A Pivotal Prefix Based Filtering Algorithm for String Similarity Search

Dong Deng, Guoliang Li, Jianhua Feng
Department of Computer Science, Tsinghua University, Beijing, China
dd11@mails.tsinghua.edu.cn, liguoliang@tsinghua.edu.cn,fengjh@tsinghua.edu.cn

ABSTRACT

We study the string similarity search problem with edit-distance constraints, which, given a set of data strings and a query string, finds the similar strings to the query. Existing algorithms use a signature-based framework. They first generate signatures for each string and then prune the dissimilar strings which have no common signatures to the query. However existing methods involve large numbers of signatures and many signatures are unnecessary. Reducing the number of signatures not only increases the pruning power but also decreases the filtering cost. To address this problem, we propose a novel pivotal prefix filter which significantly reduces the number of signatures. We prove the pivotal filter achieves larger pruning power and less filtering cost than state-of-the-art filters. We develop a dynamic programming method to select high-quality pivotal prefix signatures to prune dissimilar strings with non-consecutive errors to the query. We propose an alignment filter that considers the alignments between signatures to prune large numbers of dissimilar pairs with consecutive errors to the query. Experimental results on three real datasets show that our method achieves high performance and outperforms the state-of-the-art methods by an order of magnitude.

Categories and Subject Descriptors
H.2 [Database Management]: Database applications;
H.3.3 [Information Search and Retrieval]: Search process

Keywords
Similarity search; pivotal prefix filter; edit distance

1. INTRODUCTION

String similarity search that finds similar strings of a query string from a given string collection is an important operation in data cleaning and integration. It has attracted significant attention due to its widespread real-world applications, such as spell checking, copy detection, entity linking, and macromolecules sequence alignment [9,13,26]. Edit distance is a well-known metrics to quantify the similarity between strings. Two strings are said to be similar if their edit distance is not larger than an edit-distance threshold. Many existing systems, e.g., Oracle[1], PostgreSQL[2] and Lucene[3], support string similar search with edit-distance constraints.

Existing studies to address this problem usually employ a filter-verification framework [3,6,9,13,14,26]. In the filter step, they devise effective filters, utilize the filters to prune large numbers of dissimilar strings, and obtain some candidates. In the verification step, they verify the candidates to generate the final answers by computing their real edit distances to the query. Existing methods focus on devising effective filters to achieve high pruning power. The prefix filter [2,4] is a dominant filtering technique. It first generates a prefix for each string such that if two strings are similar, their prefixes share a common signature. It then utilizes this property to prune large numbers of dissimilar strings that have no common signatures to the query. Some signature-based filters are proposed to improve the prefix filter, such as position prefix filter [24,25], adaptive prefix filter [21], and symmetric filter [17].

It is worth noting that the number of signatures has an important effect on the pruning power and filtering cost. On one hand, reducing the number of signatures will decrease the matching probability between the signatures of two strings, and thus the pruning power will become larger. On the other hand, reducing the number of signatures will decrease the comparison cost to check whether two strings share common signatures, and the filtering cost will also decrease. Accordingly, reducing the number of signatures not only increases the pruning power but also decreases the filtering cost. However, simply reducing the number of signatures may lead to miss results, and it calls for effective methods which can reduce the number of signatures without missing any results.

To address this problem, we propose a pivotal prefix filter which can significantly reduce the number of signatures while still finding all results. We prove that the pivotal prefix filter outperforms state-of-the-art filters in terms of both pruning power and filtering cost. As there may be multiple strategies to generate pivotal prefixes, we develop a dynamic programming method to select high-quality pivotal prefix signatures in order to prune large numbers of dissimilar strings with non-consecutive errors to the query. When there are many consecutive errors, the pivotal prefix filter may generate many false positives. To address this issue, we

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2593675.

[1] www.oracle-base.com/articles/11g/utl_match-string-matching-in-oracle.php#edit_distance
[2] www.postgresql.org/docs/8.3/static/fuzzystrmatch.html
[3] lucene.apache.org/core/4_6_1/suggest/index.html

propose an alignment filter that considers the alignments between signatures to prune large numbers of dissimilar pairs with consecutive errors to the query.

To summarize, we make the following contributions.

- We propose a novel pivotal prefix filter which has much smaller signature size than existing filters while still finding all results. We prove that our pivotal prefix filter has larger pruning power and less filtering cost than state-of-the-art filters.
- We develop effective techniques to select high-quality pivotal prefix signatures to prune dissimilar strings with non-consecutive errors to the query.
- We propose an alignment filter to prune dissimilar strings with consecutive errors to the query.
- Experimental results show the superiority of our method in terms of both performance and pruning power compared to state-of-the-art approaches.

The rest of the paper is organized as follows. We formulate the problem and review related works in Section 2. Section 3 introduces our pivotal prefix filter. We present the framework in Section 4 and propose to select high-quality pivotal signatures in Section 5. We propose an alignment filter in Section 6 and generalize the method in Section 7. We conduct experiments in Section 8 and conclude in Section 9.

2. PRELIMINARIES

2.1 Problem Definition

The string similarity search problem takes as input a string dataset, a query string, a similarity metrics, and a similarity threshold and outputs all the strings in the dataset that are similar to the query string. A variety of functions have been proposed to measure the similarity of two strings [20]. In this paper we focus on the edit-distance function. The edit distance between two strings r and s is the minimum number of edit operations needed to transform one string to another, denoted as $\texttt{ed}(r, s)$. There are three kinds of edit operations, insertion, deletion and substitution. For example, $\texttt{ed}(\text{“}\texttt{youtbecom}\text{”}, \text{“}\textit{yotdecom}\text{”}) = 2$ as the first one can be transformed to the second one by deleting '`u`' and substituting '`b`' for 'd'.

Next we formally define the similarity search problem with edit-distance constraints as below.

Definition 1 (String Similarity Search). *Given a string dataset $\mathcal{R}$, a query string s, and an edit-distance threshold τ, the string similarity search with edit-distance constraints finds all strings $r \in \mathcal{R}$ such that $\texttt{ed}(r, s) \leq \tau$.*

For example, consider the dataset $\mathcal{R}$ with 5 strings and the query $s =$ “`yotubecom`” in Table 1. Suppose the edit-distance threshold is $\tau = 2$. $r_4 =$ “*youtbecom*” is a result as $\texttt{ed}(\text{“}\texttt{yotubecom}\text{”}, \text{“}\textit{youtbecom}\text{”}) = 2 \leq \tau$.

2.2 q-gram-based Filters

q-grams. Many existing studies use q-grams to support similarity search. A q-gram of a string is its substring with length q. For example, the 2-gram set of the query string $s =$ “`youtbecom`” in Table 1 is {`yo`,`ou`,`ut`,`tb`,`be`,`ec`,`co`,`om`}. A *positional* q-gram is the q-gram associated with its start position in the string. For example, $\langle \texttt{yo}, 1\rangle, \langle \texttt{ou}, 2\rangle, \cdots, \langle \texttt{om}, 8\rangle$ are positional q-grams of s. We use q-grams and *positional* q-grams interchangeably if the context is clear.

Count Filter. As one edit operation destroys at most τ q-grams, if two strings are similar, they require to share enough common q-grams. Using this property, existing methods propose the count filter [13]: Given two strings r and s, if they share less than $\max(|r|, |s|) - q + 1 - q\tau$ common grams, they cannot be similar, where $|r|$ is the length of r.

Prefix Filter. For each string s, the prefix filter first sorts the q-grams using a universal order and selects the first $q\tau+1$ q-grams as the prefix. Given two strings r and s, based on the count filter, if their prefixes have no common q-gram, they must have less than $\max(|r|, |s|) - q + 1 - q\tau$ common q-grams. Using this property, existing methods use the prefix filter to do pruning [24]: Given two strings r and s, if their prefixes have no common q-gram, they cannot be similar.

Mismatch Filter. The mismatch filter [24] improves the prefix filter by selecting the minimum subset of the prefix which requires $\tau + 1$ edit operations to destroy the q-grams in the subset. Obviously the prefix size of the mismatch filter is between $\tau + 1$ and $q\tau + 1$.

Length Filter. If the length difference of two strings is larger than τ, they cannot be similar.

Position Filter. When checking whether the prefixes of two strings have common signatures, we only check their signatures with position difference within τ, because for any transformation from a string to another in which the two signatures with position difference larger than τ are aligned, the number of edit operations in the transformation must be larger than τ (as the length difference of prefixes of the two strings before the matching signatures is larger than τ).

Comparison with Existing Filters. Bayardo et. al. [2] proposed a framework to support similarity search and join using the prefix filter. Xiao et. al. proposed the position prefix filter [25] and mismatch filter [24] to reduce the signature size. Although we utilize the same filter-and-verification framework to support similarity search, we propose the pivotal prefix to further reduce the signature size and develop an alignment filter to effectively detect consecutive errors.

2.3 Related Work

String Similarity Search. There are many studies on string similarity search [3,6,9,13,14,26]. Most of them utilized a signature-based framework where strings are similar to the query only if they share common signatures with the query. Li et. al. [13] proposed to use q-grams as signatures and studied how to efficiently merge q-gram-based inverted lists to achieve high search performance. Zhang et. al. [26] also used q-grams as signatures and utilized B^+-tree index structure to index q-grams and support range and ranking queries. Li et. al. [14] proposed variable length q-grams (VGram) as signatures to support string similarity search. Chaudhuri et. al. [3] proposed a new similarity function and devised efficient search algorithms to support the function. Hadjieleftheriou et. al. [9] addressed the data update problem in similarity search. Deng et. al. [6] studied the top-k similarity search problem using a trie index structure.

Similarity Joins. The string similarity joins that find all similar string pairs from two datasets have also been studied [1,2,4,7,8,11,15,17,19,21,22,25]. An experimental study is taken in [11]. AllPair [2] utilized the prefix filter to do similarity joins. `PPJoin` [25] proposed the prefix position filter and the suffix position filter to improve AllPair. ED-Join [24] proposed the location-based mismatch filter and the content filter to reduce the prefix length and detect consecutive errors. `AdaptJoin` [21] proposed an adaptive prefix filter framework to adaptively select the prefixes. PassJoin [15] partitioned strings into a fixed number of segments, took segments as signature and studied how to select the minimum number of substrings to reduce the number of candi-

Table 1: Dataset $\mathcal{R}$ and Query String s.

(a) Dataset $\mathcal{R}$, $q = 2$ and $\tau = 2$

id	string	$\lvert\mathtt{q}(r)\rvert$	positional prefix q-gram $\mathtt{pre}(r)$	universal order $\langle order, q\text{-gram}, frequency\rangle$
r_1	imyouteca	8	$\{\langle im,1\rangle,\langle my,2\rangle,\langle te,6\rangle,\langle ca,8\rangle,\langle yo,3\rangle\}$	$\langle 1:im,1\rangle\langle 2:my,1\rangle\langle 3:te,1\rangle\langle 4:bu,1\rangle$
r_2	ubuntucom	8	$\{\langle bu,2\rangle,\langle un,3\rangle,\langle nt,4\rangle,\langle uc,6\rangle,\langle om,8\rangle\}$	$\langle 5:un,1\rangle\langle 6:nt,1\rangle\langle 7:uc,1\rangle\langle 8:bb,1\rangle$
r_3	utubbecou	8	$\{\langle bb,4\rangle,\langle ou,8\rangle,\langle ut,1\rangle,\langle ub,3\rangle,\langle co,7\rangle\}$	$\langle 9:tb,1\rangle\langle 10:oy,1\rangle\langle 11:yt,1\rangle\langle 12:ca,2\rangle$
r_4	youtbecom	8	$\{\langle tb,4\rangle,\langle om,8\rangle,\langle yo,1\rangle,\langle ou,2\rangle,\langle ut,3\rangle\}$	$\langle 13:om,2\rangle\langle 14:yo,3\rangle\langle 15:ou,3\rangle\langle 16:ut,3\rangle$
r_5	yoytubeca	8	$\{\langle oy,2\rangle,\langle yt,3\rangle,\langle ca,8\rangle,\langle yo,1\rangle,\langle ub,5\rangle\}$	$\langle 17:ub,3\rangle\langle 18:co,3\rangle\langle 19:tu,3\rangle\langle 20:be,3\rangle\langle 21:ec,4\rangle$

(b) Query

s =yotubecom	$\mathtt{pre}(s) = \{\langle ot,2\rangle,\langle om,8\rangle,\langle yo,1\rangle,\langle ub,4\rangle,\langle co,7\rangle\}$

Table 2: Notation Table

$\mathtt{q}(r)/\mathtt{q}(s)$	q-gram set of r/s sorted by a universal order.
$\mathtt{pre}(r)/\mathtt{pre}(s)$	prefix of $\mathtt{q}(r)/\mathtt{q}(s)$ with size $q\tau+1$.
$\mathtt{suf}(r)/\mathtt{suf}(s)$	suffix of $\mathtt{q}(r)/\mathtt{q}(s)$ with size $\lvert\mathtt{q}(r)\rvert - \lvert\mathtt{pre}(r)\rvert/\lvert\mathtt{q}(s)\rvert - \lvert\mathtt{pre}(s)\rvert$.
$\mathtt{piv}(r)/\mathtt{piv}(s)$	$\tau+1$ disjoint q-grams selected from $\mathtt{pre}(r)/\mathtt{pre}(s)$.

dates. Qchunk [17] partitioned strings into several chunks with a fixed length as signatures and probed the q-grams to generate candidates. Vchunk [22] improved Qchunk by allowing various lengths of chunks. PartEnum [1] defined a model to evaluate the candidate size based on the signature. We extend the model to evaluate the filtering cost and pruning power and utilize it to guide the filter design. TrieJoin [19] utilized trie to support similarity joins. Gravano et. al. [8] implemented similarity joins in databases.

Query Autocompletion: There are some studied on query autocompletion [5,10,16,23], which finds strings with prefixes similar to the query. Ji et. al. [10] and Chaudhuri et. al. [5] proposed to use a trie index to answer the query. Xiao et. al. [23] proposed a neighborhood deletion based method to solve the query autocompletion problem. Kim et. al. [12] solved the top-k approximate substring matching problem.

3. THE PIVOTAL PREFIX FILTER

This section proposes a novel pivotal prefix filter(Section 3.1) and compares it with state-of-the-art filters(Section 3.2).

3.1 The Pivotal Prefix Filter

Given two strings r and s, we first split each of them into a set of q-grams, denoted as $\mathtt{q}(r)$ and $\mathtt{q}(s)$. Then we sort q-grams in $\mathtt{q}(r)$ and $\mathtt{q}(s)$ by a universal order, e.g., q-gram frequency in ascending order. Without loss of generality, we first suppose the edit-distance threshold is given as τ and will discuss how to support queries with different thresholds in Section 7. We denote the prefixes of $\mathtt{q}(r)$ and $\mathtt{q}(s)$ with size $q\tau+1$ as $\mathtt{pre}(r)$ and $\mathtt{pre}(s)$ respectively (see Table 2). The prefix filter prunes $\langle r,s\rangle$, if $\mathtt{pre}(r)\cap\mathtt{pre}(s)=\phi$ [2].

It is worth noting that some q-grams in the prefix are unnecessary and we can select a subset of the prefix to do further pruning. Obviously reducing the number of q-grams not only increases the filtering power but also decreases the filtering cost. For ease of presentation, we first introduce some notations as shown in Table 2. We denote the suffix of r with size $\lvert\mathtt{q}(r)\rvert - \lvert\mathtt{pre}(r)\rvert$ as $\mathtt{suf}(r)$, i.e. $\mathtt{suf}(r) = \mathtt{q}(r) - \mathtt{pre}(r)$. Two q-grams are said to be *disjoint* if they have no overlap (i.e., the difference of their start positions is no smaller than q). We select $\tau+1$ *disjoint* q-grams from $\mathtt{pre}(r)$ and denote the set of these q-grams as $\mathtt{piv}(r)$. It is worth noting that there may be multiple cases of $\tau+1$ *disjoint* q-grams in the prefix. Here we use $\mathtt{piv}(r)$ to denote any set of $\tau+1$ *disjoint* q-grams in $\mathtt{piv}(r)$. We will prove that there must exist $\tau+1$ *disjoint* q-grams in the prefix for any string and discuss how to select high-quality *disjoint* q-grams in Section 5. Obviously we have $\mathtt{piv}(r)\subseteq\mathtt{pre}(r)$. We call the q-grams in $\mathtt{pre}(r)$ as *prefix q-grams* and the q-grams in $\mathtt{piv}(r)$ as *pivotal q-grams*. The set of prefix q-grams is called *prefix* and the set of pivotal q-grams is called *pivotal prefix*.

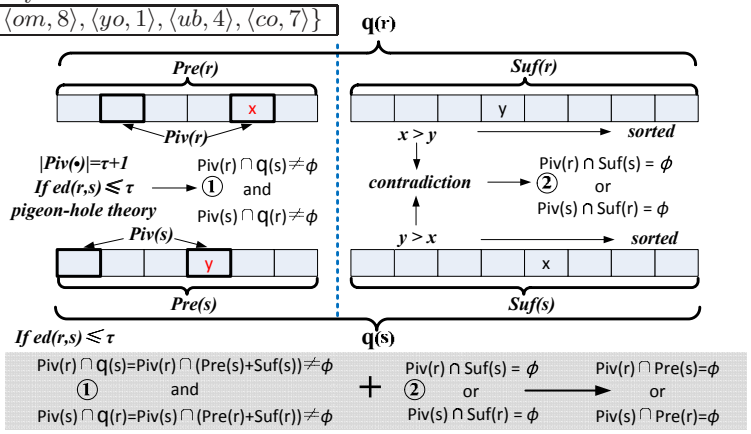


Figure 1: The Pivotal Prefix Filter.

EXAMPLE 1. *Consider the query string s and the data string r_1 in Table 1. The global q-gram order is shown in Table 1. Using this global order, we have* $\mathtt{q}(r_1) = \{\mathtt{im,my,te,ca,yo,ou,ut,ec}\}$ *and* $\mathtt{q}(s) = \{\mathtt{ot,om,yo,ub,co,tu,be,ec}\}$. *Suppose $\tau=2$, the prefix size is $q\tau+1=5$. We have* $\mathtt{pre}(r_1) = \{\mathtt{im,my,te,ca,yo}\}$, $\mathtt{pre}(s) = \{\mathtt{ot,om,yo,ub,co}\}$, $\mathtt{suf}(r_1) = \{\mathtt{ou,ut,ec}\}$ *and* $\mathtt{suf}(s) = \{\mathtt{tu,be,ec}\}$. *Here we randomly select $\tau+1$ disjoint q-grams and set* $\mathtt{piv}(r_1) = \{\mathtt{im,te,ca}\}$ *and* $\mathtt{piv}(s) = \{\mathtt{ot,om,ub}\}$.

Cross Prefix Filter. For any strings r and s, we have if $\mathtt{pre}(r)\cap\mathtt{piv}(s)=\phi$ and $\mathtt{piv}(r)\cap\mathtt{pre}(s)=\phi$, r and s cannot be similar. Lemma 1 proves the correctness of this observation and Figure 1 also illustrates the basic idea why the observation is correct.

LEMMA 1. *If strings r and s are similar, $\mathtt{piv}(r)\cap\mathtt{pre}(s)\neq\phi$ or $\mathtt{pre}(r)\cap\mathtt{piv}(s)\neq\phi$.*

PROOF. First, we prove $\mathtt{piv}(r)\cap\mathtt{suf}(s)=\phi$ or $\mathtt{piv}(s)\cap\mathtt{suf}(r)=\phi$. We can prove it by contradiction. Suppose $\mathtt{piv}(r)\cap\mathtt{suf}(s)\neq\phi$ and $\mathtt{piv}(s)\cap\mathtt{suf}(r)\neq\phi$, and $x\in\mathtt{piv}(r)\cap\mathtt{suf}(s)$ and $y\in\mathtt{piv}(s)\cap\mathtt{suf}(r)$. On one hand we have $x\in\mathtt{piv}(r)$ and $y\in\mathtt{suf}(r)$. As the q-grams are globally sorted, $x<y$[4]. On the other hand we have $y\in\mathtt{piv}(s)$ and $x\in\mathtt{suf}(s)$, and thus $y<x$. There is a contradiction and thus $\mathtt{piv}(r)\cap\mathtt{suf}(s)=\phi$ or $\mathtt{piv}(s)\cap\mathtt{suf}(r)=\phi$.

Second, we prove if r and s are similar, $\mathtt{piv}(r)\cap\mathtt{pre}(s)+\mathtt{piv}(r)\cap\mathtt{suf}(s)\neq\phi$ and $\mathtt{piv}(s)\cap\mathtt{pre}(r)+\mathtt{piv}(s)\cap\mathtt{suf}(r)\neq\phi$. As $\mathtt{q}(s)=\mathtt{pre}(s)+\mathtt{suf}(s)$, $\mathtt{piv}(r)\cap\mathtt{q}(s)=\mathtt{piv}(r)\cap\mathtt{pre}(s)+\mathtt{piv}(r)\cap\mathtt{suf}(s)$. We first prove that if r and s are similar, $\mathtt{piv}(r)\cap\mathtt{q}(s)\neq\phi$. We can prove it by contradiction. Suppose $\mathtt{piv}(r)\cap\mathtt{q}(s)=\phi$. As (1) one edit operation changes at most one disjoint q-gram in $\mathtt{piv}(r)$ and (2) $\mathtt{piv}(r)$ contains $\tau+1$ disjoint q-grams, it requires at least $\tau+1$ edit operations to

[4]When determining the global order, we avoid $x=y$ by taking the q-gram position into account.

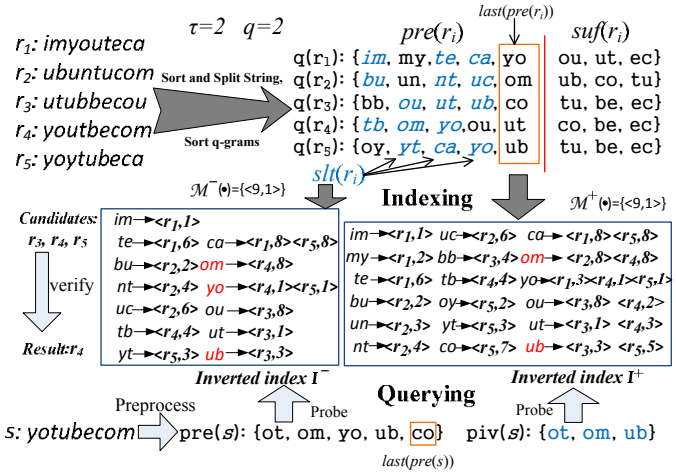


Figure 2: A Running Example.

transform r to s. This contradicts r and s are similar. Thus $\texttt{piv}(r) \cap \texttt{q}(s) \neq \phi$, i.e., $\texttt{piv}(r) \cap \texttt{pre}(s) + \texttt{piv}(r) \cap \texttt{suf}(s) \neq \phi$. Similarly we prove $\texttt{piv}(s) \cap \texttt{pre}(r) + \texttt{piv}(s) \cap \texttt{suf}(r) \neq \phi$.

Based on the first conclusion, we have two cases: (1) If $\texttt{piv}(r) \cap \texttt{suf}(s) = \phi$, as $\texttt{piv}(r) \cap \texttt{pre}(s) + \texttt{piv}(r) \cap \texttt{suf}(s) \neq \phi$ based on the second conclusion, we have $\texttt{piv}(r) \cap \texttt{pre}(s) \neq \phi$; (2) $\texttt{piv}(s) \cap \texttt{suf}(r) = \phi$, as $\texttt{piv}(s) \cap \texttt{pre}(r) + \texttt{piv}(s) \cap \texttt{suf}(r) \neq \phi$ based on the second conclusion, we have $\texttt{piv}(s) \cap \texttt{pre}(r) \neq \phi$. In any case, $\texttt{piv}(r) \cap \texttt{pre}(s) \neq \phi$ or $\texttt{pre}(r) \cap \texttt{piv}(s) \neq \phi$. Thus the lemma is proved. □

Recall the query string s and the data string r_1 in Example 1. As $\texttt{pre}(r_1) \cap \texttt{pre}(s) = \{\texttt{yo}\} \neq \phi$, the prefix filter cannot prune this pair. However, our method can prune this dissimilar pair as $\texttt{pre}(r_1) \cap \texttt{piv}(s) = \phi$ and $\texttt{piv}(r_1) \cap \texttt{pre}(s) = \phi$.

The cross prefix filter needs to check two intersections $\texttt{piv}(r) \cap \texttt{pre}(s)$ and $\texttt{pre}(r) \cap \texttt{piv}(s)$, because only if $\texttt{piv}(r) \cap \texttt{pre}(s) = \phi$ and $\texttt{piv}(s) \cap \texttt{pre}(r) = \phi$, we can prune the pair. Next we will prove that we can only check one intersection. For ease of presentation, we denote the last q-gram in $\texttt{pre}(r)$ as $\textsf{last}(\texttt{pre}(r))$. For example, the last q-grams of the five strings and query string in Table 1 are "yo, om, co, ut, ub and co" respectively. Based on $\textsf{last}(\texttt{pre}(r))$ and $\textsf{last}(\texttt{pre}(s))$, we propose a novel pivotal prefix filter and Lemma 2 proves the correctness of this filter.

Pivotal Prefix Filter. For any strings r and s, we have
Case 1: $\textsf{last}(\texttt{pre}(r)) > \textsf{last}(\texttt{pre}(s))$. If $\texttt{piv}(s) \cap \texttt{pre}(r) = \phi$, r and s cannot be similar;
Case 2: $\textsf{last}(\texttt{pre}(r)) \leq \textsf{last}(\texttt{pre}(s))$. If $\texttt{piv}(r) \cap \texttt{pre}(s) = \phi$, r and s cannot be similar.

LEMMA 2. *If strings r and s are similar, we have*
If $\textsf{last}(\texttt{pre}(r)) > \textsf{last}(\texttt{pre}(s))$, $\texttt{piv}(s) \cap \texttt{pre}(r) \neq \phi$;
If $\textsf{last}(\texttt{pre}(r)) \leq \textsf{last}(\texttt{pre}(s))$, $\texttt{piv}(r) \cap \texttt{pre}(s) \neq \phi$.

PROOF. Consider the first case, $\textsf{last}(\texttt{pre}(r)) > \textsf{last}(\texttt{pre}(s))$. We first prove $\texttt{piv}(s) \cap \texttt{suf}(r) = \phi$. For any q-gram $g \in \texttt{piv}(s)$ and $g' \in \texttt{suf}(r)$, as the q-grams are sorted, we have $g \leq \textsf{last}(\texttt{pre}(s)) < \textsf{last}(\texttt{pre}(r)) < g'$, which means $g \neq g'$. Thus $\texttt{piv}(s) \cap \texttt{suf}(r) = \phi$. Moreover, we have if r and s are similar, $\texttt{piv}(s) \cap \texttt{q}(r) = \texttt{piv}(s) \cap \texttt{pre}(r) + \texttt{piv}(s) \cap \texttt{suf}(r) \neq \phi$ as proved in Lemma 1. Thus $\texttt{piv}(s) \cap \texttt{pre}(r) \neq \phi$.

Similarly, for the second case $\textsf{last}(\texttt{pre}(r)) \leq \textsf{last}(\texttt{pre}(s))$, we can prove that $\texttt{piv}(r) \cap \texttt{pre}(s) \neq \phi$. □

EXAMPLE 2. *Consider the query string s and the data string r_2 in Table 1. We have $\texttt{pre}(r_2) = \{\texttt{bu,un,nt,uc,om}\}$ and suppose $\texttt{piv}(r_2) = \{\texttt{bu,nt,uc}\}$. As $\texttt{piv}(s) \cap \texttt{pre}(r_2) = \{\texttt{om}\} \neq \phi$, the filter based on Lemma 1 cannot prune this pair. However, the pivotal prefix filter based on Lemma 2 can prune this dissimilar pair as $\textsf{last}(\texttt{pre}(r_2)) = \texttt{om} < \textsf{last}(\texttt{pre}(s)) = \texttt{co}$ and $\texttt{piv}(r_2) \cap \texttt{pre}(s) = \phi$.*

Moreover, we can use the mismatch filter [24] to shorten the prefix $\texttt{pre}(r)$. It is worth noting that our pivotal prefix filter still works for the shorter prefix and we will prove there must exist at least $\tau+1$ disjoint q-grams even in the shorter prefix in Section 5. As the mismatch filter is orthogonal to our method, we will not discuss the details.

3.2 Comparison with State-of-the-art Filters

We compare our pivotal prefix filter with state-of-the-art q-gram-based filters. The objective of the q-gram-based filters is to prune dissimilar strings as many as possible. They require to select a set of q-grams from each of two strings as signatures, denoted as $\texttt{Sig}(r)$ and $\texttt{Sig}(s)$, and compare the two q-gram sets to check whether they share common signatures. Pruning power and filtering cost are two important issues in designing filters.

We first consider the pruning power. One one hand, the smaller the production size of the two signature sets $|\texttt{Sig}(r)| \times |\texttt{Sig}(s)|$, the smaller probability they share common q-grams, and thus the higher pruning power. On the other hand, the number of matching q-grams cannot exceed the smaller signature size of the two strings, $\min(|\texttt{Sig}(r)|, |\texttt{Sig}(s)|)$. Thus we can use the production size of two signature sets and the smaller signature set size to evaluate the pruning power. We then evaluate the filtering cost. As the q-gram sets are sorted, we can use a merge-join algorithm to find the matching q-grams if there is no index, the filtering cost depends on the sum of signature set sizes of the two strings, $|\texttt{Sig}(r)| + |\texttt{Sig}(s)|$. If a hash index is built on a signature set (Usually we build index to improve the performance as discussed in Section 4), we can use a probe-based method to check whether each q-gram in another signature set appears in the hash index, and in this case, the filtering cost depends on the size of the probing signature set. Without loss of generality, we use $\texttt{Sig}(r)$ as the probing signature set. Table 3 compares the pruning power and filtering cost of state-of-the-art q-gram-based filters used in AllPair, ED-Join, Qchunk-IndexChunk and Qchunk-IndexGram.

Given two strings r and s, the prefix filter [2] needs to select $q\tau+1$ q-grams for both of the two strings as signatures. Thus the production, the minimum, and the sum of the sizes of its two signature sets are $(q\tau+1)^2$, $q\tau+1$ and $2*(q\tau+1)$ and the size of the probing set is $q\tau+1$. The mismatch filter [24] improved the prefix filter by shortening prefix length. The prefix length is in the range of $[\tau+1, q\tau+1]$. Thus the production, the minimum and the sum of the sizes of the two signature sets are in $[(\tau+1)^2, (q\tau+1)^2]$, $[\tau+1, q\tau+1]$, and $[2*(\tau+1), 2*(q\tau+1)]$ and its probing set size is in $[\tau+1, q\tau+1]$. For Qchunk, it needs to select $\tau+1$ chunks and $l-(\lceil(l-\tau)/q\rceil-\tau)+1$ q-grams as signatures, where l is the string length. Thus the production, the minimum, and the sum of the sizes of the two signature sets are $(l-(\lceil(l-\tau)/q\rceil-\tau)+1+\tau+1)*(\tau+1)$, $\tau+1$ and $(l-(\lceil(l-\tau)/q\rceil-\tau)+1+\tau+1)+(\tau+1)$. There are two indexing methods in Qchunk, the IndexChunk and the IndexingGram, whose probing set sizes are $l-(\lceil(l-\tau)/q\rceil-\tau)+1$ and $\tau+1$ respectively. Note that the number of selected q-grams in Qchunk is rather large which depends on the string length l. Our pivotal prefix filter uses the prefix q-gram in $\texttt{pre}(r)$ and the pivotal q-grams in $\texttt{piv}(r)$ as signatures. After comparing the two q-grams $\textsf{last}(\texttt{piv}(r))$,

Table 3: Comparison with State-of-the-art Filters.

Method	$\|\mathtt{Sig}(r)\|$	$\|\mathtt{Sig}(s)\|$	Pruning Power	Filtering Cost
Preifx Filter	$q\tau+1$	$q\tau+1$	depends on	depends on
Position Mismatch Filter	$\tau+1$ to $q\tau+1$	$\tau+1$ to $q\tau+1$	$\|\mathtt{Sig}(r)\| \times \|\mathtt{Sig}(s)\|$ and	Index: $\|\mathtt{Sig}(r)\|$ (Probe set size)
Qchunk-IndexGram	$l-(\lceil\frac{l-\tau}{q}\rceil-\tau)+1$	$\tau+1$		
Qchunk-IndexChunk	$\tau+1$	$l-(\lceil\frac{l-\tau}{q}\rceil-\tau)+1$	$\min(\|\mathtt{Sig}(r)\|, \|\mathtt{Sig}(s)\|)$	No Index: $\|\mathtt{Sig}(r)\|+\|\mathtt{Sig}(s)\|$
Pivotal Prefix Filter	$\tau+1$ to $q\tau+1$	$\tau+1$		

it only requires to compare one pivotal q-gram set with one prefix q-gram set. By integrating the mismatch filter, the production, the minimum, and the sum of the sizes of the two signature sets are in $[(\tau+1)^2, (q\tau+1)*(\tau+1)]$, $\tau+1$, and $[2*(\tau+1), q\tau+1+\tau+1]$. The probing set size is in $[\tau+1, q\tau+1]$.

4. PIVOTAL PREFIX BASED FILTERING ALGORITHM

In this section, we propose the pivotal prefix based similarity search method, called PivotalSearch, which contains an offline index stage and an online query processing stage.

Indexing. Given a dataset $\mathcal{R}$, we build two inverted indexes: one for q-grams in the prefix set, denoted by $\mathcal{I}^+$; and the other for q-grams in the pivotal prefix set, denoted by $\mathcal{I}^-$[5]. We first sort the strings in $\mathcal{R}$ by their lengths. For each string $r \in \mathcal{R}$, we generate its prefix $\mathtt{pre}(r)$ and for each q-gram g with position p in $\mathtt{pre}(r)$, we insert $\langle r, p\rangle$ into inverted list $\mathcal{I}^+(g)$. Then we select the pivotal prefix $\mathtt{piv}(r)$ and for each q-gram g with position p in $\mathtt{piv}(r)$, we insert $\langle r, p\rangle$ into inverted list $\mathcal{I}^-(g)$. (We will discuss how to select pivotal prefix in Section 5.) For each q-gram g in $\mathcal{I}^-$ (and $\mathcal{I}^+$), we also build a hash map $\mathcal{M}^-$ (and $\mathcal{M}^+$), where the entries are lengths of strings in $\mathcal{I}^-(g)$ (and $\mathcal{I}^+(g)$) and the values are start positions of the corresponding strings in $\mathcal{I}^-(g)$ (and $\mathcal{I}^+(g)$). Using the hash map, we can easily get the strings on the inverted lists with a specific length (or length range) and apply the length filter to do pruning.

Example 3. *Considering the data strings in Table 1, suppose the edit-distance threshold is $\tau = 2$ and the gram length is $q = 2$, Figure 2 shows the index for the given data strings. We sort the strings in $\mathcal{R}$ and access the strings in order to build the index. For r_1, we split it into q-gram set $\mathtt{q}(r_1)$ and sort $\mathtt{q}(r_1)$ by gram frequency in ascending order. As $q\tau+1=5$, we identify the first 5 q-grams as prefix and we have $\mathtt{pre}(r_1) = \{\mathtt{im,my,te,ca,yo}\}$. We randomly select $\tau+1=3$ disjoint q-grams to obtain the pivotal q-gram set $\mathtt{piv}(r_1) = \{\mathtt{im,te,ca}\}$. Next for the 5 prefix q-grams, we insert the entries $\langle r_1,1\rangle, \langle r_1,2\rangle, \langle r_1,6\rangle, \langle r_1,8\rangle$, and $\langle r_1,3\rangle$ into $\mathcal{I}^+(\mathtt{im})$, $\mathcal{I}^+(\mathtt{my})$, $\mathcal{I}^+(\mathtt{te})$, $\mathcal{I}^+(\mathtt{ca})$, and $\mathcal{I}^+(\mathtt{yo})$ respectively. For the 3 pivotal q-grams, we insert $\langle r_1,1\rangle$, $\langle r_1,6\rangle$, and $\langle r_1,8\rangle$ into $\mathcal{I}^-(\mathtt{im})$,$\mathcal{I}^-(\mathtt{te})$, and $\mathcal{I}^-(\mathtt{ca})$ respectively. Similarly we insert r_2, r_3, r_4 and r_5 into the indexes. For each inverted list $\mathcal{I}^+(g)/\mathcal{I}^-(g)$, we build a hash map $\mathcal{M}^+/\mathcal{M}^-$. For example, for the inverted list $\mathcal{I}^+(\mathtt{im})$, we build a hash map $\mathcal{M}^+$ contains entry $\langle 9,1\rangle$ as the the start position of data strings with length 9 in $\mathcal{I}^+(\mathtt{im})$ is 1.*

The pseudo-code of our Indexing algorithm is shown in Algorithm 1. It takes as input a string dataset $\mathcal{R}$, a gram length q and an edit-distance threshold τ and outputs two inverted indexes $\mathcal{I}^+$ and $\mathcal{I}^-$. It first sorts all the strings in $\mathcal{R}$ by length, splits all the strings into q-grams, and sorts q-gram based on gram frequency (Lines 2 to 5). For each sorted q-gram set $\mathtt{q}(r)$, it generates its prefix set $\mathtt{pre}(r)$,

[5]Actually we only need to maintain one index. As the pivotal prefix is a subset of the prefix, we use a special mark to indicate whether a q-gram is a pivotal prefix.

Algorithm 1: PivotalSearch-Indexing $(\mathcal{R}, q, \tau)$

Input: $\mathcal{R}$: A String Set; q: Gram Length; τ: Threshold
Output:$\mathcal{I}^+$: Prefix Index; $\mathcal{I}^-$: Pivotal Prefix Index

```
1  begin
2      Sort strings in R by length;
3      Generate q-gram set q(r) for r ∈ R;
4      Get a global order (e.g. tf);
5      Sort q-grams in q(r) using the order;
6      for r ∈ R do
7          Generate pre(r) ;
8          for each q-gram g ∈ pre(r) do
9              Insert ⟨r, g.pos⟩ into I+(g);
10             if M+(g)[|r|]=φ then M+(g)[|r|]←|I+(g)|;
11         piv(r) =PivotsSelection(pre(r));
12         for each q-gram g ∈ piv(r) do
13             Insert ⟨r, g.pos⟩ into I−(g);
14             if M−(g)[|r|]=φ then M−(g)[|r|]←|I−(g)|;
15     return ⟨I+, I−⟩;
```

inserts them into the inverted index $\mathcal{I}^+$ and updates the hash map $\mathcal{M}^+$ (Lines 6 to 10). Next it selects $\tau+1$ pivotal q-grams from $\mathtt{pre}(r)$ to generate $\mathtt{piv}(r)$, inserts them into the inverted index $\mathcal{I}^-$ and updates the hash map $\mathcal{M}^-$ (Lines 11 to 14). Finally it returns two indexes $\mathcal{I}^-$ and $\mathcal{I}^+$ (Line 15).

Search Algorithm. Given a query string s, it outputs all the strings in $\mathcal{R}$ which are similar to the query string. It splits the query into q-grams, sorts them using the universal order, and generates the prefix set $\mathtt{pre}(s)$ and selects pivotal prefix set $\mathtt{piv}(s)$. Then for each q-gram g in $\mathtt{piv}(s)$, based on the length filtering, s is only similar to the strings with length between $|s|-\tau$ and $|s|+\tau$, thus it retrieves the start position of strings with length $|s|-\tau$ in the inverted list $\mathcal{I}^-(g)$ using the hash map $\mathcal{M}^-$, i.e., $\mathsf{start} = \mathcal{M}^-(|s|-\tau)$, and the end position, i.e., $\mathsf{end} = \mathcal{M}^-(|s|+\tau+1)-1$[6]. Then we retrieve the list $\mathcal{I}^-(g)$ and access the strings within range $[\mathsf{start}, \mathsf{end}]$, i.e., $\mathcal{I}^-(g)[\mathsf{start}, \mathsf{end}]$. For each element $\langle r, p\rangle$ in the list, if $\mathsf{last}(\mathtt{pre}(r)) > \mathsf{last}(\mathtt{pre}(s))$ and $|p - g.\mathsf{pos}| \le \tau$, we add r as a candidate.[7] Similarly, for each q-gram g in $\mathtt{pre}(s)$, we utilize $\mathcal{I}^+(g)$ and $\mathcal{M}^+$ to generate the candidates. Finally we verify the candidates.

Example 4. *Given the query s in Figure 2. We split it into q-grams, sort them by a universal order, get $\mathtt{pre}(s) = \{\mathtt{ot,om,yo,ub,co}\}$ and randomly select $\mathtt{piv}(s) = \{\mathtt{ot,om,ub}\}$. We probe the prefix q-grams index $\mathcal{I}^+$ using the pivotal q-grams in $\mathtt{piv}(s)$. As the pivotal q-gram 'ot' does not appear in the index, we get two inverted lists $\mathcal{I}^+(\mathtt{om}) = \{\langle r_2,8\rangle, \langle r_4,8\rangle\}$ and $\mathcal{I}^+(\mathtt{ub}) = \{\langle r_3,3\rangle, \langle r_5,5\rangle\}$. As $\mathsf{last}(\mathtt{pre}(r_2))$, $\mathsf{last}(\mathtt{pre}(r_3))$, $\mathsf{last}(\mathtt{pre}(r_4))$ and $\mathsf{last}(\mathtt{pre}(r_5))$ are respectively om, co,ut, and ub, which are all no larger than $\mathsf{last}(\mathtt{pre}(s)) =$co, we*

[6]If there is no such length in the hash map, we can find $\mathcal{M}^-(|s|-\tau+1)$ for the start position and $\mathcal{M}^-(|s|+\tau+2)$ for the end position.

[7]We can easily get $\mathsf{last}(\mathtt{pre}(s))$. To get $\mathsf{last}(\mathtt{pre}(r))$, we can materialize it.

Algorithm 2: PIVOTALSEARCH $(s, q, \tau, \mathcal{I}^+, \mathcal{I}^-)$

Input: s: Query; q: Gram Length; τ: Threshold; $\mathcal{I}^+$: Prefix Index; $\mathcal{I}^-$: Pivotal Prefix Index
Output: $\mathcal{A} = \{r \in \mathcal{R} \mid \mathtt{ed}(r, s) \leq \tau\}$

```
1  begin
2      Generate q-gram set q(s) and sort q-grams in q(s) ;
3      Generate pre(s);
4      Candidate set C = φ;
5      for each q-gram g ∈ pre(s) do
6          start =M+(|s| − τ); end =M+(|s| + τ + 1) − 1;
7          for i ∈ [start, end] do
8              ⟨r, p⟩ = I+(g)[i];
9              if last(pre(r))>last(pre(s)) & |p − g.pos| ≤ τ
               then Add r into C ;
10     piv(s) =PivotsSelection(pre(s));
11     for each q-gram g ∈ piv(s) do
12         start =M−(|s| − τ); end =M−(|s| + τ + 1) − 1;
13         for i ∈ [start, end] do
14             ⟨r, p⟩ = I−(g)[i];
15             if last(pre(r))≤last(pre(s)) & |p − g.pos| ≤ τ
               then Add r into C ;
16     for r ∈ C do
17         if VERIFICATION(s, r) then Add (s, r) to A;
18     return A;
```

drop all of them. Next we probe the pivotal q-grams index $\mathcal{I}^-$ using the prefix q-grams in $\mathtt{pre}(s)$. For prefix q-gram om with position 8, we probe the index and get $\mathcal{I}^-(\mathtt{om}) = \{\langle r_4, 8\rangle\}$. As $\mathsf{last}(\mathtt{pre}(r_4)) = \mathtt{ut} \leq \mathsf{last}(\mathtt{pre}(s)) = \mathtt{co}$, it can pass the pivotal prefix filter. As the position difference of the two q-grams is $8 - 8 = 0 < \tau = 2$, it can pass the position filter and we add r_4 into candidate set. We process other pivotal q-grams in the same way and finally we have three candidates r_3, r_4 and r_5. We verify them and get one result r_4.

The pseudo-code of our search algorithm is shown in Algorithm 2. It first generates the prefix set $\mathtt{pre}(s)$ (Lines 2 to 3). For each prefix q-gram g in $\mathtt{pre}(s)$, it gets the start position start and end position end of strings with lengths between $|s| - \tau$ and $|s| + \tau$ using $\mathcal{M}^+$ (Line 6). For each $i \in [\mathsf{start}, \mathsf{end}]$, it retrieves element $\langle r, p\rangle = \mathcal{I}^+(g)[i]$ (Lines 7 to 8). If $\mathsf{last}(\mathtt{pre}(r)) >= \mathsf{last}(\mathtt{pre}(s))$ and $|p - g.\mathsf{pos}| \leq \tau$, it takes r as a candidate (Line 9). Next it selects $\tau + 1$ pivotal q-grams from $\mathtt{pre}(s)$ and generates $\mathtt{piv}(s)$ (Line 10). For each pivotal q-gram g, it gets the start position start and end position end of strings within lengths between $|s| - \tau$ and $|s| + \tau$ using $\mathcal{M}^-$ (Line 12). For each $i \in [\mathsf{start}, \mathsf{end}]$, it retrieves element $\langle r, p\rangle = \mathcal{I}^-(g)[i]$ (Lines 13 to 14). If $\mathsf{last}(\mathtt{pre}(r)) < \mathsf{last}(\mathtt{pre}(s))$ and $|p - g.\mathsf{pos}| \leq \tau$, it takes r as a candidate (Line 15).

Complexity: We first analyze the space complexity. For each string $r \in \mathcal{R}$, we insert at most $q\tau + 1$ prefix q-grams into inverted index $\mathcal{I}^+$ and $\tau + 1$ pivotal q-grams into inverted index $\mathcal{I}^-$, thus the space complexity is $\mathcal{O}\big((q\tau + 1 + \tau + 1)|\mathcal{R}|\big) = \mathcal{O}\big(q\tau|\mathcal{R}|\big)$. Then we analyze the time complexity. Given a query s, we need to generate and sort the q-grams, and select the prefix. The time complexity is $\mathcal{O}(|s| + |s| \log |s| + q\tau)$. Then we probe the two inverted indexes with time complexity $\mathcal{O}\big((q\tau+1)l_s+(\tau+1)l_p\big)$ where l_s and l_p are the average inverted-list lengths of $\mathcal{I}^-$ and $\mathcal{I}^+$. As we insert $q\tau + 1$ q-grams in $\mathcal{I}^+$ and $\tau + 1$ q-grams in $\mathcal{I}^-$ for each string in $\mathcal{R}$, we can estimate $l_s = \frac{\tau+1}{q\tau+1} l_p$, thus the probing time complexity is $\mathcal{O}(2(\tau + 1)l_p) = \mathcal{O}(\tau l_p)$.

There are two challenges in our PIVOTALSEARCH method. The first one is how to select high-quality pivotal prefix. The second challenge is how to efficiently verify the candidates. We address these two challenges in Sections 5-6.

5. PIVOTAL q-gram SELECTION

We discuss how to select high-quality $\tau+1$ pivotal q-grams from the prefix q-grams. We first prove there must exist $\tau+1$ disjoint q-grams among all the prefix q-grams (Section 5.1). Then we discuss how to evaluate different pivotal prefixes (Section 5.2). Finally we devise a dynamic-programming algorithm to select the optimal pivotal prefix (Section 5.3).

5.1 Existence of Pivotal Prefix

We can prove that there must exist at least $\tau + 1$ disjoint q-grams in the prefix q-grams $\mathtt{pre}(r)$ for any string r as formalized in Lemma 3. The main reason is as follows. The prefix $\mathtt{pre}(r)$ has a requirement that it needs at least $\tau + 1$ edit operations to destroy all the q-grams in $\mathtt{pre}(r)$. Destroying all q-grams in $\mathtt{pre}(r)$ requires to apply edit operations on at least $\tau+1$ positions where the difference of any two positions is at least q, and using these $\tau + 1$ positions, we can select $\tau + 1$ disjoint q-grams.

LEMMA 3. *There must exist at least $\tau+1$ disjoint q-grams in the prefix $\mathtt{pre}(r)$ for any string r.*

PROOF. First we consider $|\mathtt{pre}(r)| = q\tau+1$. A naive way to select $\tau+1$ disjoint q-grams first sorts the $q\tau+1$ q-grams by their start positions and then partitions these $q\tau + 1$ q-grams into $\tau + 1$ groups based on the order. The last group contains the last q-gram and for $1 \leq i \leq \tau$, the i-th group contains the q-grams with orders in $[1 + q \cdot (i - 1), q \cdot i]$. The position difference between the first q-grams in each group is at least q and thus destroying them requires at least $\tau + 1$ edit operations. Thus we can take the first q-gram in each group as the pivotal q-gram.

Then we consider the prefix shortened by the mismatch filter where $|\mathtt{pre}(r)| < q\tau + 1$. The mismatch filter requires at least $\tau + 1$ edit operations to destroy all the q-grams in $\mathtt{pre}(r)$. We prove the lemma by contradiction. Suppose $\mathtt{pre}(r)$ contains less than $\tau + 1$ disjoint q-grams. We first sort the q-grams in $\mathtt{pre}(r)$ by their start positions and access these q-grams in order. We select the first q-gram as a reference q-gram. We skip all the q-grams overlapping with the reference q-gram. As the q-grams are sorted, all such q-grams contain the last character of the reference q-gram. For the first q-gram that does not overlap the reference q-gram, we select it as a new reference q-gram. We repeat these steps until all the q-grams are accessed. On one hand, as all the reference q-grams are disjoint, the number of reference q-grams is less than $\tau + 1$. On the other hand, we can apply an edit operation on the last character of each reference q-gram to destroy all the q-grams in $\mathtt{pre}(r)$. Thus we can use less than $\tau + 1$ edit operations to destroy all the q-grams in $\mathtt{pre}(r)$. This contradicts with the requirement of the mismatch filter. Thus $\mathtt{pre}(r)$ contains at least $\tau + 1$ disjoint q-grams. □

5.2 Evaluating Different Pivotal Prefixes

Although our pivotal prefix filter works for any pivotal prefixes, there may be multiple ways to select the pivotal

prefix and we want to select the best one. To this end, we need to evaluate the quality of different pivotal prefixes.

We first discuss how to select the pivotal prefix for the query string s. As shown in Algorithm 2, once we have computed the pivotal q-grams set $\texttt{piv}(s)$, we need to use them to probe the inverted index $\mathcal{I}^+$ and scan the inverted lists of q-grams in $\texttt{piv}(s)$. The longer the inverted lists we scan, the larger the filtering cost is and the smaller the pruning power is. Thus we want to select the pivotal prefix with the minimum inverted-list size. To achieve our goal, we assign each q-gram g in $\texttt{pre}(s)$ with a weight $w(g)$ which is the length of its corresponding inverted list in $\mathcal{I}^+$, i.e. $w(g) = |\mathcal{I}^+(g)|$. Our objective is to select $\tau+1$ disjoint q-grams from $\texttt{pre}(r)$ with the minimum weight. Next we formulate the optimal pivotal q-gram selection problem as below.

Definition 2 (Optimal Pivotal Prefix Selection). *Given a q-gram prefix $\texttt{pre}(s)$, the optimal pivotal q-grams selection problem is to select $\tau+1$ disjoint pivotal q-grams from $\texttt{pre}(s)$ with the minimum weight.*

Example 5. *Consider the query string s in Table 1. The weights of its 5 prefix q-grams "ot,om,yo,ub,co" are $0, 2, 3, 2, 1$ respectively. $\texttt{pre}(s)$ has 10 subsets with $\tau+1=3$ q-grams. Among them, there are four subsets with 3 disjoint q-grams. Thus there are four possible pivotal prefixes: $\{\texttt{yo,ub,co}\}$, $\{\texttt{ot,ub,co}\}$, $\{\texttt{yo,ub,om}\}$, and $\{\texttt{ot,ub,om}\}$ with weights $6, 3, 7$ and 4 respectively. The optimal pivotal prefix is $\{\texttt{ot,ub,co}\}$.*

Next we consider the data string r. Its pivotal q-grams will be inserted into the index $\mathcal{I}^-$. Given a query, we will probe $\mathcal{I}^-$ using each prefix q-gram of the query. Intuitively, if we select the low frequency prefix q-grams for the data string as pivotal q-grams, they are less likely to appear in the prefix of the query string, and this data string can be pruned by the pivotal prefix filter. Thus we can use the gram frequency as the weights of the prefix q-grams of the data string, and our objective is still to select $\tau+1$ disjoint q-grams from the prefix q-grams with the minimum weight.

Example 6. *Consider the data string r_3 in Table 1. The global gram frequency is shown on the right side of the table. The weight of its prefix q-grams "bb,ou,ut,ub,co" are $1, 3, 3, 3, 3$ respectively. There are four pivotal prefixes with 3 pivotal q-grams: $\{\texttt{ut,ub,co}\}$, $\{\texttt{ut,ub,ou}\}$, $\{\texttt{ut,bb,co}\}$, and $\{\texttt{ut,bb,ou}\}$ with weights $9, 9, 7, 7$ respectively. Both $\{\texttt{ut,bb,co}\}$ and $\{\texttt{ut,bb,ou}\}$ are optimal pivotal q-gram prefixes.*

5.3 Pivotal Prefix Selection Algorithm

To select the optimal pivotal prefix, we devise a dynamic-programming algorithm. We first sort all the prefix q-grams in $\texttt{pre}(r)$ by their start positions in ascending order and denote the k-th q-gram as g_k. For ease of presentation we use $\mathcal{W}(i,j)$ to denote the minimum weight of selecting j disjoint q-grams from the first i q-grams $g_1, g_2, \cdots, g_i$. We use $\mathcal{P}(i,j)$ to store the list of disjoint q-grams with the minimum weight. Thus we want to compute $\mathcal{P}(|\texttt{pre}(r)|, \tau+1)$.

Initially, $\mathcal{W}(i,1)$ is the minimal weight of q-grams in the first i q-grams, i.e., $\mathcal{W}(i,1) = \min_{1\le k\le i} w(g_k)$ and $\mathcal{P}(i,1) = \arg\min_{g_k} w(g_k)$. Next we discuss how to calculate $\mathcal{W}(i,j)$. We consider the possible j-th pivotal q-gram among the first i q-grams, denoted by g_k. As there should be $i-1$ q-grams before g_k, k cannot be smaller than i. Thus $k \in [j,i]$. If we select g_k as the j-th q-gram, we need to select other $j-1$ q-grams before g_k and these q-grams should not overlap with g_k. As the q-grams are sorted by positions, we only need to check whether the $(j-1)$-th q-gram overlaps with g_k. Let $g_{k'}$ denote the $(j-1)$-th q-gram that has no overlap with g_k. As there should be $i-2$ q-grams before $g_{k'}$, k' cannot be smaller than $i-1$. Thus $k' \in [i-1, k]$. It is worth noting that with the increase of k', $\mathcal{W}(k', j-1)$ monotonically decreases. Thus for each q-gram g_k, we only need to find its nearest q-gram g'_k that has no overlap with g_k. Given k, we can efficiently find is k' as follows. We first sort the prefix q-grams, we check its previous q-gram g_{k-1}. If g_{k-1} has no overlap with g_k, $k' = k-1$; otherwise we check g_{k-2}. Iteratively we can find $g_{k'}$. As $k-k'$ is at most q, the complexity to find k' for each k is $\mathcal{O}(q)$ and the total complexity for the $q\tau+1$ q-grams is $\mathcal{O}(q^2\tau)$.

Using k and k', we deduce the following recursion formula

$$\mathcal{W}(i,j) = \min_{i\le k\le j} \mathcal{W}(k', j-1) + w(g_k). \quad (1)$$

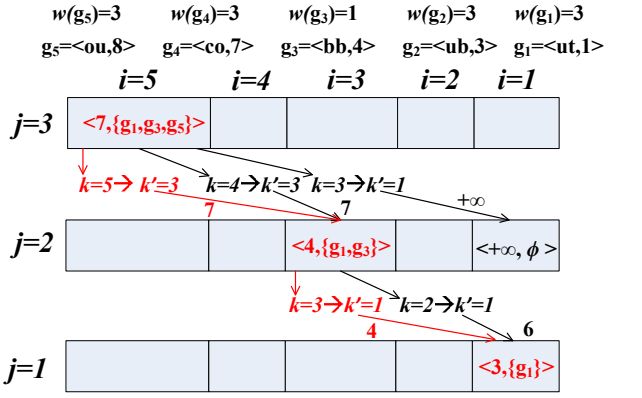
where k' depends on k. Based on Equation 1, we can find the i-th q-gram g_k and $(i-1)$-th q-gram $g_{k'}$, and $\mathcal{P}(i,j) = \mathcal{P}(k', j-1) \cup \{g_k\}$. As the size of the matrix is $\mathcal{O}(q\tau^2)$ and for each j, $k \in [i,j]$, the complexity of the dynamic-programming algorithm is $\mathcal{O}(q^2\tau^3)$.

Based on the recursion function, we develop a dynamic-programming algorithm OptimalSelection to find the optimal pivotal prefix. The pseudo-code is shown in Algorithm 3. The OptimalSelection algorithm takes as input i and j, which denote selecting j pivotal q-grams from the first i sorted prefix q-grams $g_1, g_2, \cdots, g_i$, and outputs $\mathcal{W}(i,j)$ and $\mathcal{P}(i,j)$. If $j=1$, it returns the minimum weight of the first i q-grams, and the corresponding q-gram with the min-

Algorithm 3: OptimalSelection

Input: i: First i prefix q-grams;
j: Number of pivotal q-grams to select;
Output: $\langle \mathcal{W}(i,j), \mathcal{P}(i,j)\rangle$;

```
1 begin
2   if j = 1 then return ⟨min_{1≤k≤i} w(g_k), {g_k}⟩;
3   if i < j then return ⟨+∞, φ⟩;
4   if P(i,j) ≠ φ then return ⟨W(i,j), P(i,j)⟩;
5   for k = j to i do
6     Find max k' < k s.t. g_{k'} has no overlap with g_k;
7     ⟨W(k',j−1), P(k',j−1)⟩ =OptimalSelection(k',j−1);
8     if w(g_k) + W(k',j−1) is minimum then
        return ⟨W(k',j−1)+w(g_k), P(k',j−1) ∪ {g_k}⟩ ;
```



***For i=1, j=1, as j=1* set the entry as $\langle min_{1\le k\le i} w(g_k), \{g_k\}\rangle$**

***For i=1, j=2, as i < j* set the entry as $\langle +\infty, \phi\rangle$**

Figure 3: A Runing Example for Optimal Pivotal Prefix Selection.

imum weight (Line 2). If $i < j$, as it cannot select j pivotal q-grams from $i < j$ prefix q-grams, the algorithm returns a maximum weight and an empty set (Line 3). If the two values $\mathcal{W}(i,j)$ and $\mathcal{P}(i,j)$ have already been computed, it directly returns them (Line 4). Next it selects k, k' with minimum $\mathcal{W}(k', j-1) + w(g_k)$ (Line 5 to 8). Finally, it returns $\langle \mathcal{W}(k', j-1) + w(g_k), \mathcal{P}(k', j-1) \cup \{g_k\}$(Line 8). The time complexity is $\mathcal{O}(q^2\tau^3)$.

EXAMPLE 7. *Consider the query string s and data string r_3 in Table 1. We cannot prune r_3 as $\mathsf{last}(\mathtt{pre}(r_3)) = \mathtt{co} \leq \mathsf{last}(\mathtt{pre}(s)) = \mathtt{co}$ and $\mathtt{piv}(r_3) \cap \mathtt{pre}(s) = \{\mathtt{ub}\}$. However we can use the OptimalSelection function to select the optimal pivotal q-grams. For $\mathsf{q}(r_3)$, we sort its q-grams by their positions and get $g_1 = \langle \mathtt{ut}, 1\rangle$, $g_2 = \langle \mathtt{ub}, 3\rangle$, $g_3 = \langle \mathtt{bb}, 4\rangle$, $g_4 = \langle \mathtt{co}, 7\rangle$ and $g_5 = \langle \mathtt{uu}, 8\rangle$ with weights $3, 3, 1, 3$ and 3 respectively as shown in Figure 3. Next we call algorithm OptimalSelection$(5, 3)$, which selects k from $3, 4, 5$ such that $\mathcal{W}(k', 2) + w(g_k)$ is minimum where k' equals to $3, 3, 1$ respectively. For $k = 3$, as $k' = 1 < 2$, $\mathcal{W}(k', 2) = +\infty$. We recursively call OptimalSelection$(3, 2)$ to calculate $\mathcal{W}(3, 2)$. In this round, we select k from 3 and 2 and the corresponding k' are both 1 and calculate $\mathcal{W}(1, 1)$. As $j = 1$ we return $\mathcal{W}(1, 1) = 3$ and $\mathcal{P}(1, 1) = \{g_1\}$. We find $k = 3$ achieves minimal value $\mathcal{W}(1, 1) + w(g_3) = 4$, and return $\mathcal{W}(3, 2) = 4$ and $\mathcal{P}(3, 2) = \{g_1, g_3\}$. Finally we get $\mathcal{W}(5, 3) = 7$ and $\mathcal{P}(5, 3) = \{g_1, g_3, g_5\}$. Thus we have $\mathtt{piv}(r_3) = \{\mathtt{bb}, \mathtt{ou}, \mathtt{ut}\}$. As $\mathtt{piv}(r_3) \cap \mathtt{pre}(s) = \phi$, we can prune r_3.*

6. ALIGNMENT FILTER ON PIVOTAL Q-GRAMS

The pivotal prefix filter is effective to prune the dissimilar strings with non-consecutive errors to the query since τ non-consecutive errors may exactly destroy $q\tau$ q-grams. However it is not effective for consecutive errors, because τ consecutive errors may only destroy $\tau + q - 1$ q-grams, which is much smaller than $q\tau$, and thus the pivotal prefix filter may involve many false positives. For example, consider the string r_5 and the query s in Table 1. The 2 consecutive errors on '$\mathtt{om}$' of s only destroy 3 q-grams in r_5 and we find the pivotal prefix filter cannot prune this dissimilar string. To address this problem, we propose an alignment filter to detect the consecutive errors.

Transformation Alignment. The pivotal prefix filter has two cases to generate candidates: (1) $\mathtt{pre}(r) \cap \mathtt{piv}(s) \neq \phi$; or (2) $\mathtt{pre}(s) \cap \mathtt{piv}(r) \neq \phi$. In any case there is a pivotal q-gram of a string matching a prefix q-gram of another string. Without loss of generality, we suppose a pivotal q-gram of s matching a prefix q-gram of r. In any transformation from s to r with $\mathtt{ed}(s, r)$ edit operations, each pivotal q-gram of s, e.g., the i-th pivotal q-gram $\mathtt{piv}_i(s)$, will be transformed to a substring of r, e.g., $\mathsf{sub}_i(r)$. We call $\mathtt{piv}_i(s)$ is *aligned* to $\mathsf{sub}_i(r)$. As the $\tau+1$ pivotal q-grams are disjoint, these $\tau+1$ substrings are also disjoint($\mathsf{sub}_i(r)$ may be empty). Suppose $\mathsf{err}_i = \mathtt{ed}(\mathtt{piv}_i(s), \mathsf{sub}_i(r))$. For any transformation from s to r with $\mathtt{ed}(s, r)$ operations, we have $\sum_{i=1}^{\tau+1} \mathsf{err}_i \leq \mathtt{ed}(s, r)$ as stated in Lemma 4. Figure 4 illustrates the basic idea.

LEMMA 4. *For any transformation from s to r with $\mathtt{ed}(s, r)$ operations, we have $\sum_{i=1}^{\tau+1} \mathsf{err}_i \leq \mathtt{ed}(s, r)$.*

PROOF. On one hand, we have the edit operations on the $\tau + 1$ disjoint pivotal q-grams all belong to the $\mathtt{ed}(s, r)$ edit operations in the transformation. On the other hand, the number of edit operations on each pivotal q-gram cannot be smaller than err_i. Thus we have $\sum_{i=1}^{\tau+1} \mathsf{err}_i \leq \mathtt{ed}(s, r)$. □

Based on Lemma 4, if $\sum_{i=1}^{\tau+1} \mathsf{err}_i > \tau$, s and r cannot be similar. As there may be many transformations from s to r with $\mathtt{ed}(s, r)$ operations, it is rather expensive to enumerate every transformation to compute err_i and check whether $\sum_{i=1}^{\tau+1} \mathsf{err}_i > \tau$. To address this issue, we find that for any transformation, $\mathsf{err}_i = \mathtt{ed}(\mathtt{piv}_i(s), \mathsf{sub}_i(r))$ is not larger than the minimum edit distance between the pivotal q-gram $\mathtt{piv}_i(s)$ and any substring of r, denoted by $\mathsf{sed}(\mathtt{piv}_i(s), r)$, which is called *substring edit distance* between $\mathtt{piv}_i(s)$ and r [18]. Thus we have $\mathsf{err}_i = \mathtt{ed}(\mathtt{piv}_i(s), \mathsf{sub}_i(r)) \geq \mathsf{sed}(\mathtt{piv}_i(s), r)$ and $\sum_{i=1}^{\tau+1} \mathsf{sed}(\mathtt{piv}_i(s), r) \leq \sum_{i=1}^{\tau+1} \mathsf{err}_i$.

The substring edit distance between $g = \mathtt{piv}_i(s)$ and r, i.e., $\mathsf{sed}(g, r)$, can be computed using a dynamic-programming algorithm [18]. We use a matrix $\mathcal{M}$ with $|g|+1 = q+1$ rows and $|r|+1$ columns to compute $\mathsf{sed}(g, r)$, where $\mathcal{M}[i][j]$ is the edit distance between the prefix of g with length i and some substrings of r ending with $r[j]$. Thus the substring edit distance between g and r is

$$\mathsf{sed}(g, r) = \min_{1 \leq i \leq |r|+1} \mathcal{M}[i][|g|+1].$$

To compute $\mathcal{M}[i][j]$, we first initialize $\mathcal{M}(0, j) = 0$ for $1 \leq j \leq |r|+1$ and $\mathcal{M}(i, 0) = i$ for $1 \leq i \leq q+1$. Then we compute $\mathcal{M}[i][j]$ using the following recursive function

$$\mathcal{M}(i,j) = \begin{cases} \mathcal{M}(i-1, j-1) & g[i] = r[j] \\ \min \begin{cases} \mathcal{M}(i-1, j) + 1 \\ \mathcal{M}(i, j-1) + 1 \\ \mathcal{M}(i-1, j-1) + 1 \end{cases} & g[i] \neq r[j] \end{cases}$$

The complexity of computing the substring edit distance is $\mathcal{O}(q|r|)$, and thus the filter cost is very high. To further improve the performance, we have an observation that, a pivotal q-gram $g = \mathtt{piv}_i(s)$ cannot be aligned to some substrings of r. More specifically, based on the position filter, if s and r are similar, g can only be aligned to a substring in of r between positions $g.\mathsf{pos} - \tau$ and $g.\mathsf{pos} + q - 1 + \tau]$, denoted by $r[g.\mathsf{pos} - \tau, g.\mathsf{pos} + q - 1 + \tau]$ (If $g.pos - \tau < 1$, we set the begin position as 1. If $g.pos + q - 1 + \tau > |r|$, we set the end position as $|r|$.)

Accordingly we only need to compute the substring edit distance between g and $r[g.\mathsf{pos} - \tau, g.\mathsf{pos} + q - 1 + \tau]$[8], and the complexity is $\mathcal{O}\big(q(q + 2\tau)\big) = \mathcal{O}(q^2 + q\tau)$. Moreover, we have if $|j - i| > \tau$, $\mathcal{M}(i, j) > \tau$ and we can skip these entries in the dynamic-programming algorithm, i.e. we only compute the entry $\mathcal{M}(i, j)$ where $1 \leq i \leq |q| + 1$ and $i - \tau \leq j \leq i + \tau$. Thus the time complexity decreases to $\mathcal{O}(q\tau)$ and the total time complexity to compute $\sum_{i=1}^{\tau+1} \mathsf{sed}(\mathtt{piv}_i(s), r[\mathtt{piv}_i(s).\mathsf{pos} - \tau, \mathtt{piv}_i(s).\mathsf{pos} + q - 1 + \tau])$ is $\mathcal{O}(q\tau^2)$. This is much smaller than directly computing $\mathtt{ed}(r, s)$ with time cost $\mathcal{O}(min(|r|, |s|) * \tau)$ as $min(|r|, |s|)$ is much larger than q.

Based on this observation, we design an *alignment filter*.

Alignment Filter. For any two strings r and s with a matching pivotal q-gram, if $\sum_{i=1}^{\tau+1} \mathsf{sed}(\mathtt{piv}_i(s), r[\mathtt{piv}_i(s).\mathsf{pos} - \tau, \mathtt{piv}_i(s).\mathsf{pos} + q - 1 + \tau]) > \tau$, r and s cannot be similar as formalized in Lemma 5.

LEMMA 5 (ALIGNMENT FILTER). *For any two strings r and s, if r and s are similar, we have*

$\sum_{i=1}^{\tau+1} \mathit{sed}(\mathtt{piv}_i(s), r[\mathtt{piv}_i(s).\mathit{pos} - \tau, \mathtt{piv}_i(s).\mathit{pos} + q - 1 + \tau]) \leq$

PROOF. Based on Lemma 4 and the length filter, the lemma is easy to prove. □

[8] Li et. al. [15] proposed some techniques to further reduce the substring range. Their techniques are orthogonal to ours and can be integrated into our method.

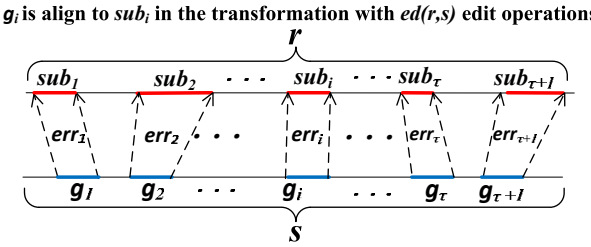


Figure 4: The Alignment Filter. $\sum_{i=1}^{\tau+1} \mathsf{err}_i \leq \mathsf{ed}(s,r)$.

Algorithm 4: VERIFICATION

Input: s: Query string; r: Data string; q: Qram length; $\mathtt{piv}(s)$: Pivotal q-grams of s; τ: Threshold;

```
1 begin
2     errors = 0;
3     for each q-gram g ∈ piv(s) do
4         errors+=sed(g, r[g.pos − τ, g.pos + q − 1 + τ], τ);
5         if errors > τ then return false;
6     if ed(r, s) ≤ τ then return true ;
7     else return false ;
```

Verification Algorithm. Next we incorporate the alignment filter into our framework and propose the VERIFICATION algorithm as shown in Algorithm 4. For any data string r and a query string s with a matching pivotal q-gram g, we first perform the alignment filter by checking whether $\sum_{i=1}^{\tau+1} \mathsf{sed}(\mathtt{piv}_i(s), r[\mathtt{piv}_i(s).\mathsf{pos}-\tau, \mathtt{piv}_i(s).\mathsf{pos}+q-1+\tau]) > \tau$. If yes, the pair will be pruned by the filter; otherwise we verify it by calculating their real edit distance using the dynamic-programming algorithm.

For example, consider the record r_5 and query string s in Table 1. As $\mathsf{last}(\mathtt{pre}(r_5)) \leq \mathsf{last}(\mathtt{pre}(s))$ and $\mathtt{piv}(r_5) \cap \mathtt{pre}(s) = \mathtt{yo}$, r_5 is a candidate and we use Algorithm 4 to verify it. For each of the three pivotal q-grams, we calculate their substring edit distance $\mathsf{sed}(\mathtt{ot}, r_5[2-2, 2+2-1+2] = \mathit{yoytu}) = 1$, $\mathsf{sed}(\mathtt{om}, r_5[8-2, 8+2-1+2] = \mathit{beca}) = 2$. As the total number of errors (3) is already larger than the threshold (2), we can prune r_5.

Comparison with Content Filter. ED-Join [24] proposed a content-based mismatch filter to detect the consecutive errors. Its time complexity is $\mathcal{O}(\tau|\Sigma|+l)$ where Σ is the symbol table for the dataset and l is the string length. Obviously the complexity of content filter is too high while our alignment filter has a low time complexity of $\mathcal{O}(q\tau^2)$, which is independent of the string length and symbol table size.

7. SUPPORTING DYNAMIC THRESHOLDS

In this section we discuss how to support dynamic edit-distance thresholds. Along the same line as existing works `Qchunk`, `AdaptJoin`, `Flamingo` and $\mathtt{B}^{ed}\mathtt{tree}$, we also assume there is a maximum edit-distance threshold $\hat{\tau}$ for the query. This is because, if $|s| - q + 1 - q\tau \leq 0$, all strings will be taken as a candidate in the q-gram-based methods. To make q-gram-based method have pruning power, the threshold cannot exceed $\lfloor \frac{|s|-q}{q} \rfloor$. A naive way to support dynamic edit-distance thresholds is to build $\hat{\tau}+1$ indexes for threshold $\tau \in [0, \hat{\tau}]$. For each query with a threshold τ, we use the corresponding index to answer the query. However this method involves large space overhead. To address this issue, we build incremental indexes $\mathcal{I}_\tau^+$ and $\mathcal{I}_\tau^-$ for each $0 \leq \tau \leq \hat{\tau}$.

Indexing. Consider a dataset $\mathcal{R}$ and a maximum edit-distance threshold $\hat{\tau}$. We first split the strings into q-grams and get a universal order. For each string $r \in \mathcal{R}$, we sort its q-grams based on the universal oder. We access the q-grams in order and select the maximum q-gram such that destroying the q-gram requires at least one edit operation, denoted by g_0. Obviously g_0 is the first q-gram in $\mathtt{q}(r)$. Next we select the maximal q-gram g_1 such that destroying all q-grams before g_1 (including g_1) requires at least two edit operations. Similarly, we generate $g_2, \cdots, g_{\hat{\tau}}$[9]. Obviously $\{g_1, g_2, \cdots, g_i\}$ is a pivotal prefix of r for threshold i and the set of q-grams before g_i is a prefix set of r for threshold i. We insert g_i into $\mathcal{I}_i^-$ and the q-grams between g_{i-1} and g_i (including g_i and excluding g_{i-1}) into $\mathcal{I}_i^+$ for $1 \leq i \leq \hat{\tau}$.[10]

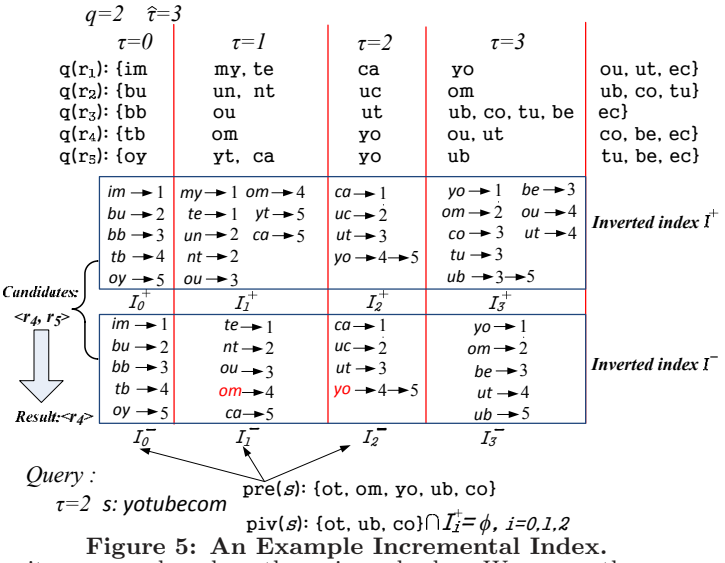


Figure 5: An Example Incremental Index.

EXAMPLE 8. *Consider the data strings in Table 1. Suppose the maximum edit distance threshold is* $\hat{\tau} = 3$. *We build four incremental indexes as shown in Figure 5. We take* r_4 *as an example. First we get* $\mathtt{q}(r_4) = \{\langle \mathtt{tb}, 4\rangle, \langle \mathtt{om}, 8\rangle, \langle \mathtt{yo}, 1\rangle, \langle \mathtt{ou}, 2\rangle, \langle \mathtt{ut}, 3\rangle, \langle \mathtt{co}, 7\rangle, \langle \mathtt{be}, 5\rangle, \langle \mathtt{ec}, 6\rangle\}$. *We have* $g_0 = \mathtt{tb}, g_1 = \mathtt{om}, g_2 = \mathtt{yo}, g_3 = \mathtt{ut}$. *Especially,* $g_3 = \mathtt{ut}$ *because destroying the q-grams before* g_3 *requires at most 4 edit operations and destroying the q-grams before the previous q-gram of* g_3 *(i.e.,* `ou`*) requires at most 3 edit operations. Then we insert* `tb`, `om`, `yo`, `ut` *respectively into* $\mathcal{I}_0^-, \mathcal{I}_1^-, \mathcal{I}_2^-, \mathcal{I}_3^-$, *and* $\{\mathtt{tb}\}, \{\mathtt{om}\}, \{\mathtt{yo}\}, \{\mathtt{ou}, \mathtt{ut}\}$ *respectively into* $\mathcal{I}_0^+, \mathcal{I}_1^+, \mathcal{I}_2^+, \mathcal{I}_3^+$.

Search Algorithm. The search algorithm is the same as that in Section 4, except that (1) To select the optimal pivotal prefix, for each q-gram g in the query string s, its token weight is $\sum_{i=0}^{\tau} |\mathcal{I}_i^+(g)|$; and (2) For each q-gram in prefix $\mathtt{pre}(s)$ we use indexes $\mathcal{I}_i^-$ and for each q-gram in pivotal prefix $\mathtt{piv}(s)$ we use indexes $\mathcal{I}_i^+$ for $0 \leq i \leq \tau$. For example, consider the query string s in Table 1 with threshold $\tau = 2$. We compute $\mathtt{pre}(s)$ and $\mathtt{piv}(s)$ as shown in Figure 5. The weights of the five prefix q-grams in $\mathtt{pre}(s)$ are $0, 1, 2, 0, 0$ respectively. We probe the incremental inverted indexes and get two candidates r_4 and r_5. The alignment filter can prune the candidate r_5 and finally we get one result r_5.

8. EXPERIMENT

We conducted experiments to evaluate the performance and scalability of our method. We compared with four state-of-the-art studies, `Flamingo`, `AdaptJoin`, `Qchunk` and $\mathtt{B}^{ed}\mathtt{tree}$. We obtained the source codes of `AdaptJoin` and $\mathtt{B}^{ed}\mathtt{tree}$ from the authors, downloaded the source code of `Flamingo` from "Flamingo" project homepage[11] and imple-

[9] ED-Join [24] proposed a solution to get the maximum q-gram g_i which which requires at least $i+1$ edit operations to destroy all q-grams before g_i

[10] We also insert g_0 into $\mathcal{I}_0^+$.

[11] `http://flamingo.ics.uci.edu/`

Table 4: Datasets.

Datasets	Cardinality	Avg Len	Max Len	Min Len
Title	4,000,000	100.6	386	54
Title Query	4,000	100.08	307	54
DNA	2,476,276	108.0	108	108
DNA Query	2,400	100.08	108	108
URL	1,000,000	28.03	193	20
URL Query	1,000	28.07	68	20

mented `Qchunk` by ourselves. All the algorithms were implemented in C++ and compiled using g++ 4.8.2 with -O3 flags. All the experiments were conducted on a machine running on 64bit Ubuntu Server 12.04 LTS version with an Intel Xeon E5-2650 2.00GHz processor and 16GB memory.

Datasets: We used three real datasets PubMed Title[12], DNA[13], and URL[14] in the experiments. The PubMed Title dataset is a medical publication title dataset. We selected 4,000,000 titles in the dataset and randomly chose 4,000 titles as query strings. The DNA dataset contained 2,476,276 DNA reads and we randomly selected 2400 reads as query strings. The URL dataset is a set of hyperlinks. We used 1,000,000 entries as dataset and 1000 entries as query strings. The details of the dataset are shown in Table 4. We tuned the parameter q to achieve the best performance and in the experiments we used $q = 8, 8, 6, 6, 4, 4$ for $\tau = 2, 4, 6, 8, 10, 12$ on the PubMed Title dataset, $q = 12, 12, 12, 10, 9, 8$ for $\tau = 2, 4, 6, 8, 10, 12$ on the DNA dataset, and $q = 6, 3, 3, 2, 2$ for $\tau = 1, 2, 3, 4, 5$ on the URL dataset for all algorithms.

8.1 Evaluation on the pivotal q-gram techniques

In this section, we evaluated the effectiveness and efficiency of our proposed filters and the optimal pivotal q-gram selection techniques. We implemented five methods. (1) `CrossFilter` only utilized the cross prefix filter (based on Lemma 1). (2) `PivotalFilter` only adopted the pivotal prefix filter (based on Lemma 2). (3) `CrossSelect` utilized the cross prefix filter and the optimal selection algorithm to select the pivotal q-grams. (4) `PivotalSelect` used the pivotal prefix filter and the optimal selection algorithm to select pivotal q-grams. (5) `Mismatch` was the state-of-the-art mismatch filter [24]. In the verification step, we used the dynamic programming algorithm (with the optimization of verifying $\tau + 1$ cells in each row [15]) to verify candidates for all of these methods. We tested their candidate numbers and average search time on the three datasets. The results are shown in Figures 6-7.

From Figure 6, we can see `CrossFilter` was better than `Mismatch`. `CrossSelect` had smaller numbers of candidates than `CrossFilter`. `PivotalFilter` and `PivotalSelect` further reduced the candidate numbers. For example, on the DNA dataset under the edit-distance threshold $\tau = 8$, `Mismatch` had 9 million candidates while `CrossFilter` only had 3 million candidates. `CrossSelect` further reduced the candidate number to 2 million. `PivotalSelect` had the minimum number of candidates, 1 million. This is because the optimal pivotal q-gram selection technique selected the pivotal q-grams with the minimum inverted-list sizes and thus reduced candidate numbers, and the cross prefix filter and the pivotal prefix filter removed unnecessary q-grams from the prefix and thus reduced the candidate number. By combining them together, `PivotalSelect` further reduced the candidate number.

We also tested the average search time of the five methods. Figure 7 shows the results. We can see `CrossFilter` had smaller average search time than `Mismatch` while `CrossSelect` and `PivotalFilter` outperformed `CrossFilter`. `PivotalSelect` achieved the best performance. For example, on the PubMed Title dataset with edit-distance threshold $\tau = 12$, the average search time for `Mismatch` and `CrossFilter` were 82.5 milliseconds and 60 milliseconds, while the average search time for `CrossSelect` and `PivotalFilter` were 45 milliseconds and 50 milliseconds. `PivotalSelect` further reduced the average search time to 25 milliseconds. This is because `PivotalSelect` not only reduced large numbers of candidates but also improved the filtering cost.

8.2 Evaluation on the alignment filter

In this section we evaluated the alignment filter and compared with content filter [24]. We implemented three methods: (1) `NoFilter` utilized the dynamic programming algorithm to verify candidates as discussed in Section 8.1, (2) `AlignFilter` first used the alignment filter and then utilized the `NoFilter` algorithm to verify candidates, (3) `ContentFilter` first used the content filter and then utilized the `NoFilter` algorithm to verify candidates. We used `PivotalSelect` to generate candidates. We compared them on candidate numbers and average verification time. Figures 8-9 show the results.

The Real bar in Figure 8 indicates the number of answers. Compared with `NoFilter`, `ContentFilter` reduced the number of candidates by 1-2 orders of magnitude. Moreover, `AlignFilter` significantly outperformed `ContentFilter` by 2-4 orders of magnitude and the candidate number of `AlignFilter` was close to the number of real results. For example, on the PubMed Title dataset with edit-distance threshold $\tau = 12$, `NoFilter` had 54 million candidates and `ContentFilter` had 3 million candidates. `AlignFilter` further reduced the number to 5,500 which is very close to the number of real answers 4,400. This is because the alignment filter can effectively detect the consecutive edit errors on the low-frequency pivotal q-grams to prune large numbers of dissimilar strings, while `ContentFilter` only considered the difference of the number of characters appeared between query and candidate strings.

We compared the average verification time and Figure 9 shows the result. We can see from the figure that `AlignFilter` outperformed `NoFilter` by 2-4 times and `ContentFilter` by 1-2 times. For example, on the DNA dataset with edit-distance threshold $\tau = 12$, `AlignFilter` took 10 milliseconds while `NoFilter` took 72 milliseconds and `ContentFilter` took 40 milliseconds. This is because the alignment filter detected the errors on the mismatched pivotal q-gram instead of the whole strings which saved a lot of verification time, while `ContentFilter` scanned the whole strings.

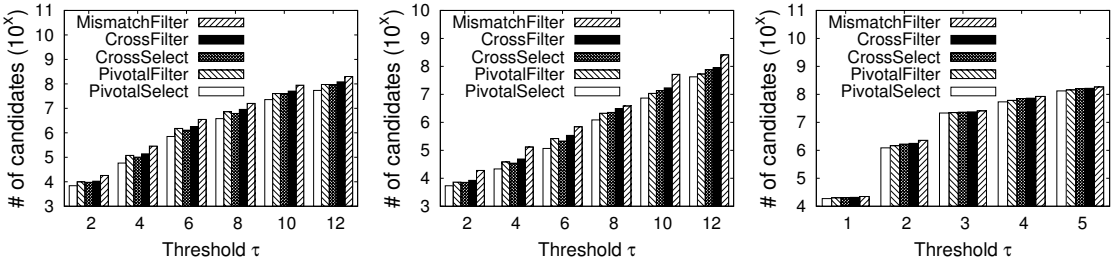
8.3 Comparison of state-of-the-art methods

In this section, we compared our method `PivotalSearch` (which utilized the `PivotalSelect` algorithm in the filtering step and the `AlignFilter` algorithm in the verification step) with state-of-the-art methods `Qchunk`, `Flamingo`, and `AdaptJoin` on the three datasets. We evaluated the average search time and candidate numbers. The results are shown in Figure 10. Each bar denotes the average search time, including the filtering time (the lower bar) that contained the

[12] http://www.ncbi.nlm.nih.gov/pubmed

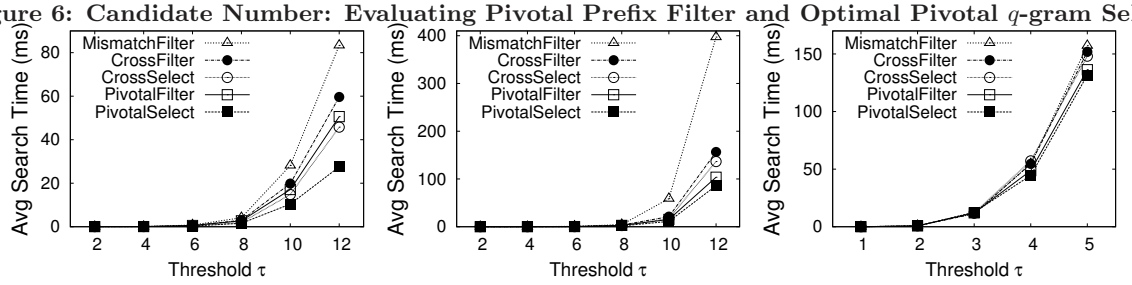
[13] http://www.ncbi.nlm.nih.gov/genome

[14] http://www.sogou.com/labs/dl/t-rank.html



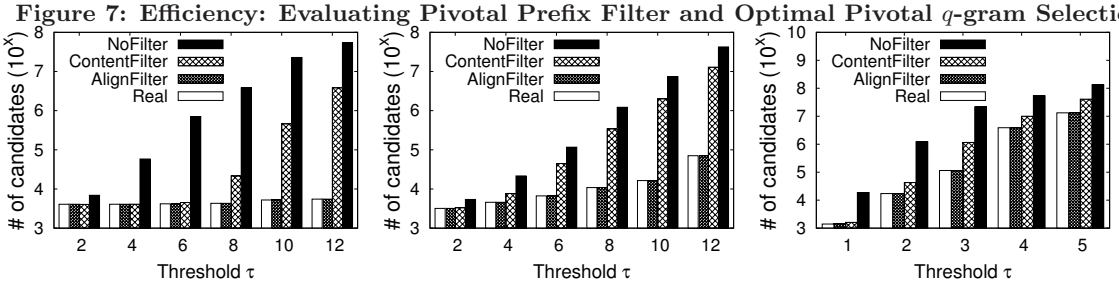
(a) Title (Avg Len = 100) (b) DNA Log (Avg Len = 108) (c) URL (Avg Len = 28)

Figure 6: Candidate Number: Evaluating Pivotal Prefix Filter and Optimal Pivotal q-gram Selection.



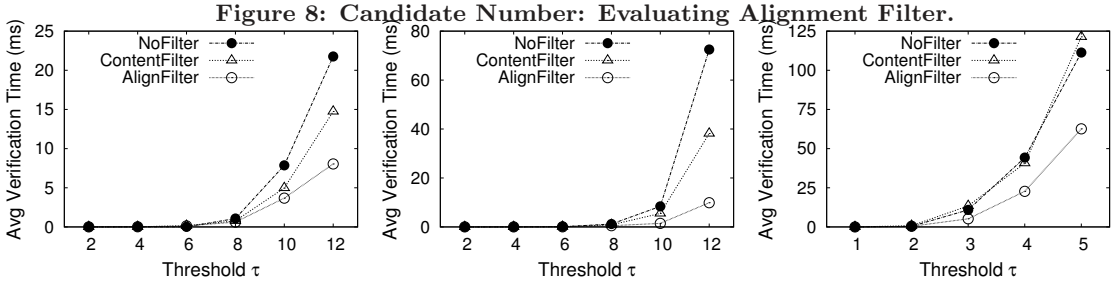
(a) Title (Avg Len = 100) (b) DNA Log (Avg Len = 108) (c) URL (Avg Len = 28)

Figure 7: Efficiency: Evaluating Pivotal Prefix Filter and Optimal Pivotal q-gram Selection.



(a) Title (Avg Len = 100) (b) DNA (Avg Len = 108) (c) URL (Avg Len = 28)

Figure 8: Candidate Number: Evaluating Alignment Filter.



(a) Title (Avg Len = 100) (b) DNA (Avg Len = 108) (c) URL (Avg Len = 28)

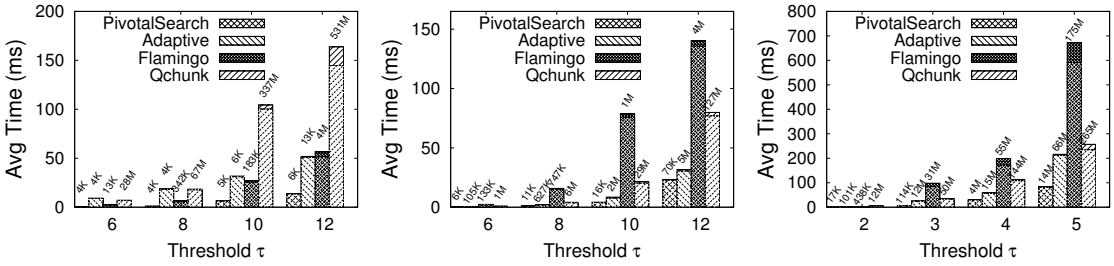
Figure 9: Efficiency: Evaluating Alignment Filter.

time for all filters (e.g., `PivotalFilter` and `AlignFilter` for our method) and verification time (the upper bar) that only contained the time for computing real edit distance. The numbers on top of the bars denote the candidate numbers. For average search time, `PivotalSearch` achieved the best performance on all datasets, and outperformed existing algorithms by 2-10 times. For example on the URL dataset with edit-distance threshold $\tau = 5$, the average search time for `AdaptJoin`, `Qchunk`, `Flamingo` and `PivotalSearch` were 215 milliseconds, 256 milliseconds, 671 milliseconds and 82 milliseconds respectively. This is because our pivotal prefix filter reduced the number of q-grams, the optimal pivotal q-gram selection algorithm selected high-quality q-grams and the alignment filter detected consecutive edit errors.

For the filtering step, our method still achieved the best performance. For example, on the DNA dataset with edit-distance threshold $\tau = 12$, the average filtering time of `PivotalSearch`, `AdaptJoin`, `Flamingo` and `Qchunk` were 25 milliseconds, 30 milliseconds, 135 milliseconds, and 80 milliseconds respectively. This is because the filtering cost of `PivotalSearch` is smaller than existing methods as explained in Table 3. For the verification step, our method also outperformed existing ones (discussed in Section 8.2).

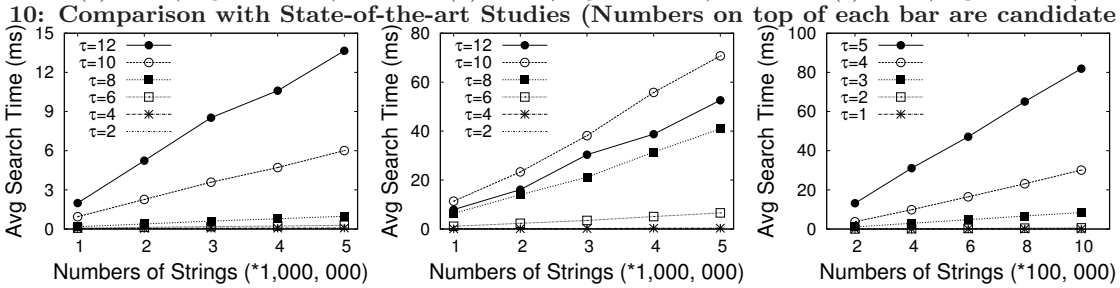
For the candidate numbers, `PivotalSearch` always generated the least number of candidates, which is 1-2 orders of magnitudes less than other methods. For example, In the URL dataset with edit-distance threshold $\tau = 3$, our `PivotalSearch` method generated 110 thousand candidates while `AdaptJoin`, `Flamingo`, and `Qchunk` generated 12 million, 31 million, and 50 million candidates. This is because our pivotal prefix filter and alignment filter are extremely effective for non-consecutive errors and consecutive errors which can filter most of dissimilar strings.

We also compared the index sizes. For example, on the PubMed Title dataset under threshold $\tau = 10 (q = 6)$, the index sizes for `Qchunk`, `AdaptJoin`, `Flamingo` and `PivotalSearch` were respectively 310MB, 260MB, 400MB, 224MB.



(a) Title (Avg Len = 100) (b) DNA (Avg Len = 108) (c) URL (Avg Len = 28)

Figure 10: Comparison with State-of-the-art Studies (Numbers on top of each bar are candidate numbers).



(a) Title (Avg Len = 100) (b) DNA (Avg Len = 108) (c) URL (Avg Len = 28)

Figure 11: Scalability.

8.4 Scalability

In this section we evaluated the scalability of our method. We used the same queries and varied the dataset sizes. The results are shown in Figure 11. We can see that our method scaled very well on the three datasets. For example, on the PubMed Title dataset with edit-distance threshold $\tau = 10$, we varied the dataset sizes from 1 million to 5 million. The search time were respectively 1 milliseconds, 2.3 milliseconds, 3.5 milliseconds, 4.8 milliseconds, and 6 milliseconds. This is also attributed our effective filters which can prune more dissimilar strings on larger datasets.

9. CONCLUSION

We studied the string similarity search problem with edit-distance thresholds. We proposed a pivotal prefix filter which can significantly reduce the number of signatures. We proved that the pivotal prefix filter outperforms state-of-the-art filters. We devised a dynamic-programming algorithm to select the optimal pivotal prefix. To detect the consecutive errors, we proposed an alignment filter to prune large numbers of dissimilar strings with consecutive errors. Experimental results on real datasets show our method significantly outperformed state-of-the-art studies.

Acknowledgements. This work was partly supported by NSF of China (61272090 and 61373024), 973 Program of China (2011CB302206), Beijing Higher Education Young Elite Teacher Project (YETP0105), Tsinghua-Tencent Joint Laboratory, "NExT Research Center" funded by MDA, Singapore (WBS:R-252-300-001-490), and FDCT/106/2012/A3.

10. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[3] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

[5] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, pages 707–718, 2009.

[6] D. Deng, G. Li, J. Feng, and W.-S. Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013.

[7] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.

[8] 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 *VLDB*, pages 491–500, 2001.

[9] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD Conference*, pages 429–440, 2009.

[10] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 433–439, 2009.

[11] Y. Jiang, G. Li, and J. Feng. String similarity joins: An experimental evaluation. *PVLDB*, 2014.

[12] Y. Kim and K. Shim. Efficient top-k algorithms for approximate substring matching. In *SIGMOD Conference*, pages 385–396, 2013.

[13] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[14] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[15] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[16] G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20(4):617–640, 2011.

[17] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD Conference*, pages 1033–1044, 2011.

[18] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.

[19] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

[20] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[21] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.

[22] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. *IEEE Trans. Knowl. Data Eng.*, 25(8):1916–1929, 2013.

[23] C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane. Efficient error-tolerant query autocompletion. *PVLDB*, 6(6):373–384, 2013.

[24] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[25] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[26] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, pages 915–926, 2010.