 7.3 (Prefix Signatures).

- Normalized Weighted Intersection: The prefix $\mathcal{P}^{\mathcal{N}}_{\theta}(s)$ for Normalized Weighted Intersection similarity is defined with respect to:

$$\pi = \arg \max_{1 \leq \pi \leq l} \sum_{i=\pi}^{l} W(\lambda_i^s) \geq \theta \|s\|_1.$$

- Dice: The prefix $\mathcal{P}^{\mathcal{D}}_{\theta}(s)$ for Dice similarity is defined with respect to:

$$\pi = \arg \max_{1 \leq \pi \leq l} \sum_{i=\pi}^{l} W(\lambda_i^s) \geq \frac{\theta}{2-\theta} \|s\|_1.$$

- Cosine: The prefix $\mathcal{P}^{\mathcal{C}}_{\theta}(s)$ for Cosine similarity is defined with respect to:

$$\pi = \arg \max_{1 \leq \pi \leq l} \sqrt{\sum_{i=\pi}^{l} W(\lambda_i^s)^2} \geq \theta \|s\|_2.$$

An important concern regarding prefix signatures is that the token ordering function significantly affects the pruning power of the prefix filter. Since the prefix filter evaluates the intersection between two prefix signatures, the rarer the tokens contained in the prefixes are, the lower the probability of a false positive intersection becomes. Rare tokens by definition appear in a small number of prefixes, while frequent tokens will tend to appear in the majority of prefixes. For that reason the prefix filter is often used in combination with frequency based token weights (e.g., *idf* weights).

In the special case where token weights are uniform (i.e., $\forall\lambda \in \Lambda$, $W(\lambda) = c$ for some constant c), notice that defining the order function $\prec$ to be a function of W is meaningless. Nevertheless, we can still effectively utilize prefix signatures, since $\|s\|_1 = \sqrt{\|s\|_2} = c\|s\|_0$ (where $\|s\|_0$ is the cardinality of the sequence, frequency-set, set). The reasoning of prefix filtering remains exactly the same as before by considering prefix and suffix *lengths* as opposed to *weights*. Once again, in order to improve the pruning effectiveness of the filter the order function $\prec$ is defined to be the decreasing token *idf* weight order (irrespective of the uniform weighing function W that is used to compute string similarity).

7.1.1 All-Match Selection Queries

To answer selection queries we can build an inverted index on the prefix signatures. Given a dataset of strings S, we build an inverted index on the signatures of all $s \in S$. The inverted index consists of one list per token present in the union of prefix signatures. Given a query string $v = \{\lambda_1^v, \dots, \lambda_m^v\}$ and query threshold $\theta' \geq \theta$, first we build the prefix signature $\mathcal{P}_{\theta'}(v)$. The query candidate answer set consists of strings contained in the union of token lists $L(\lambda_i^v)$ s.t. $\lambda_i^v \in \mathcal{P}_{\theta'}(v)$, that satisfy Lemma 7.4. After the candidate set has been produced, a refinement step computes the exact similarity between the query and all candidates.

Note here that when strings are represented as sequences, if the similarity of strings is intended to be interpreted as the similarity of their sequences, a token/position pair is a match if and only if it agrees both on the token and the position (recall that how we interpret the representation of strings affects the semantics of similarity). On the other hand, when representing strings as frequency-sets (which are essentially bags of tokens), a token/frequency pair is a match based on the token alone (the frequency is not important, since *one occurrence* of the token is enough to yield a non-empty intersection of prefixes).

A drawback of the prefix inverted index is that a minimum query threshold θ needs to be determined in advance. Smaller θ by construction implies longer prefixes on average, adversely affecting the size of the inverted index. Answering a query v with threshold $\theta' \geq \theta$

is straightforward, since the inverted index for minimum threshold θ indexes more information than that needed for answering queries with threshold $\theta' > \theta$. The index cannot be used to answer queries with threshold $\theta' < \theta$.

A more intuitive way of understanding the prefix inverted index is to see it as a special case of a full inverted index, where each token list is only partially populated by only those strings that contain that token specifically in their prefix. Now, we can use the *heaviest first* algorithm presented in Section 6 to answer queries, assuming that the algorithm traverses all lists up to $L(\lambda_\pi^v)$ only. Recall that *heaviest first* processes token lists in decreasing order of token weights (implicitly using the concept of prefix filtering). Straightforwardly, even if the algorithm traversed all query lists, the maximum possible remaining potential of unseen candidates after list $L(\lambda_\pi^v)$ is processed (refer to Lemmata 6.4, 6.7, 6.10, 6.14) would be smaller than the query threshold, instructing the algorithm to stop adding new candidates in the candidate set. Notice that for the prefix based inverted index, it is not beneficial to actually scan lists beyond $L(\lambda_\pi^v)$ since lists are only partially populated, hence not useful for computing tighter similarity upper bounds, for already discovered candidates (the entries of the candidates might not be present in those partially populated lists).

The fundamental difference between a prefix inverted index and a full inverted index using the *heaviest first* strategy is that the latter prunes strings incrementally using exactly the same concept as the prefix filter, but for always increasing prefix lengths (every new list processed essentially increases the length of the prefixes examined, by one token), and at the same time computes the actual similarity of candidate strings incrementally, ultimately eliminating the need for a refinement step. In addition, the full index can answer queries with arbitrary thresholds θ. These advantages come at the cost of increased index size and potentially slower query evaluation since the fully populated lists might contain strings that share a token with the query but not within the prefix, which would have otherwise not been considered. On the other hand, the pruning power of the full inverted index is at least as good or better than that of the prefix inverted index, since by virtue of fully populated lists, the algorithm computes much tighter similarity upper bounds every time a new list is processed. The prefix inverted index might have a performance advantage over the fully populated inverted index if the former fits in main memory while the latter does not, but this is data and query dependent and hard to quantify since the refinement step of the prefix filter would still have to access strings from secondary storage using random accesses.

Based on these observations, one could advocate for a hybrid approach that creates both a full and a prefix inverted index. Then, given a query, we scan the prefix index to find candidates using the tokens in the prefix of the query and also scan the full inverted index using the suffix of the query to compute tighter upper bounds. Notice that this approach cannot compute the actual similarity scores without also accessing the full lists of the tokens in the query prefix; some candidates identified in the first phase using the prefix lists might have a token in common with the query prefix that nevertheless appears in the suffix of the data string, and hence is not contained in the prefix inverted index.

7.1.2 Top-*k* Selection Queries

Prefix signatures cannot be used for answering top-k selection queries due to the fact that a minimum pre-determined threshold θ needs to be decided in advance in order to build the prefix inverted index. Since the k-th similarity between any data string and a given query in the worst case could be 0, for $\theta = 0$ the signature index will degenerate to a full inverted index.

7.1.3 All-Match Join Queries

Filtering techniques excel at processing join queries. Assume that we are performing a join between two string datasets and neither dataset is already indexed. If the full inverted index of either dataset cannot fit in main memory, answering the query requires partitioning the data appropriately. Instead, we can build an inverted index of significantly reduced size by using prefix signatures. More importantly, since we build the prefix inverted indexes on the fly, we can build signatures using θ exactly equal to the query threshold, which has the immediate consequence that we are building the minimal inverted index needed for identifying all answers with respect to any string in the second dataset. The actual algorithms are the same as those discussed in Section 6.3 and hence a detailed discussion here is omitted.

When using the sorted block nested loop join algorithm with prefix filters, there is an additional optimization that we can apply, which reduces the size of the prefixes indexed by taking advantage of the fact that the input datasets are sorted in increasing L_p-norm order.

Definition 7.4 (Jaccard Reduced Prefix). Let $s = \{\lambda_1^s, \dots, \lambda_m^s\}$ be a string with its tokens sorted in increasing $\prec$ order. Let θ be a Jaccard similarity threshold. The reduced prefix of s is defined as $\mathcal{P}^{\mathcal{J}}_{\frac{2\theta}{\theta+1}}(s)$.

For simplicity let $\phi = \frac{2\theta}{\theta+1}$. The following is true:

Lemma 7.5. Given two strings s, r s.t. $\|s\|_1 \leq \|r\|_1$, the reduced prefix $\mathcal{P}^{\mathcal{J}}_{\phi}(s)$ of s and the prefix $\mathcal{P}^{\mathcal{J}}_{\theta}(r)$ of r, it holds that

$$\mathcal{J}(s,r) \geq \theta \Rightarrow \mathcal{P}^{\mathcal{J}}_{\phi}(s) \cap \mathcal{P}^{\mathcal{J}}_{\theta}(r) \neq \emptyset.$$

Proof. Let $\mathcal{P}^{\mathcal{J}}_{\phi}(s) \cap \mathcal{P}^{\mathcal{J}}_{\theta}(r) = \emptyset$. It holds that

$$\begin{aligned}\|s \cap r\|_1 &= \|\mathcal{P}^{\mathcal{J}}_{\phi}(s) \cap \mathcal{P}^{\mathcal{J}}_{\theta}(r)\|_1 + \|\mathcal{P}^{\mathcal{J}}_{\phi}(s) \cap \mathcal{S}^{\mathcal{J}}_{\theta}(r)\|_1 \\ &\quad + \|\mathcal{S}^{\mathcal{J}}_{\phi}(s) \cap \mathcal{P}^{\mathcal{J}}_{\theta}(r)\|_1 + \|\mathcal{S}^{\mathcal{J}}_{\phi}(s) \cap \mathcal{S}^{\mathcal{J}}_{\theta}(r)\|_1 \\ &\leq \max(\|\mathcal{S}^{\mathcal{J}}_{\phi}(s)\|_1, \|\mathcal{S}^{\mathcal{J}}_{\theta}(r)\|_1).\end{aligned}$$

By definition

$$\mathcal{J}(s,r) = \frac{\|s \cap r\|_1}{\|s\|_1 + \|r\|_1 - \|s \cap r\|_1} \leq \frac{\|s \cap r\|_1}{2\|s\|_1 - \|s \cap r\|_1}.$$

Consider the two cases:

(1) Let $\|s \cap r\|_1 \leq \|\mathcal{S}^{\mathcal{J}}_{\phi}(s)\|_1 < \frac{2\theta}{\theta+1}\|s\|_1$. Then,

$$\mathcal{J}(s,r) < \frac{\frac{2\theta}{\theta+1}\|s\|_1}{2\|s\|_1 - \frac{2\theta}{\theta+1}\|s\|_1} = \theta.$$

(2) Let $\|s \cap r\|_1 \le \|\mathcal{S}^{\mathcal{J}}_{\theta}(r)\|_1$. We prove by contradiction. Assume that

$$\mathcal{J}(s,r) \ge \theta \Rightarrow \|s \cap r\|_1 \ge \frac{\theta}{\theta+1}(\|s\|_1 + \|r\|_1).$$

By definition

$$\|s \cap r\|_1 \le \|\mathcal{S}^{\mathcal{J}}_{\theta}(r)\|_1 \Rightarrow \|s \cap r\|_1 < \theta\|r\|_1.$$

Thus

$$\frac{\theta}{\theta+1}(\|s\|_1 + \|r\|_1) < \theta\|r\|_1 \Rightarrow \|s\|_1 < \theta\|r\|_1.$$

But, from Lemma 6.11

$$\mathcal{J}(s,r) \ge \theta \Rightarrow \|s\|_1 \ge \theta\|r\|_1,$$

which is a contradiction. Hence $\mathcal{J}(s,r) < \theta$. □

Lemma 7.5 states that if we know that strings are processed in increasing order of L_1-norms, then for every string indexed, we only need to index a reduced prefix of the string. Once a new string r is processed, it is guaranteed to have L_1-norm larger than or equal to all strings s already indexed, hence we can probe the index using the prefix $\mathcal{P}^{\mathcal{J}}_{\theta}(r)$ of r to find all candidate pairs using the reduced prefix principle.

It is easy to see that the index reduction principle does not apply to Weighted Intersection and Normalized Weighted Intersection similarity. In addition, it can be shown that it does not help reduce the prefixes for Dice similarity. On the other hand it can help reduce the prefixes for Cosine similarity.

Definition 7.5 (Cosine Reduced Prefix). The reduced prefix of string s is defined as $\mathcal{P}^{\mathcal{C}}_{\sqrt{\theta}}(s)$.

The following is true:

Lemma 7.6. Given two strings s, r s.t. $\|s\|_2 \le \|r\|_2$, the reduced prefix $\mathcal{P}^{\mathcal{C}}_{\sqrt{\theta}}(s)$ of s and the prefix $\mathcal{P}^{\mathcal{C}}_{\theta}(r)$ of r, it holds that

$$\mathcal{C}(s,r) \ge \theta \Rightarrow \mathcal{P}^{\mathcal{C}}_{\sqrt{\theta}}(s) \cap \mathcal{P}^{\mathcal{C}}_{\theta}(r) \ne \emptyset.$$

7.1.4 All-Match Self-join Queries

As with join queries, the algorithms for self-join queries are similar to the ones discussed in Section 6.4, where the full inverted index constructed during the incremental indexing phase is simply replaced by a prefix signature inverted index. Similarly, the prefix reduction principle discussed in Section 7.1.3 can be employed for self-join queries by sorting the input dataset in L_p-norm order and incrementally indexing only the reduced prefixes of strings (see Algorithm 6.4.1).

7.1.5 Top-k Join and Self-join Queries

Once again top-k join and self-join queries can be answered by using the all-match join and self-join algorithms with prefix inverted indexes instead of full inverted indexes. The algorithms simply identify any k candidates during initialization, and use the k-th smallest similarity as the initial threshold θ. Notice that in the worst case the initial k-th similarity can be 0, and the prefix inverted index will degenerate to a full inverted index (each prefix is the complete string). In fact, there is no better solution known when the input datasets do not fit in main memory.

Nevertheless, there exists another effective strategy when there is enough available memory to fit one dataset and its prefix inverted index in main memory. The strategy uses an optimistic approach that assumes that the k-th smallest similarity is the maximum possible similarity (e.g., one for Normalized Weighted Intersection, Jaccard, Dice, and Cosine similarity), and starts indexing prefixes incrementally, one token at a time. This strategy is effective only in main memory, since the algorithm has to repeatedly access each string when it is time to index its next prefix token. The top-k self-join algorithm for Jaccard similarity is shown as Algorithm 7.1.1. The same algorithm, with slight modifications, can be used to answer top-k join queries between two sets S, R as well (e.g., a simple way to do this is to run a slightly modified self-join algorithm on the union $S \cup R$). It can work for Weighted Intersection, Normalized Weighted Intersection, Dice and Cosine similarity.

In the beginning the algorithm indexes only the first token of each string. When a new token s_π from a given string s is indexed, the

Algorithm 7.1.1: Top-k Join(S, k)

```
$R$ is a min-heap initialized with the first $k$ pairs in $S$
$E$ is a max-heap initialized with one entry $\langle \ddot{s}, 1, 1.0 \rangle$ per $s \in S$
$L(\lambda_i^s)$ are empty token lists
Let $\mathcal{J}_k$ denote the similarity of the top element in $R$
while $E$ is not empty
    $\langle \ddot{s}, \pi, \mathcal{J}_s^\pi \rangle \leftarrow E$.pop
    if $\mathcal{J}_s^\pi \leq \mathcal{J}_k$
        then Break
    for all $(\ddot{r}, \|r\|_1) \in L(\lambda_\pi^s)$
        do if $\mathcal{J}_k \|r\|_1 \leq \|s\|_1 \leq \|r\|_1 / \mathcal{J}_k$
            then $R$.insert$(\mathcal{J}(s,r), (s,r))$
                 $R$.pop
    $\mathcal{J}_s^{\pi+1} = \frac{\|s\|_1 - \|\lambda_1^s \cdots \lambda_{\pi-1}^s\|_1}{\|s\|_1}$
    if $\mathcal{J}_s^{\pi+1} > \mathcal{J}_k$
        then $L(\lambda_\pi^s) \leftarrow (\ddot{s}, \|s\|_1)$
             $E$.push$(\langle \ddot{s}, \pi + 1, \mathcal{J}_s^{\pi+1})$
```

algorithm probes the existing index (as it is being built incrementally) to identify candidate pairs s, r, immediately computes the exact similarity $\mathcal{J}(s, r)$ for all r, and updates the top-k result heap as necessary. Then, the algorithm inserts s in token list $L(\lambda_\pi^s)$ if necessary, and also computes a hypothetical maximum similarity threshold between s and any unseen string that does not match with s on the already indexed tokens. The algorithm continues iteratively until there is no string s whose hypothetical maximum similarity threshold is larger than the k-th similarity in the result set.

In more detail, first the algorithm initializes a min-heap with k arbitrary pairs (e.g., the first k pairs in S) sorted according to their similarity and creates an event max-heap containing one entry per string in S. Each event $\langle \ddot{s}, \pi, \mathcal{J}_s^\pi \rangle$ is a triple consisting of a string identifier $\ddot{s}$, a prefix length π, and a similarity upper bound $\mathcal{J}_s^\pi$. The event heap is initialized with one event $\langle \ddot{s}, 1, 1.0\rangle$ per string $s \in S$, and maintained in decreasing order of $\mathcal{J}_s^\pi$.

The similarity upper bound for a given string s is computed by considering the best case scenario. Abusing notation, let $\mathcal{P}^\pi(s)$ ($\mathcal{S}^\pi(s)$) denote the prefix (suffix) of string s containing exactly the first π tokens (all remaining tokens). By construction, the tokens in the current indexed prefix $\mathcal{P}^{\pi'}(r)$ of any string r that is not already contained in the answer set, do not match any of the tokens in the already indexed prefix $\mathcal{P}^\pi(s)$ of s. Then, the best-case similarity upper bound between s and r is when all of the tokens in the current (unseen) suffix $\mathcal{S}^{\pi'}(r)$ of r match those in the unknown suffix $\mathcal{S}^\pi(s)$ of s. The potential Jaccard similarity becomes

$$\mathcal{J}(s,r) \leq \frac{\|\mathcal{S}^\pi(s)\|_1}{\|\mathcal{P}^\pi(s)\|_1 + \|\mathcal{P}^{\pi'}(r)\|_1 + \|\mathcal{S}^\pi(s)\|_1} \leq \frac{\|\mathcal{S}^\pi(s)\|_1}{\|s\|_1}.$$

Here, we need to compute the upper bound similarity of s with respect to an arbitrary string r, thus based only on information in s; dropping the term $\|\mathcal{P}^{\pi'}(r)\|_1$ from the denominator and maximizing the numerator yields the desired upper bound. Hence

$$\mathcal{J}_s^\pi = \frac{\|\mathcal{S}^\pi(s)\|_1}{\|s\|_1} = \frac{\|s\|_1 - \|\mathcal{P}^\pi(s)\|_1}{\|s\|_1}.$$

An interesting observation here is that if we assume *unit token weights* we can improve the similarity upper bound based on the fact that during each iteration of the algorithm the indexed prefixes consist of exactly the same number of tokens for all strings. In that case we know that $\pi' = \pi$ (i.e., $\|\mathcal{P}^{\pi'}(r)\|_1 = \|\mathcal{P}^\pi(s)\|_1$), hence the similarity upper bound becomes

$$\mathcal{J}_s^\pi = \frac{\|\mathcal{S}^\pi(s)\|_1}{\|s\|_1 + \|\mathcal{P}^\pi(s)\|_1}.$$

The algorithm proceeds by extracting the top event from the event heap, say $\langle \ddot{s}, \pi, \mathcal{J}_s^\pi\rangle$, and then scans inverted list $L(\lambda_\pi^s)$ to identify new candidates. The exact similarity of each candidate is computed and the top-k result heap is updated appropriately. Then, the algorithm needs to index the new token λ_π^s and compute the similarity upper bound $\mathcal{J}_s^{\pi+1}$ of s with any unseen string that does not agree with s

in any of the already indexed tokens $\lambda_1^s, \ldots, \lambda_\pi^s$. Then, it has to insert a new event $\langle \ddot{s}, \pi + 1, \mathcal{J}_s^{\pi+1} \rangle$ in the event heap, if and only if $\mathcal{J}_s^{\pi+1}$ is larger than the k-th similarity in the result heap. The algorithm stops once there are no more events to process or the k-th similarity in the result heap is larger than the similarity upper bound at the top of the event heap.

We can further improve the pruning efficiency of this algorithm by using the fact that the token lists are created incrementally in decreasing order of similarity upper bounds. Let string s with current indexed prefix $\mathcal{P}^\pi(s)$ and last prefix token λ_π^s. Let string r with current indexed prefix $\mathcal{P}^{\pi'}(r)$ whose first common token with s is token λ_π^s (hence $\lambda_\pi^s \in \mathcal{S}^{\pi'}(r)$). By definition, the intersection of the current prefixes is empty $\mathcal{P}^\pi(s) \cap \mathcal{P}^{\pi'}(r) = \emptyset$. Since λ_π^s is the first token in common between s and r, and tokens are processed in sorted weight order, it holds that

$$\|s \cap r\|_1 = \|\mathcal{S}^\pi(s) \cap \mathcal{S}^{\pi'}(r)\|_1 \leq \min(\|\mathcal{S}^\pi(s)\|_1, \|\mathcal{S}^{\pi'}(r)\|_1).$$

There are two cases to consider

$$\|s \cap r\|_1 \leq \begin{cases} \|\mathcal{S}^\pi(s)\|_1 = \|s\|_1 \mathcal{J}_s^\pi, & \text{if } \|s\|_1 \mathcal{J}_s^\pi \leq \|r\|_1 \mathcal{J}_r^{\pi'} \quad (1) \\ \|\mathcal{S}^{\pi'}(r)\|_1 = \|r\|_1 \mathcal{J}_r^{\pi'}, & \text{if } \|r\|_1 \mathcal{J}_r^{\pi'} \leq \|s\|_1 \mathcal{J}_s^\pi. \quad (2) \end{cases}$$

For case (1)

$$\begin{aligned} \|s \cup r\|_1 &\geq \|s\|_1 + \|r\|_1 - \|s\|_1 \mathcal{J}_s^\pi \\ &\geq \|s\|_1 + \|s\|_1 \frac{\mathcal{J}_s^\pi}{\mathcal{J}_r^{\pi'}} - \|s\|_1 \mathcal{J}_s^\pi \Rightarrow \\ \mathcal{J}(s,r) &= \frac{\|s \cap r\|_1}{\|s \cup r\|_1} \\ &\leq \frac{\|s\|_1 \mathcal{J}_s^\pi}{\|s\|_1 + \|s\|_1 \frac{\mathcal{J}_s^\pi}{\mathcal{J}_r^{\pi'}} - \|s\|_1 \mathcal{J}_s^\pi} \\ &\leq \frac{\mathcal{J}_s^\pi \mathcal{J}_r^{\pi'}}{\mathcal{J}_s^\pi + \mathcal{J}_r^{\pi'} - \mathcal{J}_s^\pi \mathcal{J}_r^{\pi'}}. \end{aligned}$$

And for case (2)

$$\begin{aligned}
\|s \cup r\|_1 &\geq \|s\|_1 + \|r\|_1 - \|r\|_1 \mathcal{J}_r^{\pi'} \\
&\geq \|r\|_1 \frac{\mathcal{J}_r^{\pi'}}{\mathcal{J}_s^{\pi}} + \|r\|_1 - \|r\|_1 \mathcal{J}_r^{\pi'} \Rightarrow \\
\mathcal{J}(s,r) &= \frac{\|s \cap r\|_1}{\|s \cup r\|_1} \\
&\leq \frac{\|r\|_1 \mathcal{J}_r^{\pi'}}{\|r\|_1 \frac{\mathcal{J}_r^{\pi'}}{\mathcal{J}_s^{\pi}} + \|r\|_1 - \|r\|_1 \mathcal{J}_r^{\pi'}} \\
&\leq \frac{\mathcal{J}_s^{\pi} \mathcal{J}_r^{\pi'}}{\mathcal{J}_s^{\pi} + \mathcal{J}_r^{\pi'} - \mathcal{J}_s^{\pi} \mathcal{J}_r^{\pi'}}.
\end{aligned}$$

We denote the new similarity upper bound with

$$\mathcal{J}' = \frac{\mathcal{J}_s^{\pi} \mathcal{J}_r^{\pi'}}{\mathcal{J}_s^{\pi} + \mathcal{J}_r^{\pi'} - \mathcal{J}_s^{\pi} \mathcal{J}_r^{\pi'}}.$$

The following holds

Lemma 7.7. The function $f(x,y) = \frac{xy}{x+y-xy}$ is monotonically increasing in x and y.

Proof. $f(x,y)$ is monotonically increasing in x and y if and only if the function $g(x,y) = \frac{1}{f(x,y)}$ is monotonically decreasing in x and y. But

$$g(x,y) = \frac{1}{x} + \frac{1}{y} - 1,$$

which is clearly monotonically decreasing in x and y. $\square$

Hence the new upper bound similarity threshold $\mathcal{J}'$ is monotonically increasing in $\mathcal{J}_s^{\pi}$ and $\mathcal{J}_r^{\pi'}$. Since by construction strings are processed in decreasing order of $\mathcal{J}_s^{\pi}$, they are also processed in decreasing order of $\mathcal{J}'$. Given a new string r, ready to be inserted in list $L(\lambda_\pi^s)$, as we are scanning $L(\lambda_\pi^s)$ to identify new candidate pairs with respect to r, we can stop scanning once we encounter the first string s s.t. $\mathcal{J}' < \mathcal{J}_k$. No subsequent string can have a larger similarity upper bound, since, by construction, they have been inserted in $L(\lambda_\pi^s)$ in decreasing order of $\mathcal{J}_s^{\pi}$. Furthermore, since all remaining strings r in the event queue are also processed in decreasing order of $\mathcal{J}_r^{\pi'}$, no string r will ever need to access any string $s \in L(\lambda_\pi^s)$ after the point where the scan has stopped, hence we can delete from $L(\lambda_\pi^s)$ all remaining strings to save space.

A drawback of this algorithm is that it might have to compute the exact similarity between the same pair of strings as many times as the number of common tokens between those strings. An obvious optimization is to insert the similarity of each pair into a hash table the first time it is computed. But this would result in a potentially large hash table. Instead, a simple improvement is to first compute whether a pair will be identified a second time, the first time its similarity is computed, and insert it in the hash table only if necessary. This is possible, since it is easy to compute an upper bound on the similarity of the pair and hence a worst case prefix length that will have to be examined. We simply compute whether the two strings have more than one token in common within this worst case prefix.

7.2 Partitioning and Enumeration Signatures

Prefix filtering is based on defining a total order of tokens in Λ. A fundamentally different approach is based on permuting the tokens instead. After tokens have been randomly permuted, we either partition tokens and hash each partition to create a signature based on the pigeonhole principle, or enumerate all possible combinations of a certain number of tokens, based on the prefix principle, and hash the resulting combinations to create a signature.

7.2.1 The Pigeonhole Signature

Consider a random permutation of tokens in Λ. Such a permutation in practice can be imposed by using a hash function $h{:}\,\Lambda \to \mathbb{N}$ (practical hash functions cannot produce a truly uniform permutation of tokens, but in reality the permutation imposed by these hash functions is good enough for applications).

Partitioning signatures are based on the pigeonhole principle. For simplicity, consider strings as sets, and let a string s be conceptually represented as a vector of dimensionality Λ, with magnitude $W(\lambda_i^s)$

for each coordinate corresponding to a token $\lambda_i^s \in s$ and 0 elsewhere (i.e., the vector space model). The basic partitioning signatures work only for unit token weights, in which case the string vectors contain 1 in each coordinate corresponding to all tokens $\lambda_i^s \in s$ and 0 elsewhere. Extensions for uniform (other than unit) and non-uniform weights will be discussed shortly.

Let the tokens in Λ be ordered using the permutation function h. In other words the token corresponding to the i-th vector coordinate is determined by h. The Hamming distance between two vectors is the number of dimensions (i.e., tokens) in which the two vectors differ, that is

Definition 7.6 (Hamming Distance). Given two $|\Lambda|$-dimensional Boolean vectors $s = \{b_1^s, \ldots, b_{|\Lambda|}^s\}$, $r = \{b_1^r, \ldots, b_{|\Lambda|}^r\}$, the Hamming distance is defined as

$$\mathcal{H}(s,r) = \sum_{i=1}^{|\Lambda|} I(b_i^s = b_i^r),$$

where $I(b_i^s = b_i^r)$ is an indicator variable that returns 1 if $b_i^s = b_i^r$ and 0 otherwise.

The definition can be straightforwardly extended for sequences and frequency-sets.

Assume that we partition the vectors into $\theta + 1$ groups of consecutive dimensions from the permuted token universe $h(\Lambda)$. If two vectors are within Hamming distance θ then by the pigeonhole principle the two vectors must completely agree in at least one partition. Given a string s we can create a simple string signature based on the pigeonhole principle by hashing the bit-vectors corresponding to each one of the $\theta + 1$ partitions using a hash function $h'\colon \{0,1\}^* \Rightarrow [0, 2^b)$ (b dependent on the desired accuracy), and maintaining the resulting $\theta + 1$ hash values. We denote this signature with PG^1. An example is shown in Figure 7.3.

Two strings are within Hamming distance θ if and only if their hash signatures agree in at least one hash value *corresponding to the same partition.* For convenience, we can concatenate the partition number along with the bit-vector corresponding to that partition and hash

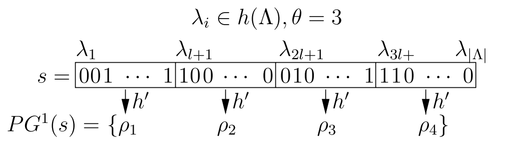


Fig. 7.3 If two vectors differ in all four hash values σ_i, then they differ in at least one bit within each partition, hence their Hamming distance is larger than $\theta = 3$.

the concatenated string, such that the partition number information is encoded in the hash value. In that case, two strings are within Hamming distance θ if and only if they have at least one hash value in common.

Lemma 7.8. Given strings s, r and their corresponding pigeonhole signatures $PG^1(s), PG^1(r)$

$$\mathcal{H}(s,r) \leq \theta \Rightarrow PG^1(s) \cap PG^1(r) \neq \emptyset.$$

Expressing buckets as bit sequences in order to compute the hash value of each bucket is not practical, since it implies the need to actually instantiate the sparse vectors corresponding to data strings (depending on the dimensionality of the vectors and the hash function used, buckets can have extremely large bit sequence representations). A more practical alternative is to hash a compact sparse representation of each bucket. This can be accomplished easily by concatenating and hashing only the indices of those coordinates that are set to 1.

The filtering effectiveness of signature PG^1 is affected by two factors. First, it is not unlikely for two strings to agree in a given partition by pure chance resulting in a false positive. Second, in practice the vector dimensionality $|\Lambda|$ is very large with respect to the average string length. This implies that the resulting vectors are very sparse. Hence, in expectation, a large number of strings will contain empty partitions (i.e., partitions with bit-vectors consisting only of zeros). Pairs of strings that share empty partitions need to be reported as candidates, which might lead to an increased number of false positives. It is tempting to consider pruning all pairs of strings that agree only on empty partitions. Nevertheless, this would lead to false dismissals, as can be easily proven by a counter-example (a match on an empty partition implies that the two strings *do not differ* on that set of coordinates which is essential information with respect to their Hamming distance).

To boost the accuracy of signature PG^1 we can create multiple copies of the signature using an independent family of permutations H. Then, if two vectors are within Hamming distance θ they have to agree in at least one partition in all permutations H. In other words, we boost the accuracy of the signature by repeating the random permutation test multiple times. Of course the tradeoff is that we have to maintain multiple independent signatures per strings.

We can also boost the accuracy of the partitioning signature by using more than $\theta + 1$ buckets. Consider a signature scheme that partitions the vector coordinates into $\theta + x, x > 1$ partitions. Then, by the pigeonhole principle once again if two vectors are within Hamming distance θ they have to agree in at least x partitions. Denote this new partitioning based signature with PG^x.

Lemma 7.9. For simplicity, let the probability that two signatures match at a given coordinate be uniform across all coordinates and equal to p. The probability of a positive answer for signature PG^x, $x \geq 1$ can be expressed as:

$$\sum_{i=x}^{\theta+x} \binom{\theta + x}{i} p^{\frac{|\Lambda|}{\theta+x} i}(1 - p^{\frac{|\Lambda|}{\theta+x}})^{\theta+x-i}.$$

A simple analysis shows that for sufficiently large values of $|\Lambda|, x$ the expected filtering effectiveness of PG^x is better than that of PG^1. This result is not surprising, considering that signature PG^x requires more space than signature PG^1 for increasing x (recall that we maintain one hash value per partition). For example, when $\theta + x = |\Lambda|$, PG^x reduces to maintaining the actual vectors, which leads to no false positive answers. Alternatively, we can also boost the accuracy of signature PG^x by using multiple permutations for an independent family of permutations H, at the cost of an increased signature size.

So far we have talked about various signatures based on random permutations and hashing for computing a lower bound on the Hamming distance between vectors. The Hamming distance can be used to derive upper bounds on the Intersection, Jaccard, Dice, and Cosine similarity for any uniform weighing function W.

Lemma 7.10. Assuming a uniform weight function $W(\lambda) = c, \lambda \in \Lambda$, given two strings s, r

$$\mathcal{I}(s,r) \geq \theta \Rightarrow \mathcal{H}(s,r) \leq |s| + |r| - 2\frac{\theta}{c}.$$

Proof.

$$\mathcal{I}(s,r) \geq \theta \Rightarrow c|s \cap r| \geq \theta,$$

and

$$\mathcal{H}(s,r) = |s| - |s \cap r| + |r| - |s \cap r| = |s| + |r| - 2|s \cap r| \Rightarrow$$

$$|s \cap r| = \frac{|s| + |r| - \mathcal{H}(s,r)}{2}.$$

Hence,

$$|s| + |r| - \mathcal{H}(s,r) \geq 2\frac{\theta}{c} \Rightarrow \mathcal{H}(s,r) \leq |s| + |r| - 2\frac{\theta}{c}. \qquad \square$$

Similarly

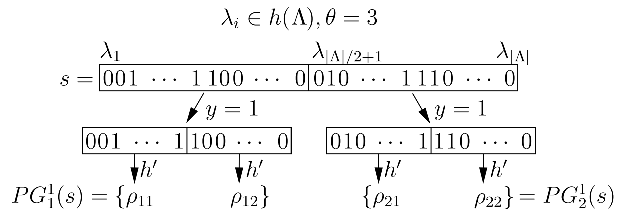
Lemma 7.11. Assuming a uniform weight function $W(\lambda) = c$, $\lambda \in \Lambda$, given two strings s, r

- Normalized Weighted Intersection:
$$\mathcal{N}(s,r) \geq \theta \Rightarrow \mathcal{H}(s,r) \leq |s| + |r| - 2\theta \max(|s|, |r|).$$
- Jaccard:
$$\mathcal{J}(s,r) \geq \theta \Rightarrow \mathcal{H}(s,r) \leq \frac{1-\theta}{1+\theta}(|s| + |r|).$$
- Dice:
$$\mathcal{D}(s,r) \geq \theta \Rightarrow \mathcal{H}(s,r) \leq (1-\theta)(|s| + |r|).$$
- Cosine:
$$\mathcal{C}(s,r) \geq \theta \Rightarrow \mathcal{H}(s,r) \leq |s| + |r| - 2\theta\sqrt{|s||r|}.$$

7.2.2 The Partenum Signature

Partenum is another popular string signature that has been proposed in the literature. The partenum signature uses a fine-grained partitioning and enumeration strategy, more complex than that of signature PG^x. The partenum signature is constructed by first partitioning the vectors into $\frac{\theta+1}{2}$ partitions of consecutive dimensions from the permuted universe $h(\Lambda)$. A simple counting argument proves that given two vectors with Hamming distance smaller or equal to θ, then at least one of the $\frac{\theta+1}{2}$ partitions has Hamming distance smaller or equal to one. Hence, we can construct one signature per partition with a new, smaller Hamming threshold $\theta' = 1$, and use the union of all partition signatures as the signature of the string. For each partition we use PG^y as the signature of the partition, by constructing $1 + y$ sub-partitions. The partenum signature consists of the union of the $\frac{\theta+1}{2}$ PG^y signatures. An example is shown in Figure 7.4. Given a pair of vectors, if all partitions are within Hamming distance greater than one, then the pair of vectors cannot be within Hamming distance smaller equal to θ. Hence, a pair of vectors is reported as a candidate if and only if at least one of the PG^y signatures corresponding to the same partition matches (meaning that at least one of the $\frac{\theta+1}{2}$ partitions has Hamming distance smaller or equal to one).

The probability of a positive answer for partenum can be expressed similarly to that of Lemma 7.9.



$\lambda_i \in h(\Lambda), \theta = 3$
λ_1
$\lambda_{|\Lambda|/2+1}$
$\lambda_{|\Lambda|}$
$s =$ 001 ··· 1 100 ··· 0 010 ··· 1 110 ··· 0
$y = 1$
$y = 1$
001 ··· 1 100 ··· 0
010 ··· 1 110 ··· 0
h'
h'
h'
h'
$PG^1_1(s) = \{\rho_{11}$ $\rho_{12}\}$
$\{\rho_{21}$ $\rho_{22}\} = PG^1_2(s)$

Fig. 7.4 First, the vector is split into $\frac{\theta+1}{2} = 2$ partitions. A pair of vectors is within Hamming distance three iff the vectors have Hamming distance smaller equal to one in at least one of the two partitions. Then, for each partition we create a PG^1 signature. We can use the PG^1 signatures to efficiently verify whether each partition between a pair of vectors is within Hamming distance one. If at least one such partition exists, it is possible that the pair of vectors is within Hamming distance three, and we report the pair as a candidate.

7.2.3 The Prefix Enumeration Signature

Extending the pigeonhole and partenum signatures for arbitrary weights is non-trivial. A naive approach would be to represent each token λ_i^s of string s using $W(\lambda_i^s)$ copies (appropriate rounding techniques can be used for real valued weights), and essentially converting a weighted set of tokens into an unweighted bag and adversely increasing the vector dimensionality. The pigeonhole signature cannot be used for arbitrary weights in practice.

A different approach is to use ideas both from the pigeonhole and prefix signatures to produce a weighted signature for the string. Assuming Weighted Intersection similarity and query threshold θ, the idea is to enumerate all minimal subsets of tokens of string s whose total weight adds up to θ, and then hash each one of these subsets into an integer using a hash function h (a minimal subset is a set that has no proper subset with total weight larger than θ). The signature of the string is the set of resulting hash values. Clearly, if $\mathcal{I}(s,r) \geq \theta$ then s and r have to have at least one of the enumerated subsets of tokens in common, resulting in a non-empty signature intersection.

The obvious drawback of this signature scheme is that for very long strings the number of minimal token subsets exceeding the query threshold will tend to be very large (especially for small thresholds), resulting in a very large signature. To decrease the size of the signature we can order every subset of tokens in decreasing weight order and hash only a prefix of each subset (e.g., we hash all minimal prefixes with total weight $\geq \theta'$, where θ' controls the size of the signature). It is expected that a large number of minimal subsets will share the same minimal prefix resulting in exactly the same hash value, and hence reducing the size of the signature. The drawback of course is that the new signature results in false positives. Also, the resulting signatures do not have a fixed size; the size heavily depends on the contents of a string. We call this signature the prefix enumeration signature.

7.2.4 All-Match Selection Queries

For evaluating Hamming distance, we can use an inverted index to answer queries using either the pigeonhole or the prefix enumeration signatures. Recall that given two strings s, r and corresponding pigeonhole signatures $PG^x(s), PG^x(r)$, $\mathcal{H}(s,r) \leq \theta$ implies that $|PG^x(s) \cap PG^x(r)| \geq x$. Answering this query using an inverted index is reminiscent of using the `multiway merge`, `threshold` and `heaviest first` algorithms to answer intersection queries on strings (with unit token weights). Indeed, we can use essentially the same algorithms to evaluate the intersection between pigeonhole signatures using the inverted index. The same ideas apply for answering queries using the prefix enumeration signature. The resulting inverted index contains one list per hash value, for all hash values contained in the signatures of the data strings.

Nevertheless, we cannot straightforwardly use any of these signatures to answer all-match selection queries for any of the other similarity measures, given that the derived upper bounds given in Lemma 7.11, depend on the lengths of both the data and the query string. Since for arbitrary queries the lengths of the query strings are not known in advance, constructing a meaningful signature for each data string a priori is not trivial. In order to build such signatures, we use the length filters (i.e., L_p-norm filters for non-uniform weights) derived for each similarity function.

Simply stated, given a particular similarity function (other than Hamming), a similarity threshold θ, and the associated L_p-norm filter, we know that for a given data string, the only query strings that can have similarity larger than θ have to fall within the norm interval specified by the norm filter. Hence, when constructing the signature of the data string, we can simply derive an upper bound on the target Hamming distance that will guarantee no false dismissals by using the minimum and maximum filtering lengths. Clearly, since Weighted Intersection is not based on string norms, we cannot use these signatures to evaluate Weighted Intersection similarity.

In particular, the following upper bounds are easily derived.

Lemma 7.12. Assuming a uniform weight function $W(\lambda) = c, \lambda \in \Lambda$, given two strings s, r

- Normalized Weighted Intersection:

$$\mathcal{N}(s,r) \geq \theta \Rightarrow \mathcal{H}(s,r) \leq \left(\frac{1}{\theta} + 1 - 2\theta\right)|r|.$$

- Jaccard:

$$\mathcal{J}(s,r) \geq \theta \Rightarrow \mathcal{H}(s,r) \leq \left(\frac{1}{\theta} - 1\right)|r|.$$

- Dice:

$$\mathcal{D}(s,r) \geq \theta \Rightarrow \mathcal{H}(s,r) \leq \left(\frac{2}{\theta} - 2\right)|r|.$$

- Cosine:

$$\mathcal{C}(s,r) \geq \theta \Rightarrow \mathcal{H}(s,r) \leq \left(\frac{1}{\theta} + 1 - 2\theta\right)|r|.$$

Notice that the pigeonhole signature and the prefix enumeration signature have to be built with respect to a pre-determined minimum query threshold θ. Queries with threshold $\theta' < \theta$ cannot be answered correctly using the existing signatures. Queries with threshold $\theta' \geq \theta$ can be answered without any false negatives.

The original partenum signature proposed in the literature uses an enumeration strategy in order to reduce the problem of identifying sub-signatures with intersection size larger than or equal to x, to that of identifying sub-signatures with non-empty intersections (similar to the prefix enumeration approach). The idea is simple. Given sub-signatures consisting of $1 + x$ partitions, enumerate all possible $\binom{1+x}{x} = 1 + x$ combinations of x partitions and hash the resulting combinations instead of each of the original $1 + x$ partitions individually. Then, if two sub-signatures agree in at least x partitions they must agree in at least one hash value. The same idea applies for the pigeonhole signature only for $\theta = 1$, otherwise the number of combinations explodes, yielding very large signatures. This is hardly a problem though, since answering intersection queries using the algorithms described in Section 6 is equally efficient for small and large intersection thresholds alike.

7.2.5 Top-k Selection Queries

Partitioning and enumeration signatures cannot be used to answer top-k selection queries due to the fact that a minimum pre-determined threshold θ needs to be decided in advance in order to build the signatures. Since the k-th Hamming distance between any data string and a given query in the worst case could be $|\Lambda|$, partitioning and enumeration signatures would degenerate to keeping the entire vectors.

7.2.6 All-Match Join and Self-join Queries

All signatures can be used to answer join and self-join queries, with algorithms similar to those discussed for the prefix filter in Sections 6.3 and 6.4, by replacing the full inverted index with a signature based inverted index. Once again, signatures for similarity functions other than Hamming have to be constructed using a best case scenario, by utilizing the minimum and maximum possible query string lengths defined by the corresponding length filters, as discussed in Section 7.2.4.

7.2.7 Top-*k* Join and Self-join Queries

Once again top-k join and self-join queries can be answered by using the all-match join and self-join algorithms with signature based inverted indexes instead of full inverted indexes. The algorithms simply identify any k candidates during initialization, and use the k-th largest Hamming distance as the initial threshold θ. Notice that in the worst case the initial k-th distance can be $|\Lambda|$, and the signature based inverted index will degenerate to a full inverted index.

7.3 The Filter Tree

When building a full inverted index on the data we can apply various filtering techniques to reduce the number of candidate strings examined. That is, we are able to apply various filtering criteria based on the specific properties of each similarity function (e.g., norm filtering) to prune strings while scanning a particular inverted list.

We can generalize this idea by applying the various filtering algorithms, using a filter tree. A filter tree is a combination of trees. For example, in the first level of the tree we partition strings according to their L_p-norms. In the second level of the tree we could partition strings according to the tokens contained in their *idf* sorted prefixes. In the third level of the filter tree we could use a positional filter for every prefix token. In other words, at the first level we use an interval tree to partition strings according to norms. Then, for every leaf node of the interval tree we create a suffix tree that indexes only the prefix tokens of all strings within the L_p-norm range of that particular interval tree leaf node. Finally, at the leaf nodes of the suffix trees we create a forest of B-trees that partitions strings according to the position of the prefix token corresponding to that particular leaf node. At the last level of the filter tree we build a forest of inverted indexes containing all the qualifying strings, as they are filtered out level by level.

For example, a certain path down the tree might correspond to all strings with L_p-norm equal to 100, that contain q-gram 'abc' in their *idf* sorted prefix, at absolute position 10 within the string. A query might have to access several paths down the tree simultaneously (e.g., it might have to follow multiple L_p-norm intervals). In the end, we merge the lists corresponding to each candidate path and finally take the union of resulting strings.

The filter tree can be implemented straightforwardly by constructing one inverted list per token, and then sorting the list in the order of the filters specified by the filter tree. In the example above, each inverted list would be sorted, first by the norms of the strings contained in the list, and then by the position of that particular token in each string. Given a query string, one can identify all candidate strings by using binary search to locate on every token list the appropriate norm interval (the norm filter); and subsequently, for every norm the appropriate positions interval (the positional filter). By examining only those token lists that correspond to the tokens contained in the *idf* prefix of the query (the prefix filter), the resulting strings are the strings that would be visited by the actual filter tree.

The order in which the filters are applied significantly affects the performance of the filter tree. Ordering depends on query and data characteristics, as well as on the filtering techniques used. Intuitively, simpler, easy to evaluate filters should be used as close to the root of the filter tree as possible. Also, filters that result in non-overlapping partitions of strings should be evaluated first, since these filters reduce the total amount of candidate paths produced. For example the L_p-norm filter results in a non-overlapping partitioning of strings since every string corresponds to exactly one L_p-norm value; on the other hand the prefix filter results in an overlapping partitioning of strings, since a single string will end up in the partition of more than one prefix tokens.

Notice that using the filter tree does not affect what types of algorithms are used to merge lists at the leaf level of the filter tree. In particular, specific optimizations can still be applied at that point to speed-up list merging. For example, one can still use the `heaviest first` algorithm to merge lists by applying the incremental L_p-norm tightening principle (see Lemma 6.15), even though an initial L_p-norm filter has already been applied at the top of the filter tree.

7.4 Index Updates

Similar updating issues with the ones discussed for full token based inverted indexes hold for signature based inverted indexes. In order to be able to perform efficient updates we need to represent token lists either as binary trees or linked lists in main memory, or as B-trees in external memory. Some signature techniques suffer from problems similar to those for inverted indexes when token weights get updated. For example, for prefix signatures a single token weight update can change the sort order of tokens, and as a consequence, affect the prefix signature of every other string in the index. In that case, the complete index needs to be rebuilt to guarantee correctness. Delayed propagation of updates in this case will inevitably lead to false dismissals. The same problem occurs with the prefix enumeration signature. The pigeonhole signature does not depend on token weights and hence can support updates efficiently.

7.5 Discussion and Related Issues

Filtering techniques are the preferred algorithm for evaluating join queries, due to the small size of the resulting inverted index. The prefix filter is a very general technique that works independently of the weighing scheme used. The pigeonhole signature is limited to uniform weights. The prefix enumeration signature can be used for arbitrary weights, it has larger size on average than the prefix signature, but it results in tighter pruning. The filter tree has proved to be very effective for answering selection queries, but its performance depends heavily on the order in which filters are composed. It is difficult to compare filtering techniques with algorithms based on full inverted indexes. The relative performance depends on whether the indexes can fit in main memory and also on the query and data characteristics. There exists an inherent tradeoff between using the index to compute the exact similarity between the query and data strings, over using the index as a filtering step that requires a subsequent verification step.

7.6 Related Work

A first instantiation of the prefix principle was proposed by Sarawagi and Kirpal [60]. The prefix filter was later formalized by Chaudhuri et al. [19]. Xiao et al. [75] proposed improved pruning conditions for Jaccard, Dice, and Cosine similarity for the prefix filter, assuming unit token weights. The extensions to arbitrary weights discussed here are straightforward. Chang and Lawler [17] used the idea of partitioning the strings using the pigeonhole principle for computing both Hamming distance and edit distance. The same partitioning strategy was used by Wu and Manber [72] for the agrep utility. The prefix enumeration and partenum signatures were proposed by Arasu et al. [6].

Bayardo et al. [10] proposed various join algorithms and optimizations related to building and indexing prefix signatures, and using advanced list merging algorithms to evaluate join queries, similar to the ones described in Section 6.3. Xiao et al. [75] extended the algorithms by Bayardo et al. to access parts of the suffix of strings as well in order to improve filtering efficiency, assuming that datasets fit in main memory. Top-k join queries were extensively studied by Xiao et al. [74], assuming that the dataset fits in main memory. In addition, Xiao et al. [74] discussed various optimizations specifically tailored for unit weights. The filter tree was proposed by Li et al. [48].

8

Algorithms for Edit Based Similarity

In this section we will cover evaluation of edit based similarity. We will show that the techniques introduced for set based similarity functions can be applied to answer unweighted edit distance queries. We will also present data structures based on tries and B-trees that can be used to answer unweighted, weighted, and normalized edit distance.

8.1 Inverted Indexes

From the outset it does not appear that edit based similarity is related to set based similarity, and by extension, to token based inverted indexes. Although, careful observation reveals that this is not true for the case of simple, unweighted edit distance (i.e., where all edit operations have unit cost).

Lemma 8.1 (*q*-gram Intersection Filter for Edit Distance). Let Λ be the universe of q-grams. Given sequences $s = \lambda_1^s \cdots \lambda_m^s$ and $r = \lambda_1^r \cdots \lambda_n^r$, $\lambda_i^s, \lambda_j^r \in \Lambda$, and edit distance threshold θ, then

$$\mathcal{E}(s,r) \leq \theta \Rightarrow \mathcal{I}(s,r) \geq \max(m,n) - q + 1 - q\theta.$$

Proof. To prove the claim, notice that s consists of $|s| - q + 1$ q-grams and r of $|r| - q + 1$ q-grams. The maximum number of q-grams is $x = \max(|s|, |r|) - q + 1$. Each edit operation can affect at most q q-grams, thus θ edit operations can affect at most $q\theta$ q-grams. Hence, if $\mathcal{E}(s,r) \leq \theta$, then with θ edit operations we can obtain s from r (or r from s) if and only if s and r have at least $x - q\theta$ q-grams in common. $\square$

The intuition is that strings within a small edit distance must have a large number of q-grams in common. Imagine that string s can be obtained from string r with a replacement of a single character. Then, intuitively, in the worst case s and r differ in at most q q-grams, since one edit operation can affect at most q q-grams.

By using Lemma 8.1 an edit distance constraint is converted into a simpler Intersection similarity constraint on q-grams. Thus, all algorithms described in Section 6, suitable for computing Weighted Intersection similarity with unit token weights, can now be applied to evaluate edit distance (notice that the actual token weighing function W is irrelevant when evaluating edit distance).

Given that strings with lengths shorter than q result in empty q-gram sets, as already discussed, for implementation purposes we can extend all strings with $q - 1$ copies of a special beginning and ending character $\# \notin \Sigma$ to avoid special cases. The intersection constraint of Lemma 8.1 now becomes

$$\mathcal{E}(s,r) \leq \theta \Rightarrow \mathcal{I}(s,r) \geq \max(|s|, |r|) + q - 1 - q\theta,$$

given that the total number of q-grams per string increases to $|s| + 2(q - 1) - q + 1$.

For brevity, in the rest we refer to the q-gram intersection threshold as

$$\tau = \max(|s|, |r|) - q + 1 - q\theta$$

(or $\tau = \max(|s|, |r|) + q - 1 - q\theta$ depending on whether extended q-gram sets are used).

Notice that Lemma 8.1 results in a very loose intersection constraint that does not take into account the positions of matching q-grams within the strings. Thus, a variety of optimizations, based on q-gram positions, can be applied over the existing inverted index based algorithms. It holds that

---

Lemma 8.2. Given strings $s = \{(\lambda_1^s, p_1^s), \ldots, (\lambda_m^s, p_m^s)\}$ and $r = \{(\lambda_1^r, p_1^r), \ldots, (\lambda_n^r, p_n^r)\}$, let the matching set of positional q-grams be $s \cap r = \bigcup\{(\lambda_i^s, p_i^s), (\lambda_j^r, p_j^r)\}$. Then

$$\mathcal{E}(s,r) \leq \theta \Rightarrow \forall \lambda_i^s, \lambda_j^r \in s \cap r : |p_i^s - p_j^r| \leq \theta.$$

Proof. Assuming that $\mathcal{E}(s,r) \leq \theta$ and that a q-gram $\lambda_i^s \in s$ matches q-gram $\lambda_j^r \in r$ with $|p_i^s - p_j^r| > \theta$, clearly we need at least $\theta + 1$ edit operations to transform the position of λ_i^s into that of λ_j^r, without affecting any other q-grams, to obtain one q-gram set from the other. This is a contradiction. $\square$

The claim states that if two strings are within edit distance θ, then the positions of the matching q-grams between the two strings cannot differ by more than θ.

We can use Lemma 8.2 when scanning inverted lists to prune candidates based on positional information of q-grams, by comparing against the positions of these q-grams in the query string. Positional information can be used for all algorithms discussed in Section 6. In the algorithms, while scanning a list corresponding to a query q-gram with position p, all strings whose positions for that q-gram differ by more than θ from p can be ignored. A string is considered a viable candidate if and only if the resulting q-gram intersection with the query exceeds threshold τ.

Finally, we can state a simple length filter for edit distance, which can be used in all algorithms for pruning.

Lemma 8.3 (Edit Distance L_0-norm Filter). Given strings s, r and edit distance threshold θ

$$\mathcal{E}(s,r) \leq \theta \Rightarrow ||s| - |r|| \leq \theta.$$

When using the list merging algorithms proposed in Section 6 for set based similarity, the resulting set of strings forms the exact answer to the query. In contrast, for edit based similarity, the resulting set of strings is a super set of the exact answer, since the various filtering techniques are used to prune candidate strings, but not to compute the actual edit distance between the query and each string (that would only be possible if the algorithm maintained all the discovered q-gram information per string, which would make the book-keeping cost prohibitive). This implies that a verification step is necessary in order to filter out false positives. The verification step is performed by verifying that the edit distance between each candidate string and the query is indeed smaller than or equal to the query threshold. For that purpose we can use the fast edit distance verification algorithm presented in Section 2.1. This is in contrast to the same algorithms used for set based similarity measures, where the actual similarity is computed during index traversal, and no subsequent verification step is needed.

One pitfall of this approach is that for a large enough threshold θ, short query lengths and an inappropriate q-gram length, the q-gram intersection constraint might return a non-positive value. In other words, potential answers might not have any q-grams in common with the query. Given that the list merging algorithms examine only those inverted lists that correspond to query q-grams, answers that do not share any q-grams with the query will never be discovered. In this case, only a linear scan of the inverted index would be able to identify all answers. For example, assume that $\theta = 1, q = 2$, and $v = abc$. Then, according to Lemma 8.1, potential answers to the query need to have at least $3 - 2 + 1 - 2 = 0$ q-grams in common. String $s = adc$ is a valid answer for this query, yet it does not have any q-grams in common with v. Similar counter-examples exist for the case of extended q-gram sets.

Finally, it should be noted here that, as a consequence of using q-grams to evaluate the edit distance, these techniques cannot be extended for evaluating either weighted (i.e., where each edit operation has a specific weight) or normalized edit distance (i.e., where the distance is normalized by the length of the strings).

8.2 Filtering Techniques

8.2.1 The Prefix, Prefix Enumeration, and Pigeonhole Signatures

Based on Lemma 8.1 and the fact that we are dealing with unit token weights, we can use all filtering techniques discussed in Section 7 without modifications. We simply build string signatures for Intersection similarity. Given that we are dealing with unit weights, we sort tokens using *idf* weights. Recall that this ordering has the advantage that prefixes contain the least frequent tokens, which decreases the probability of collisions that lead to false positives. Given query string v and an arbitrary data string s, Lemma 8.1 states that the intersection of v and s need to be at least $\tau = \max(|v|,|s|) - q + 1 - q\theta$. According to Definition 7.1, given that we have a maximum of $\max(|v|,|s|) - q + 1$ tokens per string, we need at most $\tau - 1$ tokens in the string suffix to guarantee no false negatives. Hence, the prefix signature has size equal to

$$[\max(|v|,|s|) - q + 1] - (\tau - 1) = q\theta + 1.$$

Thus, the prefixes of all strings have a fixed size of $q\theta + 1$ q-grams, which is independent of the query or data string length, and depends only on edit distance threshold θ.

An additional optimization of the prefix filter for top-k join and self-join queries is also possible, based on the fact that we are dealing with unit token weights, as discussed in Section 7.1.5.

There is no known way of extending filtering algorithms for evaluating weighted and normalized edit distance.

8.2.2 The Mismatch Filter

The mismatch filter is an extension of the prefix filter that takes into account the positional information and the content of mismatching q-grams between two prefix signatures, to perform tighter pruning.

The q-gram Intersection Filter for Edit Distance (see Lemma 8.1) is based on the fact that since one edit operation can affect at most q

q-grams we can count the minimum number of *common* q-grams that two strings need to have to be within θ edit operations.

An inverse argument is based on the number of q-grams that the strings *do not have in common* (i.e., the mismatching q-grams). The question now becomes what is the minimum number of edit operations needed to cause the observed mismatching q-grams, which is equal to the number of edit operations needed to fix these q-grams. This number is at least as large as the number of edit operations needed to simply *destroy* all mismatching q-grams. For example, consider the two strings $s =$ 'One Laptop' and $r =$ '1 Laptop'. The positional 2-gram sets are

1	2	3	4	5	6	7	8	9
'On'	'ne'	'e '	' L'	'La'	'ap'	'pt'	'to'	'op'
'1 '	' L'	'La'	'ap'	'pt'	'to'	'op'		

Assuming an edit distance threshold $\theta = 3$, the mismatching positional 2-grams belonging to s are ('On', 1), ('ne', 2), ('e', 3) (based on Lemma 8.2). Notice that a replacement of 'n' with '1' will affect two 2-grams at the same time. Hence, only two edit operations are needed in this case to destroy all three 2-grams, one for 2-grams ('On', 1), ('ne', 2) and one for 2-gram ('e ', 3).

Reasoning about mismatching q-grams leads to the following statement:

Lemma 8.4 (Mismatch Filtering). Consider strings s, r and the set of positional mismatching q-grams $Q_{s\to r} = \{(\lambda_1^s, p_1^s), \ldots, (\lambda_\ell^s, p_\ell^s)\}, \lambda_i^s \in s$. Let ϕ be the minimum number of edit operations needed to destroy all q-grams in $Q_{s\to r}$. Then, $\mathcal{E}(s,r) \geq \phi$.

Proof. If we need exactly ϕ operations to simply destroy all mismatching q-grams, we need at least ϕ operations to correct the same q-grams, in the best case scenario. □

We call the minimum number of edit operations needed to destroy a set of q-grams Q, *the minimum destroy operations* for Q. We can improve the minimum destroy operations bound by first computing the number of operations needed to destroy the q-grams in $Q_{s\to r}$ and then in $Q_{r\to s}$. Then, we take the larger of the two to be the minimum destroy operations of the mismatching q-grams of pair s,r.

An optimal greedy algorithm for computing the minimum destroy operations with cost linear in the number of q-grams is shown as Algorithm 8.2.1. The algorithm selects the next unprocessed q-gram and makes a conceptual replacement of its last character, in essence destroying the current q-gram and all subsequent q-grams incident on this character. The algorithm iterates until no more q-grams are left to destroy.

Algorithm 8.2.1: MINIMUM DESTROY OPERATIONS(Q)

```
Let $Q = \{(\lambda_1,p_1),\dots,(\lambda_\ell,p_\ell)\}$ s.t. $p_i < p_j, \forall i < j$
$e \leftarrow 0$,  $p \leftarrow 0$
for all $(\lambda_i,p_i) \in Q$
  do if $p_i > p$
       then $e \leftarrow e + 1$
            $p \leftarrow p_i + q - 1$
return $e$
```

Given query string v, the prefix filter principle dictates that a string s is a viable candidate if and only if the prefixes of v and s have a non-empty intersection. Given that the prefixes have a fixed size of $q\theta + 1$ q-grams, in the worst case the prefix of a candidate can have as many as $q\theta$ mismatching q-grams. This yields ample opportunity for the mismatch filter to conclude that there are enough mismatching q-grams within the prefixes to prune the candidate.

In fact, we can strengthen the mismatch filter pruning condition even further by computing the minimum number of mismatching q-grams within the prefix of a string that will guarantee an edit distance larger than θ. The length of the prefix containing these q-grams can potentially be smaller than $q\theta + 1$, depending on how much overlap the q-grams in the shorter prefix have. We call the shorter prefix the *mismatch prefix.* If two strings have an empty mismatch prefix intersection, then they contain enough mismatching q-grams for the edit distance to be larger than θ and the pair can be pruned using the shorter prefix.

We can identify the mismatch prefix by iteratively computing the minimum destroy operations of all prefixes with lengths in the range $[\theta + 1, q\theta + 1]$. We stop once the minimum destroy operations of the current prefix is larger than θ. The cost of this algorithm is quadratic in $q\theta + 1$. To reduce this cost we can identify the appropriate prefix using binary search, based on the monotonicity of the minimum destroy operations.

Lemma 8.5. Given a set of q-grams Q and any subset $Q' \subset Q$

$$\text{Minimum Destroy Operations}(Q') \leq \text{Minimum Destroy Operations}(Q).$$

The algorithm is shown as Algorithm 8.2.2.

Algorithm 8.2.2: MISMATCH PREFIX(s)

```
Tokenize string: $s = \{(\lambda_1^s, 1), \ldots, (\lambda_m^s, m)\}$ s.t. $idf(\lambda_i^s) \geq idf(\lambda_j^s), i < j$
$a \leftarrow \theta + 1, \quad c \leftarrow q\theta + 1$
while $a < c$
  do $b \leftarrow (a + c)/2$
     $e \leftarrow$ Minimum Destroy Operations($\{(\lambda_1^s, p_1), \ldots, (\lambda_b^s, p_b)\}$)
     if $e \leq \theta$
       then $a \leftarrow b + 1$
       else $c \leftarrow b$
return $a$
```

8.3 Trees

8.3.1 Tries and Suffix Trees

The simplest way of using a trie to find strings within small edit distance θ from the query is to generate all possible strings within edit distance θ from the query and traverse the trie using all resulting queries. Inversely, one can generate all possible strings within edit distance θ from the data strings and pre-construct a trie on the augmented dataset. Clearly, the first approach has very high query cost and the second a very high space cost.

A better approach is to compute an active set of trie nodes incrementally, by considering all prefixes of the query, starting from the empty string $\varnothing$. An active node is one that corresponds to a trie string that is within edit distance θ from the current prefix of the query string. Consider the trie in Figure 8.1 and query string $v =$ 'Found' with $\theta = 1$. For simplicity we denote each node in the tree using the full string corresponding to that node; in practice only a node identifier is used to conserve space. The active nodes for each prefix of v, along with their edit distance from that prefix, are shown in Table 8.1.

Let N be a node in the trie. Denote with $\sigma(N)$ the label of the node, and with $s(N)$ the full string corresponding to that node (i.e., the sequence of labels in the path from the root of the trie to node N). Denote with $v^i = \sigma_1 \cdots \sigma_i, 0 \le i \le |v|$, the i length prefix of query v (let v^0 denote the empty string), and let A_i denote the set of active nodes in the trie for query prefix v^i. Let every entry of A_i be a pair $(N, \mathcal{E}(s(N), v^i))$. We initialize A_0 for prefix $v^0 = \varnothing$ by adding the root node of the trie. The root corresponds to the empty string $\varnothing$ as well

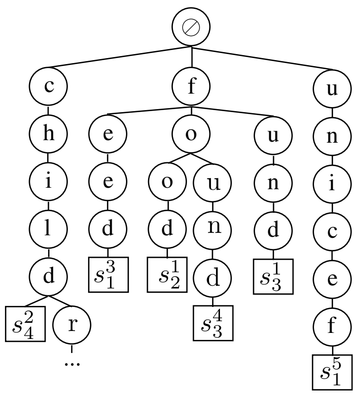


Fig. 8.1 A simple trie stricture.

Table 8.1. Active nodes for each prefix of query "Found" on the trie of Figure 5.1.

$\varnothing$	F	Fo	Fou	Foun	Found
U, 1	U, 1	Fu, 1	Fu, 1	Fun, 1	Fund, 1
F, 1	F, 0	Fo, 0	Fo, 1		
C, 1	C, 1	Foo, 1	Foo, 1		
	Fu, 1	Fe, 1			
	Fo, 1				
	Fe, 1				

(hence active node $\varnothing$ is associated with edit distance $\mathcal{E}(\varnothing,\varnothing) = 0$). We complete set A_0 by adding all nodes corresponding to strings with lengths in the interval $[1,\theta]$. These strings correspond to performing from 1 up to θ arbitrary insertions to the empty string, and are still within edit distance θ from v^0.

The key observation now is that active set A_i can be computed directly from set A_{i-1}. The recurrence relation can be derived from the recurrence relation of the dynamic programming algorithm for edit distance verification (Algorithm 2.1.1). The intuition is that we are trying to incrementally compute the dynamic programming matrix of Algorithm 2.1.1 *for all strings* contained in the trie at once (by virtue of computing the edit distance between the query string and all active paths of the trie).

Given query prefix $v^i, 1 \leq i \leq |v|$, we consider all strings with lengths within the interval $[i-\theta, i+\theta]$ (i.e., the L_0-norm filter bounds for v^i). A_0 corresponds to the first row of the dynamic programming matrix. Every time we increase i we need to compute a new row in the matrix. The new row is computed based on the values in the previous row. Assume that we are computing A_i from A_{i-1} for prefix v^i. The algorithm proceeds by examining every entry in set A_{i-1} and taking three actions:

(1) It determines whether an active node N should be added in set A_i by increasing its edit distance by one. This corresponds to a deletion of character σ_i (and value d_1 in Algorithm 2.1.1).
(2) It tests whether there exists a child node N' of N, s.t. $\sigma(N') = \sigma_i$. In that case N' is added in A_i with edit

distance equal to $\mathcal{E}(s(N'), v^i) = \mathcal{E}(s(N), v^{i-1})$. This action corresponds to a diagonal move down the matrix for a matching character (and value d_3 in Algorithm 2.1.1). An additional sub-action here is that the algorithm needs to consider all new strings contained in the trie that have lengths in the interval $[|s(N')| + 1, |s(N')| + (\theta - \mathcal{E}(s(N'), v^i))]$. This action corresponds to inserting at the end of string $s(N')$, $\theta - \mathcal{E}(s(N'), v^i)$ arbitrary characters — all these strings are still within edit distance at most θ from v^i, given that $\mathcal{E}(s(N'), v^i) \leq \theta$.

(3) It considers all child nodes N' of N, s.t. $\sigma(N) \neq \sigma_i$. These nodes are added in the active set A_i if and only if $\mathcal{E}(s(N), v^{i-1}) < \theta$. This action corresponds to a diagonal move down the matrix for a non-matching character (and value d_3 in Algorithm 2.1.1). In this case, the algorithm does not have to consider strings with lengths in the interval $[|s(N')| + 1, |s(N')| + (\theta - \mathcal{E}(s(N'), v^i))]$. The reason is that all these nodes have been added already by a *matching* parent node of N', through step 2 above, if necessary.

After all active nodes in A_{i-1} have been processed the algorithm discards duplicate nodes in A_i by preserving only the copy with the minimum edit distance from v^i (this corresponds to the minimum operation in Algorithm 2.1.1). This step is necessary since an active node can be processed multiple times as a child of another active node. For that purpose, we can organize the sets of active nodes as hash tables. Initially, table A_i is empty. As we process nodes in table A_{i-1} we probe set A_i, and if the node exists, we update its edit distance if needed, otherwise we insert the node. Notice that the order in which we process nodes from table A_{i-1} is not important. We simply iterate through all entries in the hash table in hash value order. The full algorithm appears as Algorithm 8.3.1.

Notice that Algorithm 8.3.1 can be used to compute all answers to a query string incrementally, as characters are typed by the user one by one. Every time the user types a new character, we use the already computed set $A_{|v|}$ to compute a new set $A_{|v|+1}$, by continuing from where the algorithm left off.

Algorithm 8.3.1: EDIT DISTANCE ON TRIE(v,θ)

```
Let $v = \sigma_1 \cdots \sigma_{|v|}$
$A_i$ is a hash table of pairs $(N, e)$ with key $N$
$A_0 \leftarrow (\varnothing, 0)$
for $i = 1$ to $\theta$
  do {for each node $N$ at distance $i$ from root: $A_0 \leftarrow (N, i)$
for $i = 1$ to $|v|$
  for all $(N, e) \in A_{i-1}$
    if $e + 1 \leq \theta$   // Case 1
      if $(N, e') \in A_i$ :  $A_i \leftarrow (N, \min(e + 1, e'))$
        else $A_i \leftarrow (N, e + 1)$
    for all children $N'$ of $N$
      if $\sigma(N') = \sigma_i$   // Case 2
          do  if $(N', e') \in A_i$ :  $A_i \leftarrow (N', \min(e, e'))$
                else $A_i \leftarrow (N', e)$
              for $j = 1$ to $\theta - e$
                do  for all descendants $N''$ of $N'$, $j$ characters
                      away
                    do {$A_i \leftarrow (N'', \theta - e + j)$
          else if $e + 1 \leq \theta$   // Case 3
          do  if $(N', e') \in A_i$ :  $A_i \leftarrow (N', \min(e + 1, e'))$
                else $A_i \leftarrow (N', e + 1)$
return $A_{|v|}$
```

A drawback of the trie based algorithm is that it will not scale for large edit distance thresholds since it will degenerate fast into a complete traversal of the higher levels of the trie structure. A plethora of advanced trie based algorithms for reducing the number of paths traversed have been proposed in the past. The algorithms assume complete knowledge of the query a priori. The same algorithms can be straightforwardly modified to work on suffix trees. Finally, it is not hard to see that these techniques can be adapted for computing weighted edit distance (given a substitution matrix and a cost function for each edit operation).

8.3.2 B-Trees

An alternative approach is to use a B-tree structure to store the strings in a one-dimensional order that provides guarantees on the retrieval of strings based on unweighted edit distance.

To index the strings with the B-tree, it is necessary to construct a mapping from the string domain to integer space. Formally:

Definition 8.1 (String Order). Given the string domain Σ^*, a string order is a mapping function $\phi : \Sigma^* \rightarrow \mathbb{N}$, mapping each string to an integer value.

The definition above implies that the mapping function ϕ uniquely decides the string order. Therefore, we abuse notation to represent with ϕ both the actual string order imposed as well as the mapping function itself. Note that some strings might be mapped to the same integer by the function ϕ. In many cases, the use of a concrete mapping function is ineffective on both computation and storage. To alleviate this problem, it is better if we do not construct the mapping explicitly. This requirement can be fulfilled if the string order satisfies the following desirable property on comparability:

Property 8.1 (Comparability). A string order ϕ is efficiently comparable if it takes linear time to verify if $\phi(s)$ is larger than $\phi(r)$ for any string pair s and r.

The verification method is supposed to take linear time with respect to the lengths of the strings. It is easy to see that the insertion and deletion operations on the B-tree rely only on the comparability of the string order. Therefore, any string order having Property 8.1 can be used to index strings on the B-tree. Algorithm 8.3.2 can be used for locating the first leaf node of the tree potentially containing a given target string. Each intermediate node N in the B-tree contains m splitting strings $\{s_1, \ldots, s_m\}$ and $m + 1$ pointers to children nodes $\{N_1, \ldots, N_{m+1}\}$. The algorithm identifies the first splitting string with mapping value larger than that of the query string and iteratively

Algorithm 8.3.2: FindNode(v, N)

```
if $N$ is the leaf node: return $N$
for each splitting string $s_j \in N$
  do {if $\phi(v) \leq \phi(s_j)$ : return FindNode$(v, N_j)$
return FindNode$(v, N_{m+1})$
```

searches the subtree from the corresponding pointer all the way to the leaf level of the tree. The implementation of insertion, deletion, and split operations follows similar strategies by using the new comparison oracle between $\phi(s)$ and $\phi(r)$.

From Algorithm 8.3.2, we can see that all strings stored in the sub-tree rooted at N_j have mapping values in $[\phi(s_{j-1}), \phi(s_j)]$ by construction. To simplify notation, we say that $s \in [s_i, s_j]$ if $\phi(s_i) \leq \phi(s) \leq \phi(s_j)$.

The following property enables the B-tree structure to handle range queries based on edit distance:

Property 8.2 (Lower Bounding). A string order ϕ is lower bounding if it efficiently returns the minimal edit distance between string v and any $s \in [s_i, s_j]$.

With Property 8.2 the B-tree can handle range queries using Algorithm 8.3.3. In the algorithm we use $LB(s_i, [s_{j-1}, s_j])$ to denote the lower bound on the edit distance between s_i and any string $s \in [s_{j-1}, s_j]$. Algorithm 8.3.3 iteratively visits the nodes with lower bound edit distance no larger than θ and verifies the strings found at the leaf level of the tree using Algorithm 2.1.1. Notice that the algorithm might have to traverse multiple paths down the tree (as opposed to the standard B-tree traversal algorithm). The minimal and maximal strings s^l and s^u indicate the boundaries of any string in a given subtree with respect to the string order ϕ. This information can be retrieved from the parent node, as the algorithm implies. It is easy to verify that this algorithm accurately returns all strings within distance θ from query

Algorithm 8.3.3: RangeQuery(v, N, θ, s^l, s^u)

```
if $N$ is the leaf node
  do for each $s \in N$
       if Edit Distance Verification$(v, s, \theta)$
         do Include $s$ in query result
  else
  do if $LB(v, [s^l, s_1]) \leq \theta$
       do RangeQuery$(v, N_1, \theta, s^l, s_1)$
     for $j = 2$ to $m$
       do if $LB(v, [s_{j-1}, s_j]) \leq \theta$
            do RangeQuery$(v, N_j, \theta, s_{j-1}, s_j)$
     if $LB(v, [s_m, s^u]) \leq \theta$
       do RangeQuery$(v, N_{m+1}, s_m, s^u)$
```

string v if the lower bounding property holds. The efficiency of the algorithm depends on the tightness of the lower bound. We discuss concrete string orders that satisfy Property 8.2 in the next section.

If a string order ϕ supports range queries, it also directly supports top-k selection queries on the B-tree structure. We simply use a min-heap to keep the current top-k similar strings and update the threshold θ with the distance value of the top element in the heap. The detailed algorithm is shown in Algorithm 8.3.4.

The standard all-match join algorithm on B-trees for traditional one-dimensional data discovers all node pairs $\{N_1, N_2\}$ on the same level of the tree, with non-empty value overlap on their ranges. Expansions are conducted by adding join candidates after testing every pair of children drawn from N_1 and N_2, respectively. For string join queries with threshold θ the standard algorithm is applicable if the following property holds:

Property 8.3 (Pairwise Lower Bounding). Given two string intervals $[s^l, s^u]$ and $[r^l, r^u]$, the string order ϕ is pairwise lower bounding if it returns the lower bound on the distance between any $s \in [s^l, s^u]$ and any $r \in [r^l, r^u]$.

Algorithm 8.3.4: TopKQuery$(v, N, \theta, s^l, s^u, H)$

```
if $N$ is the leaf node
  do for each $s \in N$
       do if Edit Distance Verification$(v, s, \theta)$
            do Insert $s$ into $H$
               if $|H| > k : H$.pop
               Update the global threshold $\theta$
  else
  do if $LB(v, [s^l, s_1]) \leq \theta$
       do TopKQuery$(v, N_1, \theta, s^l, s_1, H)$
     for $j = 2$ to $m$
       do if $LB(v, [s_{j-1}, s_j]) \leq \theta$
            do TopKQuery$(v, N_j, \theta, s_{j-1}, s_j, H)$
     if $LB(v, [s_m, s^u]) \leq \theta$
       do TopKQuery$(v, N_{m+1}, s_m, s^u, H)$
```

We use $LB([s^l, s^u], [r^l, r^u])$ to denote the lower bound edit distance between the two string intervals. This property allows the direct adoption of the standard join algorithm, as is shown in Algorithm 8.3.5. The algorithm recursively expands the nodes in depth-first order, to give a clear idea on the joining process. In practice we can use a heap to store the node pairs and pop out the next candidate to join if it satisfies the minimal distance lower bound.

Finally, this B-tree index scheme is also capable of handling normalized edit distance. This is achievable if the maximal length of the strings within an interval can be estimated by the string order:

Property 8.4 (Length Bounding). Given any string interval $[s_i, s_j]$, the string order ϕ is length bounding if it efficiently returns an upper bound on the length of any string $s \in [s_i, s_j]$.

The length bounding property can be combined with any of the query processing algorithms, by dividing the lower bound edit distance with the maximal length of the string intervals. Therefore, this index

Algorithm 8.3.5: JoinQuery(N_1, N_2, θ)

```
if $N_1$ and $N_2$ are leaf nodes
  do for each $s \in N_1$ and $r \in N_2$
       do if Edit Distance Verification($s, r, \theta$)
            do Insert $(s, r)$ into the result
else
  do for child node $N_i$ of $N_1$ and child node $N_j$ of $N_2$
       do for all intervals $[s^l, s^u] \in N_i, [r^l, r^u] \in N_j$
            do if $LB([s_i^l, s_i^u], [s_j^l, s_j^u]) \leq \theta$
                 do JoinQuery($N_i, N_j, \theta$)
```

Table 8.2. Necessary properties with respect to query type and distance function.

	Range	Top-k	All-Match Join
Edit Distance	P8.1,P8.2	P8.1,P8.2	P8.1,P8.3
Normalized E.D.	P8.1,P8.2,P8.4	P8.1,P8.2,P8.4	P8.1,P8.3,P8.4

structure seamlessly supports both edit distance and normalized edit distance, if the underlying string order is consistent with these properties. We summarize the necessary properties the mapping function ϕ needs to have in order to be able to support the three types of string similarity queries based on edit distance and normalized edit distance in Table 8.2.

8.3.2.1 Dictionary order

Dictionary Order is the most straightforward choice for the string order. Dictionary order obeys comparability, lower bounding, and pairwise lower bounding. It does not satisfy length bounding. Therefore, it can be used to index on edit distance for range, top-k, and all-match join queries.

Given an alphabet Σ, there is a pre-defined order on all letters in Σ, i.e., $\{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$. We simply assume that the index of σ_i in Σ can be calculated by the permutation function $\pi(\sigma_i)$. Then, we map the string domain to an integer space where strings are sorted in dictionary order. We denote this sorted dictionary order with ϕ_d.

It is obvious that dictionary order follows the property of comparability. Given two strings s and r, it is sufficient to find the most significant position p where the two strings differ. If $\pi(s[p]) < \pi(r[p])$, we can assert that s precedes r in dictionary order, and vice versa. This comparison can be done in linear time with respect to the length of the strings — we do not need to actually instantiate order ϕ_d.

Dictionary order is also consistent with the property of lower bounding. Given a string interval $[\phi_d(s_i), \phi_d(s_j)]$ (or for simplicity $[s_i, s_j]$) in dictionary order, we know that all strings in this interval must share the longest common prefix of s_i and s_j, i.e., $LCP(s_i, s_j)$. To be more precise, if $s \in [s_i, s_j]$, we have:

$$\forall p \in [1, |LCP(s_i, s_j)|], \quad s[p] = s_i[p] = s_j[p]. \tag{8.1}$$

Let $p = |LCP(s_i, s_j)|$. In fact, we can actually use letter $s[p+1]$ to refine the lower bound even further:

$$s_i[p+1] \leq s[p+1] \leq s_j[p+1], \quad p = |LCP(s_i, s_j)|. \tag{8.2}$$

For example, consider the string interval [‘food’, ‘found’]. Any string within this interval must have the prefix ‘fo’ and the third character must be in the interval [‘o’, ‘u’]. The suffix after the third character can be any valid string in the alphabet (with arbitrary length). Notice that this does not imply that all of these strings with unknown arbitrary suffixes are actually contained in interval [‘food’, ‘found’]. We simply return a super-set of the strings in the interval which is sufficient for computing a correct (albeit) loose lower bound.

Equation (8.2) is valid only if $|s_i| > |LCP(s_i, s_j)|$. If $|s_i| = |LCP(s_i, s_j)|$ (i.e., s_i is completely covered by s_j), then no further refinement of the lower bound is possible.

Given Equations (8.1) and (8.2), we can now derive an efficient lower bound computation between a query string v and any string $s \in [s_i, s_j]$, based on the edit distance verification Algorithm 2.1.1. Table 8.3 shows a running example of the computation of the lower bound on the edit distance between query ‘fund’ and interval [‘food’, ‘found’], with threshold $\theta = 1$. Notice that the third row of the table uses a candidate

Table 8.3. Edit distance lower bound estimation between a string and a string interval for dictionary order.

	∅	f	u	n	d
∅	0	1	2	3	
f	1	0	1		
o	2	1	1	2	
{‘o’, …, ‘u’}	3		1	2	3

letter set {‘o’,… ,‘u’} to represent the string interval. A query letter will match the third row if and only if that letter is contained in the respective set of letters. Since the given interval provides information only on the three first characters of the strings within the interval, the algorithm stops on the third row and estimates the lower bound on the edit distance as the smallest value on that row (assuming a best case scenario where a string exists within the interval matching the suffix of the query exactly). In this case the algorithm returns 1 as the estimated lower bound between the given query string and the string interval. We skip the details of the algorithm which can be easily implemented by modifying Algorithm 2.1.1.

Similarly, we can compute a lower bound edit distance between two string intervals $[s^l, s^u]$ and $[r^l, r^u]$, by combining the prefixes of the boundary strings from these two intervals.

Notice that it is impossible to compute $\phi_d(s)$ for a given string s, given that there is an infinite number of strings in the dictionary order between any two string. As already mentioned it is not necessary to instantiate the mapping, since we can efficiently verify in linear time the order of two strings. In practice we store the actual keys inside each B-tree node instead of the mapping, which of course increases the storage requirements of this structure (as is usually the case for string B-trees).

8.3.2.2 Gram counting order

The dictionary order collects useful information only on the prefixes of the strings. In many cases, unfortunately, the discriminative information of the strings is scattered in different positions. This motivates the use of q-grams instead of prefixes to summarize the string set. In this section, we describe a string order based on counting the number of q-grams within a string. We use a hash function to map q-grams to a set of buckets of fixed cardinality. We show that the *Gram Counting Order* has the properties of comparability, lower bounding, pairwise lower bounding, and length bounding. Therefore, it can be used to index on edit distance and normalized edit distance for range and top-k selection queries, and join queries.

A q-gram set can be intuitively represented as a vector in a high dimensional space where each dimension corresponds to a distinct q-gram (i.e., the vector space model). This solution, however, incurs high storage cost. To compress the information on the vector space, we use a hash function to map each q-gram to a set of L buckets, and count the number of q-grams in each bucket. Thus, the q-gram set is transformed into a vector of L non-negative integers.

The q-gram mismatch lemma (Lemma 8.4) states that the edit distance between two strings s, r is no smaller than

$$\mathcal{E}(s,r) \geq \max\left(\frac{|Q(s) \setminus Q(r)|}{q}, \frac{|Q(r) \setminus Q(s)|}{q}\right). \tag{8.3}$$

Here, $Q(s) \setminus Q(r)$ is the set of q-grams in $Q(s)$ and not $Q(r)$, and vice versa. After mapping strings from gram space to the bucket space, a new lower bound holds. If s' and r' are the L-dimensional bucket vector representations of s and r, respectively, the edit distance between s and r is no smaller than

$$\max\left(\sum_{s'[\ell]>r'[\ell]} \frac{s'[\ell] - r'[\ell]}{q}, \sum_{r'[\ell]>s'[\ell]} \frac{r'[\ell] - s'[\ell]}{q}\right) \tag{8.4}$$

for $1 \leq \ell \leq L$. Stated simply, the edit distance should be at least as large as the largest difference in the q-gram counts between any pair of corresponding buckets in the bucket representation (correcting for the fact that one edit operation can change up to q q-grams).

To achieve a tighter lower bound we apply z-order on the q-gram counting vectors to cluster strings with similar vectors as best as possible in the one-dimensional B-tree space. Given a vector s', z-order interleaves the bits from all vector components in a round robin fashion. For example, let the binary values of the vector components be '11', '10', '01', '11', for $L = 4$. Then, the z-order value of the vector is

'11011011'. Therefore, each string is indexed in the B-tree according to the z-order value of its q-gram counting vector. Formally, the gram counting order ϕ_{gc} for string s is defined as

$$\phi_{gc}(s) = \text{zorder}(s'). \tag{8.5}$$

The property of comparability is straightforwardly satisfied since it only requires to compare the binary representations of the z-order values on vectors s' and r' to verify their order in the B-tree.

Next, we analyze the property of lower bounding. For a string interval $[s_i, s_j]$ in z-order representation, a lower bound and upper bound on the number of q-grams in each bucket for all strings in the interval can be derived. This is easy to see with an example. Let s_i and s_j have binary z-order values '11011011' and '11011110', respectively. Any string between them must have the prefix '11011', while the remaining bits can be either 0 or 1. In Figure 8.2 it can be seen that some accurate estimation on the bucket values can be recovered if the common prefix is long enough. Specifically, the first bucket B_1 is clearly 3, since both bits are deterministically decided. For the rest of the buckets, the set of possible values can also be calculated with the confirmed prefix bits.

Assume that for a given string interval $[s_i, s_j]$ the lower and upper bound values on bucket B_ℓ are B_ℓ^l and B_ℓ^u (for $1 \leq \ell \leq L$). After transforming the query v to vector v' in gram counting order, we apply Equation (8.4) using the lower and upper bound values of the buckets, to reach some new lower bound on the edit distance from v to any string contained in the interval. The pairwise lower bounding property can be achieved similarly by retrieving the bound pairs $(B_\ell^l(s), B_\ell^u(s))$ and $(B_\ell^l(r), B_\ell^u(r))$ for $[s^l, s^u]$ and $[r^l, r^u]$, respectively.

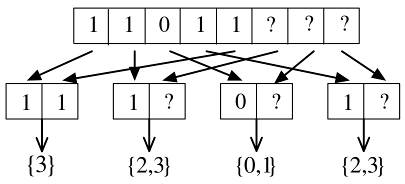


Fig. 8.2 Example of bounding the values in buckets on z-order.

The gram counting order has the length bounding property as well. Given vector s' for string s in gram counting order, the length of s is $\sum_{\ell=1}^{L} s'[\ell] - q + 1$, by the definition of q-grams. This implies that the length of the strings in string interval $[s_i, s_j]$ is bounded in the interval $[\sum_{\ell=1}^{L} B_\ell^l - q + 1, \sum_{\ell=1}^{L} B_\ell^u - q + 1]$. This allows us to correctly answer queries based on normalized, as well as standard edit distance.

8.3.2.3 Gram location order

In gram counting order the positional information of the q-grams is simply discarded. Another string order, called *Gram Location Order*, exploits the positions of the q-grams to improve edit distance based pruning. The gram location order satisfies comparability, lower bounding, and pairwise lower bounding, and hence supports all types of queries but not normalized edit distance.

To conduct the transformation, the q-grams are extracted along with their actual positions in the string. By hashing the q-grams to integers, each positional q-gram is represented by a vector of two entries, the hash value of the q-gram and the position of the q-gram. We use a hash value instead of the *idf* to avoid the updating issues related with computing *idf*s when strings are added or deleted from the B-tree.

To avoid storing large sets of hash-value/position pairs in the intermediate B-tree nodes, we sort the set elements based on the increasing two-dimensional z-order value of each element (i.e., the z-order value of pair (h_s, p_s), where h_s is the hash value of the q-gram). Then, for each set we preserve only the first L elements in the z-order, and finally sort the elements in increasing order of positions. In the rest we use s' to denote the top-L positional q-grams for string s, i.e., $s' = \{(h_1^s, p_1^s), \ldots, (h_L^s, p_L^s)\}$, in which h_i^s is the q-gram hash value, p_i^s is the corresponding position of the q-gram in the original string s, and $p_1^s \leq \cdots \leq p_L^s$.

Simply stated, we preserve only a pseudo-random subset of positional q-grams per string, and approximately compare strings based on these subsets only (the pseudo-random ordering is imposed by the hash function h and the z-order used for each vector component, which is used in order to prune both based on the q-gram hashes and the positions simultaneously).

The mapping function of gram location order is formally defined as the dictionary order on the positional q-gram set:

$$\phi_{gl}(s) = \phi_d(s'). \tag{8.6}$$

After transforming every string to the top-L positional q-gram sets, the property of comparability is guaranteed due to the property of dictionary order. In the following, we analyze the property of lower bounding and pairwise lower bounding.

Similar to dictionary order, given a string interval $[s_i, s_j]$ in gram location order, all strings in this interval share a common prefix $LCP(s'_i, s'_j)$ of positional q-grams. If a query string v is also represented by a positional q-gram set v', the lower bound on the edit distance from v to any string $s \in [s_i, s_j]$ can be estimated by counting the number of mismatched positional q-grams between v' and $LCP(s'_i, s'_j)$. For that purpose we can use the mismatch filter condition from Lemma 8.4.

Definition 8.2 (Positional Mismatch). Given edit distance threshold θ and two positional q-gram sets v' and s', a positional q-gram $(h^v_i, p^v_i) \in v'$ incurs a mismatch if either holds: 1. For all x, no $(h^s_x, p^s_x) \in s'$ exists s.t. $h^s_x = h^v_i$ and $|p^s_x - p^v_i| \leq \theta$; 2. No mismatching (h^v_j, p^v_j) has been found already, s.t. $p^v_i - p^v_j \leq \theta$ for any $p^v_j < p^v_i$.

The first condition checks for potential matches between the same q-grams in v and s within distance θ. The second condition checks whether there exists a mismatching q-gram preceding the current q-gram within distance θ from the position of the current q-gram. If such a mismatch exists, then it might be possible, in a best case scenario, to correct both q-grams with one edit operation simultaneously (since they overlap), and thus we cannot count both q-grams as mismatches. Note that the definition above assumes that each string is long enough to contain enough positional q-grams. It is easy to modify the definition in case that the number of q-grams is not large enough. We can state the following lemma.

Lemma 8.6. Given a query string v and a string interval $[s_i, s_j]$, the edit distance between v and any string $s \in [s_i, s_j]$ is lower bounded by the number of mismatches from v' to $LCP(s'_i, s'_j)$.

To get a tighter lower bound on the edit distance estimation we can also reverse the process by counting the mismatched positional q-grams from $LCP(s'_i, s'_j)$ to v'. After counting the mismatches on both directions, the larger one will be returned as the lower bound value. The algorithm takes linear time with respect to the string lengths, since all positional q-grams are sorted on positions.

On the property of pairwise lower bounding, similar techniques can be applied by constructing the longest common prefix for both string intervals $[s^l, s^u]$ and $[r^l, r^u]$. Again, the number of mismatches from both positional q-gram sets is counted, the larger of which is the estimated lower bound.

8.4 Discussion and Related Issues

The state-of-the-art filtering algorithm for edit distance computation is the mismatch filter. It can prune strings both based on the locations of matching q-grams and mismatching q-grams, which leads to tighter pruning than all other approaches. Filtering algorithms cannot be adapted to evaluate either weighted or normalized edit distance. For selection queries and datasets that fit in maim memory, the trie based algorithm is a very efficient approach, especially for computing edit distance when query strings are given incrementally, as is the case with interactive applications, since they reuse computation performed in previous steps. Trie based algorithms can be used to evaluate weighted edit distance. The B-tree algorithms are the best alternative when considering incremental updates. They also have the benefit that the same B-tree structure can be used to answer all types of queries (selections, joins, all-match, and top-k queries), without the need to specify a minimum query threshold a priori. They are also the most efficient for long strings and large edit distance thresholds. The performance of all other algorithms deteriorates significantly for large thresholds. Finally, the

B-tree algorithms can be used to evaluate normalized edit distance, but not weighted edit distance.

8.5 Related Work

Edit distance based queries have been studied in the past in literature related to the approximate dictionary lookup problem. Dictionary lookup simply returns a yes or no answer, if there exists a dictionary string within edit distance θ from a query string. Minsky and Papert [53] originally formulated the dictionary lookup for Hamming distance. Yao and Yao [77] presented a solution for Hamming distance $\theta = 1$. Brodal and Gasieniec [14] presented an improved algorithm in terms of space and query time. Manber and Wu [51] proposed an algorithm for *edit distance* with threshold $\theta = 1$. The algorithm generates all possible strings within edit distance one from each string in the dataset and indexes those strings using Bloom filters. The index can then identify whether there exists a string within edit distance one from any query string. An improved solution for edit distance one was given by Brodal and Venkatesh [13]. Arslan and Eğecioğlu [7] presented a novel algorithm based on tries and arbitrary edit distance thresholds.

The dictionary lookup problem is fundamentally easier than the approximate dictionary match problem, and both are easier than the approximate text match problem, which has received a lot of attention in the past. Most solutions for the approximate text match problem can be used to solve the approximate dictionary match problem. Nevertheless, due to its simplicity, the dictionary match problem is amenable to certain optimizations and simplifications that result in faster indexes and filtering techniques in practice. A survey of related literature on the offline text matching problem was conducted by Chan et al. [16]. A survey of the online text matching problem was conducted by Navarro [54]. A comprehensive exposition of all algorithms for online approximate string matching was conducted by Stephen [62].

The q-gram intersection filter for edit distance was first proposed by Jokinen and Ukkonen [39]. In the same paper the authors gave the first algorithm that used q-gram inverted lists to compute edit distance.

Gravano et al. [31] used similar reasoning to solve the dictionary match problem using standard relational database technology, by representing token lists as relational tables and expressing the list merging process as join queries. Efficient list merging algorithms based on the `multiway merge` strategy, independent of a DBMS, were proposed by Sarawagi and Kirpal [60]. Li et al. [48] proposed an optimization of the `multiway merge` algorithm for edit distance.

The idea of filtering techniques in information retrieval dates at least as far back as the 1970s [36]. The idea of filtration for approximate text matching using Hamming distance was first proposed by Karp and Rabin [40]. Owolabi and McGregor [57] were the first to make the observation that similar strings should have at least some substring of sufficient length in common. A filtering algorithm for the text matching problem for Hamming based on sampling the text was proposed by Grossi and Luccio [32], and later improved by Pevzner and Waterman [58]. A sampling based solution for edit distance was proposed by Chang and Lawler [17] and later simplified by Takaoka [64] using q-grams. Two filtering algorithms for edit distance based on q-grams were proposed by Ukkonen [67]. The location information of q-grams was first used by Sutinen and Tarhio [63] in combination with a partitioning strategy. Chang and Lawler [17] used the idea of partitioning the strings using the pigeonhole principle. The same partitioning strategy was also used by Wu and Manber [72]. This idea was later extended to the partenum signature by Arasu et al. [6]. Sarawagi and Kirpal [60] gave the first algorithm based on using the prefix filter principle. The prefix filter was later formalized by Chaudhuri et al. [19]. The mismatch filter was proposed by Xiao et al. [73].

Jokinen and Ukkonen [39] were the first to present an algorithm based on suffix automatons for the text matching problem using edit distance. An algorithm for the text matching problem using a suffix trie and edit distance was proposed by Ukkonen [68]. An improvement on this algorithm was proposed by Cobbs [22]. Cole et al. [23] presented an even faster algorithm. Their algorithm uses a variety of auxiliary data structures (e.g., compressed tries for subtrees of the suffix tree and a specialized index for the query) to reduce the number of children nodes that need to be traversed at every level of the compressed trie. This algorithm was later improved by Chan et al. [15]. A cache conscious extension of the same algorithm for external memory applications was proposed by Hon et al. [37].

The detailed expansion based trie algorithm for incrementally computing the edit distance of every prefix of a query string was independently proposed by Ji et al. [38] and Chaudhuri and Kaushik [20]. These algorithms were tailored specifically for computing edit distance as a user is typing a query one letter at a time. The main difference from the dictionary reporting problem is that the whole query string is not known in advance, hence certain optimizations based on knowing the query are not applicable. In addition, these algorithms reuse previous computation to efficiently retrieve the new answers, after a new letter is typed. Conceptually the two algorithms are the same. In practice they differ in the order that nodes are added in the active nodes list. Chaudhuri and Kaushik [20] need to process every node before all its children in order to maintain correctness (which necessitates maintaining a queue, resulting in some additional space). Ji et al. [38] only need to maintain a hash table, since the order in which nodes are processed is not important. The B-tree based algorithms were proposed by Zhang et al. [78].

9 Conclusion

Approximate string matching is a fundamental operation in text data management. This work presented algorithms for selection and join queries using set based and edit based similarity measures and arbitrary token weighing schemes. There exists a large body of work in text data management related to approximate string processing that was not covered here. An important aspect of approximate string matching is to efficiently incorporate domain knowledge into the search mechanism. Domain knowledge can significantly affect search results and improve query relevance in a variety of settings. Specifically, the context of a query is of key importance in the similarity between strings. For example, previous work has recognized the significance of synonyms in many text search applications [4, 5]. In most cases, synonyms are context dependent. For example, 'Avenue of the Americas' and '6th Avenue' are synonyms only in the context of New York City. Other types of domain knowledge can help index and query the data more efficiently. For example, the inherent hierarchical structure of mailing addresses is an important factor for indexing such data. Given that the vast majority of address based queries focus on particular cities or states, a simple partitioning of the data (with possible spatial overlapping) on a per city or state level can improve querying efficiency substantially. Also, such information can help decide partitioning strategies for distributed applications (e.g., in a map-reduce setting). Clearly, incorporating domain knowledge in string similarity searching is a non-trivial task. Furthermore, a large number of applications depend heavily on efficient solutions to the approximate string matching problem as a primitive operation. Examples include entity resolution, record linkage, data cleaning, and deduplication [12, 21, 28, 27] data integration [35], text analytics and more. Every application has unique properties and presents idiosyncrasies that necessitate the development of specialized solutions. It is our belief that approximate string matching will remain of fundamental importance as new application domains and problems in text data management arise, but a solid foundation will guarantee robust solutions in the future.

Acknowledgments

We would like to thank our colleagues in the field of approximate string processing and especially Chen Li, Wei Wang, Arvind Arasu, Rares Vernica, and Shengyue Ji for many useful discussions and clarifications. We are also grateful to Gerhard Weikum and the anonymous reviewers of this work for many useful suggestions and constructive criticism on the material. We would like to extend a special thanks to jpl for suggesting many valuable improvements and optimizations to various algorithms and, of course, for proof reading! We would also like to express our thanks to Now Publishers, and especially to James Finlay for making this possible.

References

[1] S. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, October 1990.

[2] A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein, "Pattern matching with swaps," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, p. 144, 1997.

[3] A. Andoni and K. Onak, "Approximating edit distance in near-linear time," in *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pp. 199–204, 2009.

[4] A. Arasu, S. Chaudhuri, and R. Kaushik, "Transformation-based framework for record matching," in *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 40–49, 2008.

[5] A. Arasu, S. Chaudhuri, and R. Kaushik, "Learning string transformations from examples," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 514–525, 2009.

[6] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *Proceedings of Very Large Data Bases (VLDB)*, pp. 918–929, 2006.

[7] A. N. Arslan and Ö. Eğecioğlu, "Dictionary look-up within small edit distance," in *Proceedings of the Annual International Conference on Computing and Combinatorics (COCOON)*, pp. 127–136, 2002.

[8] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval.* Addison Wesley, May 1999.

[9] Z. Bar-Yossef, T. S. Jayram, R. Krauthgamer, and R. Kumar, "Approximating edit distance efficiently," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 550–559, 2004.

[10] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *WWW*, pp. 131–140, 2007.

[11] A. Behm, S. Ji, C. Li, and J. Lu, "Space-constrained gram-based indexing for efficient approximate string search," in *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 604–615, 2009.

[12] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom, "Swoosh: A generic approach to entity resolution," *The VLDB Journal*, vol. 18, no. 1, pp. 255–276, 2009.

[13] G. Brodal and S. Venkatesh, "Improved bounds for dictionary look-up with one error," *Information Processing Letters*, vol. 75, no. 1–2, pp. 57–59, 2000.

[14] G. S. Brodal and L. Gasieniec, "Approximate dictionary queries," in *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 65–74, 1996.

[15] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong, "Compressed indexes for approximate string matching," in *Proceedings of the Annual European Symposium (ESA)*, pp. 208–219, 2006.

[16] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong, "A linear size index for approximate pattern matching," in *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 49–59, 2006.

[17] W. I. Chang and E. L. Lawler, "Approximate string matching in sublinear expected time," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 116–124, vol. 1, 1990.

[18] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and efficient fuzzy match for online data cleaning," in *Proceedings of ACM Management of Data (SIGMOD)*, pp. 313–324, 2003.

[19] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *Proceedings of International Conference on Data Engineering (ICDE)*, p. 5, 2006.

[20] S. Chaudhuri and R. Kaushik, "Extending autocompletion to tolerate errors," in *Proceedings of ACM Management of Data (SIGMOD)*, pp. 707–718, 2009.

[21] S. Chaudhuri, A. D. Sarma, V. Ganti, and R. Kaushik, "Leveraging aggregate constraints for deduplication," in *Proceedings of ACM Management of Data (SIGMOD)*, pp. 437–448, 2007.

[22] A. Cobbs, "Fast approximate matching using suffix trees," in *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 41–54, 1995.

[23] R. Cole, L.-A. Gottlieb, and M. Lewenstein, "Dictionary matching and indexing with errors and don't cares," in *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pp. 91–100, 2004.

[24] D. Comer, "The ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

[25] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[26] G. Cormode and S. Muthukrishnan, "The string edit distance matching problem with moves," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 1, pp. 1–19, 2007.

[27] M. G. Elfeky, A. K. Elmagarmid, and V. S. Verykios, "Tailor: A record linkage tool box," in *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 17–28, 2002.

[28] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 19, no. 1, pp. 1–16, 2007.

[29] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pp. 102–113, 2001.

[30] P. Ferragina and R. Grossi, "The string b-tree: A new data structure for string search in external memory and its applications," *Journal of the ACM (JACM)*, vol. 46, pp. 236–280, 1999.

[31] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *Proceedings of Very Large Data Bases (VLDB)*, pp. 491–500, 2001.

[32] R. Grossi and F. Luccio, "Simple and efficient string matching with k mismatches," *Information Processing Letters*, vol. 33, no. 3, pp. 113–120, 1989.

[33] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, "Fast indexes and algorithms for set similarity selection queries," in *Proceedings of International Conference on Data Engineering (ICDE)*, 2008.

[34] M. Hadjieleftheriou, N. Koudas, and D. Srivastava, "Incremental maintenance of length normalized indexes for approximate string matching," in *Proceedings of ACM Management of Data (SIGMOD)*, pp. 429–440, 2009.

[35] A. Halevy, A. Rajaraman, and J. Ordille, "Data integration: The teenage years," in *Proceedings of Very Large Data Bases (VLDB)*, pp. 9–16, 2006.

[36] M. C. Harrison, "Implementation of the substring test by hashing," *Communications of the ACM*, vol. 14, no. 12, pp. 777–779, 1971.

[37] W.-K. Hon, T.-W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter, "Cache-oblivious index for approximate string matching," in *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 40–51, 2007.

[38] S. Ji, G. Li, C. Li, and J. Feng, "Efficient interactive fuzzy keyword search," in *WWW*, pp. 371–380, 2009.

[39] P. Jokinen and E. Ukkonen, "Two algorithms for approximate string matching in static texts," in *Proceedings of the Mathematical Foundations of Computer Science (MFCS)*, pp. 240–248, 1991.

[40] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[41] J. D. Kececioglu and D. Sankoff, "Exact and approximation algorithms for the inversion distance between two chromosomes," in *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 87–105, 1993.

[42] D. K. Kim, J.-S. Lee, K. Park, and Y. Cho, "Efficient algorithms for approximate string matching with swaps," *Journal of Complexity*, vol. 15, no. 1, pp. 128–147, 1999.

[43] D. E. Knuth, "The art of computer programming, volume 3 (2nd ed.)," in *Sorting and Searching*, Addison Wesley Longman Publishing Co., Inc., 1998.

[44] N. Koudas, A. Marathe, and D. Srivastava, "Flexible string matching against large databases in practice," in *Proceedings of Very Large Data Bases (VLDB)*, pp. 1078–1086, 2004.

[45] N. Koudas, A. Marathe, and D. Srivastava, "Propagating updates in SPIDER," in *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 1146–1153, 2007.

[46] N. Lester, A. Moffat, and J. Zobel, "Efficient online index construction for text databases," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 3, pp. 1–33, 2008.

[47] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals (in Russian)," *Doklady Akademii Nauk SSSR*, vol. 163, no. 4, pp. 845–848, 1965.

[48] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 257–266, 2008.

[49] C. Li, B. Wang, and X. Yang, "Vgram: Improving performance of approximate queries on string collections using variable-length grams," in *Proceedings of Very Large Data Bases (VLDB)*, pp. 303–314, 2007.

[50] J. P. Linderman, Personal Communication, 2011.

[51] U. Manber and S. Wu, "An algorithm for approximate membership checking with application to password security," *Information Processing Letters*, vol. 50, no. 4, pp. 191–197, 1994.

[52] W. J. Masek and M. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System Sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[53] M. L. Minsky and S. A. Papert, *Perceptrons*: *An Introduction to Computational Geometry.* MIT Press, 1969.

[54] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[55] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[56] R. Ostrovsky and Y. Rabani, "Low distortion embeddings for edit distance," *Journal of the ACM*, vol. 54, no. 5, pp. 23–36, 2007.

[57] O. Owolabi and D. R. McGregor, "Fast approximate string matching," *Software — Practice & Experience*, vol. 18, no. 4, pp. 387–393, 1988.

[58] P. A. Pevzner and M. S. Waterman, "A fast filtration algorithm for the substring matching problem," in *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 197–214, 1993.

[59] S. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computing Surveys*, vol. 39, no. 2, p. 4, 2007.

[60] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *Proceedings of ACM Management of Data (SIGMOD)*, pp. 743–754, 2004.

[61] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[62] G. A. Stephen, *String Searching Algorithms.* World Scientific Publishing Co., 1998.

[63] E. Sutinen and J. Tarhio, "On using q-gram locations in approximate string matching," in *Proceedings of the Annual European Symposium (ESA)*, pp. 327–340, 1995.

[64] T. Takaoka, "Approximate pattern matching with samples," in *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, pp. 234–242, 1994.

[65] W. F. Tichy, "The string-to-string correction problem with block moves," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 309–321, 1984.

[66] E. Ukkonen, "Algorithms for approximate string matching," *Information and Control*, vol. 64, no. 1–3, pp. 100–118, 1985.

[67] E. Ukkonen, "Approximate string-matching with q-grams and maximal matches," *Theoretical Computer Science*, vol. 92, no. 1, pp. 191–211, 1992.

[68] E. Ukkonen, "Approximate string matching over suffix trees," in *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 228–242, 1993.

[69] R. Vernica and C. Li, "Efficient top-k algorithms for fuzzy search in string collections," in *Proceedings of the International Workshop on Keyword Search on Structured Data (KEYS)*, pp. 9–14, 2009.

[70] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.

[71] I. H. Witten, T. C. Bell, and A. Moffat, *Managing Gigabytes*: *Compressing and Indexing Documents and Images.* John Wiley & Sons, Inc., 1994.

[72] S. Wu and U. Manber, "Fast text searching: Allowing errors," *Communications of the ACM (CACM)*, vol. 35, no. 10, pp. 83–91, 1992.

[73] C. Xiao, W. Wang, and X. Lin, "Ed-join: An efficient algorithm for similarity joins with edit distance constraints," in *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 933–944, 2008.

[74] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 916–927, 2009.

[75] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *WWW*, pp. 131–140, 2008.

[76] X. Yang, B. Wang, and C. Li, "Cost-based variable-length-gram selection for string collections to support approximate queries efficiently," in *Proceedings of ACM Management of Data (SIGMOD)*, pp. 353–364, 2008.

[77] A. C. Yao and F. F. Yao, "Dictionary look-up with one error," *Journal of Algorithms*, vol. 25, no. 1, pp. 194–202, 1997.

[78] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava, "B^{ed}-tree: An all-purpose index structure for string similarity search based on edit distance," in *Proceedings of ACM Management of Data (SIGMOD)*, 2010.

[79] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys*, vol. 38, no. 2, p. 6, 2006.