VChunkJoin: An Efficient Algorithm for Edit Similarity Joins

Wei Wang[1] Jianbin Qin[1] Chuan Xiao[1] Xuemin Lin[1]
Heng Tao Shen[2]

[1] University of New South Wales, Australia
{weiw,jqin,chuanx,lxue}@cse.unsw.edu.au

[2] University of Queensland, Australia
shenht@itee.uq.edu.au

Technical Report
UNSW-CSE-TR-1114
Augest 2011

Abstract

Similarity joins play an important role in many application areas, such as data integration and cleaning, record linkage, and pattern recognition. In this paper, we study efficient algorithms for similarity joins with an edit distance constraint. Currently, the most prevalent approach is based on extracting *overlapping* grams from strings and considering only strings that share a certain number of grams as candidates. Unlike these existing approaches, we propose a novel approach to edit similarity join based on extracting *non-overlapping* substrings, or chunks, from strings. We propose a class of chunking schemes based on the notion of tail-restricted chunk boundary dictionary. A new algorithm, *VChunkJoin*, is designed by integrating existing filtering methods and several new filters unique to our chunk-based method. We also design a greedy algorithm to automatically select a good chunking scheme for a given dataset. We demonstrate experimentally that the new algorithm is faster than alternative methods yet occupies less space.

1 Introduction

A similarity join finds pairs of objects from two datasets such that the pair's similarity value is no less than a given threshold. Similarity joins have been widely used in important application areas such as data integration, record linkage, and pattern recognition. While early research in this area focuses on similarities defined in the Euclidean space, current research has spread to various similarity (or distance) functions, including set overlap and containment [24, 20], Jaccard similarity [10, 34], cosine similarity [3], edit distance [9, 2, 33], and other more sophisticated functions [5, 1].

In this work, we focus on *edit similarity joins*, i.e., similarity joins with an edit distance constraint of no more than a constant τ [2]. The major technical difficulty lies in the fact that the edit distance metric is complex and costly to compute. In order to scale to large datasets, all existing algorithms adopt the *filter-and-verify* paradigm, which first eliminates non-promising string pairs with relatively inexpensive computation and then performs verification by the edit distance computation. By far the most prevalent filtering approach is those based on *grams*; they can be further categorized into methods using fixed-length q-grams [9, 33] and recent proposals using variable-length grams (VGRAM) [18, 35].

Gram-based methods extract all substrings of certain length (q-grams), or according to a dictionary (VGRAMs), for each string. Therefore, subsequent grams are overlapping and are highly redundant. This results in large index size and incurs high query processing costs when the index cannot be entirely accommodated in the main memory. For example, for q-gram-based edit similarity join algorithms [9, 33], the total size of the q-grams will be about *six* times as large as the size of the text strings. Researchers tackle this issue by omitting some of the grams when building the index. Depending on the techniques used, discarding some grams may [21, 27, 13] or may not [26, 4] miss some similar pairs.

In this paper, we study a novel approach to edit similarity joins based on the idea of chunking. Our work starts with the observation that the main redundancy in gram-based approaches lies in the existence of overlapping grams. We asked the simple question: *why can't we just keep all the* non-overlapping *grams and perform edit similarity joins?* Intuitively, for each string, we divide it into several disjoint substrings, or *chunks*, and we only need to process and index these chunks. This will deliver a massive improvement of space usage: for each string of length l, we only need to use $4l/avg\_chunk\_len$ bytes to store the hashed representation of the chunks rather than $4l$ bytes for gram-based methods.

However, this simple idea does *not* work in such simple form, as a single edit operation may destroy *all* the chunks (we call it the *avalanching effect*), hence there is no way we can identify similar strings in such scenario.

Example 1. *Consider the naïve fixed-length chunking algorithm where each chunk has length 2 (i.e., essentially non-overlapping version of bi-grams). Consider the following two strings and their resulting chunks (underlined).*

$$\underline{\texttt{ab}}\quad\underline{\texttt{cd}}\quad\underline{\texttt{ef}}\quad\underline{\texttt{gh}}$$

$$\underline{\texttt{bc}}\quad\underline{\texttt{de}}\quad\underline{\texttt{fg}}\quad\underline{\texttt{h}}$$

Although the two strings have edit distance of 1, they do not *share any chunk.*

This apparent difficulty probably contributes to fact that, to our best knowledge, there is *no* chunking-based approach for *exact* approximate string matching or join with edit distance constraints.

In this paper, we show that there exist so-called *robust chunking schemes* that have the salient property that each edit operation destroys at most two chunks. We give a sufficient condition to construct a particular sub-class of robust chunking schemes based on the notion of *tail-restricted chunk boundary dictionaries*. We then propose a new edit similarity join algorithm, named VChunkJoin. Thanks to the property of the robust chunking scheme, there is a tight lower bound, $LB_{s;t}(\tau)$, on the number of common chunks shared by two strings s and t if their edit distance is no more than τ. After applying the prefix filtering and location-based mismatch filtering methods, we have the following main filtering condition: we only need to index l chunks ($\tau + 1 \leq l \leq 2\tau + 1$) for a string s such that any string with no more than τ edit distance from s will share at least one chunk. Hence the number of signatures generated by our algorithm is *guaranteed* to be fewer than or equal to that of the q-grams-based method [3], since $q \geq 2$. Another feature of our chunk-based method is that it can use all the existing filtering methods (length, count and position, prefix, location-based mismatching, content-based mismatching filterings) as well as new filtering methods unique to our chunk-based method, including *rank*, *chunk number* and *virtual CBD* filterings. We also consider the problem of selecting a good chunking scheme for a given dataset. Although the problem is NP-hard under some simplifying assumptions, we design an efficient greedy algorithm which achieves good results in practice. We have conducted extensive experiments using several real datasets. One interesting finding is that *our proposed algorithm occupies less space (both in memory and on the disk) than alternative methods yet it is still faster.*

Our contributions can be summarized as:

- We are the first to introduce chunk-based method for the exact edit similarity join. We devise a class of robust chunking scheme that has a quite tight lower bound on the number of chunks shared by similar strings. Although we focus on the edit similarity join in the paper, the proposed chunking method is equally useful to the approximate string matching under the edit distance metric.
- We design an efficient edit similarity join algorithm named VChunkJoin. The algorithm leverages several new, powerful filters in addition to existing ones.
- We devise an efficient CBD selection algorithm to find a good chunking scheme that facilitates the edit similarity join.
- We perform extensive experiments on several real datasets. Our proposed algorithm is shown to outperform several existing algorithms in both speed and space usage.

The rest of the paper is organized as follows: Section 2 gives the problem definition and introduces necessary backgrounds. We describe our chunking scheme in Section 3 and how to use it for edit similarity join in Section 4. We show the hardness of finding the optimal CBD and give an algorithm to find a good one in Section 5. Experimental results are presented and analyzed in Section 6. Section 7 surveys related work. Section 8 concludes the paper.

2 Problem Definition and Preliminaries

2.1 Problem Definition

Let Σ be a finite alphabet of symbols; each symbol is also called a *character*. A string s is an ordered array of symbols drawn from Σ. We use $s[i\,..\,j]$ to denote the substring of s starting from the i-th character and ending at the j-th character. All subscripts start from 1. The length of string s is denoted as $|s|$. Each string s is also assigned an identifier $s.id$. All input string sets are assumed to be in the increasing order of string length. $ed(s,t)$ denotes the edit distance between strings s and t, which measures the minimum number of edit operations (insertion, deletion, and substitution) to transform s to t (and vice versa). In practice, it can be computed in $O(|s||t|)$ time and $O(\min(|s|,|t|))$ space using the standard dynamic programming [30].

Given two sets of strings R and S, a similarity join with edit distance constraints or *edit similarity join* [6] returns pairs of strings from R and S, such that their edit distance is no larger than a given threshold τ, i.e., $\{\,\langle r,s\rangle \mid ed(r,s)\le\tau, r\in R, s\in S\,\}$. For ease of exposition, we will focus on self joins in this paper, i.e., $\{\,\langle r_i,r_j\rangle \mid ed(r_i,r_j)\le\tau\wedge r_i.id<r_j.id, r_i\in R, r_j\in R\,\}$.

2.2 Previous Approaches

A widely used method for answering edit similarity joins is to relax the edit distance constraint to a weaker constraint on the number of *matching q-grams* . A q-gram is a contiguous substring of length q. Given a string s, we move a sliding window of width q over s to extract the q-grams of the string. The starting position of each q-gram in s is called *position* [9]. A *positional* q-gram (or simply q-gram if no ambiguity) is a q-gram together with its position, represented in the form of *(qgram, pos)* [9]. Two q-grams are called *matching* if they have the same content and their positions are within the edit distance threshold τ. If two strings s and t are within edit distance τ, they must satisfy the following two conditions [9]:

- **Count + Position Filtering**: s and t must share at least $LB_{s;t}(\tau)=(\max(|s|,|t|)-q+1)-q\tau$ *matching* q-grams.

- **Length Filtering**: $||s| - |t|| \le \tau$.

Let w be a q-gram and $df(w)$ indicates the number of strings that contain w. The *inverse document frequency* of w, $idf(w)$, is defined as $1/df(w)$.

We sort the q-grams in a string by decreasing order of their *idf* values and increasing order of their positions. We call the sorted array the *q-gram array* of the string. Given a q-gram array x, $str(x)$ denotes the corresponding string. The i-th positional q-gram is captured by $x[i]$, where $x[i].token$ denotes the the q-gram and $x[i].pos$ denotes its position (in the original string). The k-prefix of x is its first k entries, i.e., $x[1\,..\,k]$.

An inverted index maps a q-gram w to an array I_w of entries in the form of (id, pos), where id is the identifier of the string that contains w, and pos is the position of w in the string identified by id.

Several existing approaches employ the prefix filtering technique [6, 33] to quickly filter out the candidate pairs that are guaranteed not to satisfy the count filtering condition. The intuition is that if two strings meet the $LB_{s;t}(\tau)$ threshold, they should share at least one matching q-gram if we look into part of their q-grams. We formally state the prefix filtering principle for edit similarity joins in Lemma 1.

Lemma 1. *Consider two q-gram arrays x and y. If $ed(str(x), str(y)) \le \tau$, the $(q\tau+1)$-prefix of x and the $(q\tau+1)$-prefix of y must have at least one matching q-gram.*

We consider a prefix-filtering-based similarity join algorithm, All-Pairs-Ed [33], for edit similarity join (See Algorithm 1). This algorithm is chosen to make it easier to understand the state-of-the-art edit similarity join algorithm, Ed-Join and our proposed VChunkJoin algorithm (Algorithm 2).

The input for All-Pairs-Ed algorithm is a set of q-gram arrays, sorted in increasing order of their lengths. It iterates through each q-gram array x, and builds an in-memory inverted index on the q-grams on-the-fly. For each q-gram w in the $(q\tau + 1)$-prefix of x, it probes the inverted index to find candidate q-gram arrays y that contain a matching q-gram to w. x and all of its candidates will be further checked by the Verify algorithm.

Algorithm 1: All-Pairs-Ed (R, τ)

```
Data: $x$ and $y$ are $q$-gram arrays that generated from strings; $x.strlen$ is the length of the string that $x$ is
      generated from.
1  $S \leftarrow \emptyset$;
2  Initialize the inverted index $I$;
3  for each $q$-gram array $x \in R$ do
4      $A \leftarrow$ empty map from id to boolean;
5      $p_x \leftarrow q\tau + 1$ ;                    /* can be improved by location-based mismatch filtering */
6      for $i = 1$ to $p_x$ do
7          $w \leftarrow x[i].token$;   $pos_x \leftarrow x[i].pos$;
8          for each $(y, pos_y) \in I_w$ such that $y.strlen \ge x.strlen - \tau$ and $A[y]$ has not been initialized do
9              if $|pos_x - pos_y| \le \tau$ then
10                 $A[y] \leftarrow$ true ;                                      /* $y$ is a candidate */
11
12         $I_w \leftarrow I_w \cup \{ (x, pos_x) \}$ ;                      /* index the current $q$-gram $w$ */
13     Verify$(x, A)$;
14 return $S$
```

In the Verify algorithm, count and position filterings are applied to each candidate pair. Those that pass both filters will be further checked by calculating their edit distance.

The current state-of-the-art Ed-Join algorithm [33] improves the All-Pairs-Ed algorithm mainly in the following aspects:

- The q-gram prefix length, p_x, is reduced such that at least $\tau + 1$ edit operations are needed to destroy all the q-grams in the shortened prefix. (Line 5 in Algorithm 1)
- Improved verification algorithm that uses L_1 distance based filtering.

An interesting alternative is proposed recently in [18, 35]. The idea is to use variable-length grams, and it aims at striking a balance between rare and frequent tokens. Compared to the traditional q-gram-based methods, it has the advantage of smaller index size yet faster query execution speed. However, it has to compute and keep an NAG vector for each string to memorize how many VGRAMs are affected when different numbers of edit operations are applied. This increases the index building time and index size.We will consider this method in the experiment.

3 Chunk Boundary Dictionary and vchunks

In this section, we propose a class of chunking scheme based on the notion of *tail-restricted CBDs*. We will then show such chunking scheme is guaranteed not to have the avalanching effect.

3.1 Tail-Restricted CBDs

A chunking scheme divides a string into disjoint substrings, each called a *chunk*. We focus on chunking schemes that are governed by *chunk boundary dictionaries* (CBDs). A CBD consists of a set of strings, each encoding a particular *rule*. For example, the rule `zza` means for every match of the pattern `zza` in a string s, the last character in the match will be used to divide the string (i.e., `a` in this example). For example, the string `xzzzazaaa` will be partitioned into two chunks by the rule `zza` as: `xzzza` and `zaaa`. We use $C(s, D)$ to denote the set of chunks obtained by partitioning the string s with respect to the CBD D. The number of chunks of a string s, $s.cn$, is defined as $|C(s, D)|$.

Recall Example 1. It is obvious that chunks should not be of fixed-length, or more generally the chunk boundaries should not be determined by absolute offsets. Therefore, we need to investigate schemes such that the chunk boundaries are determined only by the local content [26]. On the other hand, the same example can be interpreted as a chunking scheme using a CBD consisting of all the bi-grams. Hence, not all CBDs are immune to the avalanching effect.

In this paper, we propose a family of CBDs (namely *tail-restricted CBD scheme*) which will result in chunking schemes such that at most two chunks will be destroyed per edit operation. This family of schemes depends on a partitioning scheme Γ which divides the alphabet Σ into two disjoint subsets: the prefix character set $\mathcal{P}$ and the suffix character set $\mathcal{S}$ (i.e., $\mathcal{P} \cup \mathcal{S} = \Sigma$ and $\mathcal{P} \cap \mathcal{S} = \emptyset$). Each rule in the CBD can be described using the regular expression: $[\mathcal{P}]^*[\mathcal{S}]$, where $[\mathcal{P}]^*$ means any string made of characters from $\mathcal{P}$ (including the empty string) and $[\mathcal{S}]$ means a single character from $\mathcal{S}$. A rule u is made redundant by another rule v if v is a suffix of u (including the case where $v = u$). We define *a minimal CBD* is a CBD that has no redundant rules.

A CBD is said to be *conflict free* if the chunking result does not depend on the order of the rules applied. It is easy to see that a tail-restricted CBD is always conflict free, because $\mathcal{P}$ and $\mathcal{S}$ are disjoint. However, if we merge two tail-restricted CBDs into one, the resulting CBD might not be conflict free. Nevertheless, a sufficient condition to test if the resulting CBD is conflict free is to first collect the last character of all the rules into $\mathcal{S}$, and then test if characters in $\mathcal{S}$ only appear in the last character of any rule.

Example 2. *Consider the string* `as␣soon␣as␣possible`*. Let* $\mathcal{S} = \{\,\mathtt{a}, \mathtt{b}\,\}$*, and* $\mathcal{P} = \Sigma \setminus \mathcal{S}$*. Let the CBD* $D_1 = \{\,\mathtt{a}, \mathtt{b}\,\}$*. Then the string* s *will be partitioned into the following set of chunks:*

$$\underline{\mathtt{a}}\quad \underline{\mathtt{s␣soon␣a}}\quad \underline{\mathtt{s␣possib}}\quad \underline{\mathtt{le}}$$

If we choose $D_2 = \{\,\mathtt{a}, \mathtt{xb}\,\}$*, then* $C(s, D_2)$ *will consist of only three chunks:*

$$\underline{\mathtt{a}}\quad \underline{\mathtt{s␣soon␣a}}\quad \underline{\mathtt{s␣possible}}$$

Let $D_3 = \{\,\mathtt{a}, \mathtt{xb}, \mathtt{xa}\,\}$*.* D_3 *is not minimal as any string matching the last rule (i.e.,* `xa`*) will always match the first rule (i.e.,* `a`*). In fact,* D_3 *will always result in the same chunks as* D_2*.*

The following theorem shows that this class of CBDs has the salient property that an edit operation destroys at most two chunks.

Theorem 1. *An edit operation on a string* s *will destroy at most two chunks in* $C(s, D)$ *if* D *is a tail-restricted CBD.*

Proof. (Sketch) Given a string t, we list all the *candidate chunk boundaries* as those positions that one of the suffix characters occurs. The chunking process exams the substring preceding each candidate chunk boundary and decides one is an actual chunk boundary if it matches one of the pattern in the CBD.

A key observation is that any substring between two candidate chunk boundaries[1] does not contain any character from $\mathcal{S}$. Consider a substitution edit operation. It can occur either on a candidate chunk boundary or somewhere between two candidate chunk boundaries.

- For the former case, if the result of a substitution is still a character from $\mathcal{S}$, we may or may not have a chunk boundary in the original position. Even if the chunk boundary is destroyed, we just destroyed two chunks.

- For the latter case, the edit operation might affect the candidate chunk boundary after the location of the edit operation. This will destroy at most two chunks.

Insertions and deletions can be analyzed in a similar way. □

Example 3. *Consider the string* `abcdefgh` *and the CBD* $\{$ `ab`, `cd`, `ef`, `gh` $\}$. *The string is partitioned into four chunks:*

$$\underline{\texttt{ab}}\quad \underline{\texttt{cd}}\quad \underline{\texttt{ef}}\quad \underline{\texttt{gh}}$$

If we substitute the character `d` *with* `x`*, the string will be partitioned into three chunks:*

$$\underline{\texttt{ab}}\quad \underline{\texttt{cxef}}\quad \underline{\texttt{gh}}$$

It is shown that only two chunks `cd` *and* `ef` *are destroyed by the edit operation.*

Note that tail-restricted CBDs are just a sufficient condition to obtain the property that only a bounded number of chunks, rather than all chunks, are destroyed per edit operation. It is *not* a necessary condition though. In this paper, we will focus on tail-restricted CBDs, its application to similarity joins, and its selection algorithm. This is mainly because there is already a large amount of (minimal) tail-restricted CBDs to choose from[2], and our CBD selection algorithm (See Section 5) can already select a good CBD that results in small candidate sizes and fast join execution speed. In the rest of the paper, we will simply call tail-restricted CBDs CBDs.

Given a CBD, we can partition a string into a set of chunks. The starting position of a chunk in the string is called its *position*. The order of a chunk according to ts position in the string is called its *rank*. A *vchunk* is a chunk together with its position and rank information, represented as *(chunk, pos, rank)*. When there is no ambiguity, we use "chunk" and "vchunk" interchangeably.

3.2 Generating Chunks

We outline the chunking algorithm with respect to a CBD. We first preprocess the CBD by encoding the rules into a reverse trie. For an input string, we scan its characters to find all instances of characters in the suffix character set ($\mathcal{S}$); for each match, we scan backwards on the string and simultaneously on the trie. We will find a chunk boundary if we reach a leaf node of the trie. If there is no matching node in the trie, we will move to the next match.

4 VChunk-based Edit Similarity Joins

In this section, we discuss our proposed vchunk-based edit similarity join algorithm VChunkJoin. We will also introduce new filtering mechanisms enabled by the unique nature of the vchunks.

[1]Treat the beginning and the end of the string as special candidate chunk boundaries.
[2]The number of different Minimal CBDs is at least 4.8×10^{11} for the English alphabet $|\Sigma| = 26$.

4.1 The VChunkJoin Algorithm

The basic version of the VChunkJoin algorithm can be thought of as a chunk-based counterpart of the basic All-Pairs-Ed algorithm (Algorithm 1). Before we present the algorithm, we introduce several filters, some are analogous to those for gram-based join methods, the rest are unique to our vchunk-base join method.

Definition 1 (Matching vchunks)**.** *Two vchunks u and v are said to be matching (with respect to τ) if*

- *their contents are the same, and*
- *their positions are within τ, and*
- *their ranks are within τ.*

According to Theorem 1, for τ edit operations, at most 2τ vchunks will be destroyed. We can establish the following *matching chunk count filtering* condition.

Lemma 2 (Matching Chunk Count Filtering)**.** *Consider two strings s and t. If $ed(s,t) \leq \tau$, s and t must share at least $LB_{s;t}(\tau) = \max(|C(s,D)|, |C(t,D)|) - 2\tau$* matching *vchunks.*

Proof. Consider the edit operations from s to t. According to Theorem 1, each edit operation destroys at most 2 chunks. Hence the number of preserved chunks is at least $|C(s,D)| - 2\tau$. Now consider the edit operation from t to s, we know the common chunks is at least $|C(t,D)| - 2\tau$. Hence the Lemma is proved. □

Except for the rank part, the above filtering condition resembles the count and position filters for q-gram-based methods introduced in [9].

According to the prefix filtering principle, we only need to consider a prefix of length $2\tau + 1$ for each record (Line 5 of Algorithm 2). Notice that this prefix length is guranteed to be no larger than that of the basic All-Pairs-Ed algorithm, whose prefix length is $q\tau + 1$. As the q are usually larger than 2 [33].

4.2 Chunk Number Filtering

An effective filtering condition unique to our vchunk method is the *chunk number filtering.*

Lemma 3 (Chunk Number Filtering)**.** *If $ed(s,t) \leq \tau$, then $||C(s,D)| - |C(t,D)|| \leq \tau$.*

The above lemma holds as each edit operation will alter the number of chunks by at most one.

The chunk number filtering is unique to our vchunk-based method. Note that

- for q-gram-based methods, this filter is made redundant by the length filtering, as the number of q-grams generated from a string s is solely determined by the length of the string (i.e., $|s| - q + 1$).
- for the VGRAM method, it is not clear whether it is possible to have a similar filter on the difference of number of VGRAMs. A straight-forward adaptation will yield a bound that is rather loose. For example, assume the VGRAM dictionary contains all the bi-grams and a 10-gram (“`abcdefghij`”). Then there is only one VGRAM for the string `abcdefghij`. After one deletion (at any place), the string will suddenly have 8 VGRAMs.

Putting the above filters together, we have the VChunkJoin algorithm (Algorithm 2). Compared with the All-Pairs-Ed algorithm, this new algorithm makes three major modifications:

1. The VChunkJoin algorithm replaces q-grams with vchunks. The prefix length is shortened from $q\tau + 1$ to $2\tau + 1$, and hence significantly reduces inverted index size. This also contributes to the reduction of candidate sizes, which in turn reduces the overall running time of the join.
2. Rank and and chunk number filters (Line 9) are introduced in the algorithm to prune the candidate pairs that pass prefix-filtering and length-filtering.
3. An improved verification algorithm is used (the VerifyVChunk algorithm). We use the fast thresholded edit distance verification algorithm, which has a time and space complexity of $O(\tau \cdot \min(|s|, |t|))$ [29].

Algorithm 2: VChunkJoin (R, τ)

```
Data: $x$ and $y$ are vchunk arrays.
1  $S \leftarrow \emptyset$;
2  Initialize the inverted index $I$;
3  for each vchunk array $x \in R$ do
4      $A \leftarrow$ empty map from id to boolean;
5      $p_x \leftarrow 2\tau + 1$ ;        /* can be improved by location-based mismatch filtering */
6      for $i = 1$ to $p_x$ do
7          $w \leftarrow x[i].token$;   $pos_x \leftarrow x[i].pos$;   $rank_x \leftarrow x[i].rank$;
8          for each $(y, pos_y, rank_y) \in I_w$ such that $y.strlen \geq x.strlen - \tau$ and $A[y]$ has not been initialized
           do
9              if $|pos_x - pos_y| \leq \tau$ and $|rank_x - rank_y| \leq \tau$ and $|x.cn - y.cn| \leq \tau$ then
10                 $A[y] \leftarrow$ true ;                                  /* $y$ is a candidate */
11
12         $I_w \leftarrow I_w \cup \{ (x, pos_x, rank_x) \}$ ;                        /* index $w$ */
13     VerifyVChunk($x, A$);
14 return $S$
```

4.3 Further Optimizations

The basic version of VChunkJoin can be further improved by integrating the following filtering methods.

Location-based Mismatch Filtering

The location-based mismatch filtering [33] can be adapted to the VChunkJoin algorithm too. It is based on the observation that if two chunks in the prefix of a string are not adjacent to each other in the string, it takes at least two edit operations to destroy both of them. Applying this idea further, we can select a prefix of the $2\tau + 1$ chunks in the prefix such that they require at least $\tau + 1$ edit operations to destroy all of them.[1]

Algorithm 3: MinEditErrors (Q)

```
Input   : $Q$ is an array of vchunks (in fact, it is a prefix of the prefix vchunks of a string); $u$ records the
          number of contiguous chunks accumulated so far.
1  Sort vchunks in $Q$ in increasing order of their ranks if necessary;
2  $cnt \leftarrow 0$;   $rank \leftarrow -1$;   $u \leftarrow 0$;
3  for $i = 1$ to $|Q|$ do
4      if $Q[i].rank > rank + 1$ then
5          $cnt \leftarrow cnt + \lceil u/2 \rceil$;
6          $u \leftarrow 1$;
7      else
8          $u \leftarrow u + 1$;
9      $rank \leftarrow Q[i].rank$;
10 $cnt \leftarrow cnt + \lceil u/2 \rceil$;
11 return $cnt$
```

Algorithm 3 determines the minimum number of edit operations needed to destroy a subset (to be precise, a prefix) of chunks in the prefix of a vchunk array. We then can use binary search to determine the minimum set (or prefix) such that Algorithm 3 returns $\tau + 1$, much in the same way as [33]. The binary search algorithm has an $O(\tau \log^2 \tau)$ time complexity and it will be invoked to calculate p_x in Line 5 of Algorithm 2.

Example 4. *Consider the two strings in Example 1. Assume the CBD is* $\{$ `ab`, `cd`, `ef`, `gh` $\}$. *The two*

[1]Note that all chunks in the prefix are ordered by increasing *idf* values.

strings will be partitioned into the following chunks:

$$\underline{\texttt{ab}}\ \ \underline{\texttt{cd}}\ \ \underline{\texttt{ef}}\ \ \underline{\texttt{gh}}$$
$$\underline{\texttt{bcd}}\ \ \underline{\texttt{ef}}\ \ \underline{\texttt{gh}}$$

Assuming $idf(\texttt{bcd}) > idf(\texttt{gh}) > idf(\texttt{ef}) > idf(\texttt{cd}) > idf(\texttt{ab})$, the vchunks in the prefixes of the two strings without the location-based mismatch filtering are:

$$\{\,(\texttt{gh}, 7, 4), (\texttt{ef}, 5, 3), (\texttt{cd}, 3, 2)\,\}$$
$$\{\,(\texttt{bcd}, 1, 1), (\texttt{gh}, 6, 3), (\texttt{ef}, 4, 2)\,\}$$

With the location-based mismatch filtering, their prefixes are:

$$\{\,(\texttt{gh}, 7, 4), (\texttt{ef}, 5, 3), (\texttt{cd}, 3, 2)\,\}$$
$$\{\,(\texttt{bcd}, 1, 1), (\texttt{gh}, 6, 3)\,\}$$

In this example, this filtering reduces the prefix length of the second string from 3 to 2. Note that the two strings will form a candidate pair as the two gh *vchunks are* matching *with respect to $\tau = 1$.*

Content-based Mismatch Filtering

Content-based mismatch filtering proposed in [33] cannot be directly applied, as there might exist many short mismatching chunks. Observing that matching chunks do not contribute to the L_1 distance, we use the following filtering method for our vchunk method: in each iteration, we consider a probing window from the start of the first mismatching chunk to the end of the i-th mismatching chunk; we prune the candidate pair if the L_1 distance of string contents in the probing window is larger than 2τ.

This filtering will be integrated into the verification algorithm (VerifyVChunk) and be applied to candidate pairs before the final edit distance verification.

Virtual CBD Filtering

We can further strengthen the chunk number filtering by using several different CBDs. We call these additional CBDs *virtual CBDs*, as we only need to store the resulting chunk numbers rather than storing and indexing the resulting chunks. Since the additional storage overhead per string is small (usually 2 bytes suffice) per virtual CBD, we can afford using several virtual CBDs.

A straight-forward method to utilize multiple virtual CBDs is to use them individually to perform the chunk number filtering for candidate pairs. A candidate pair is pruned if it fails on any of the virtual CBDs.

In fact, we can perform a sophisticated pruning based on several virtual CBDs under a certain condition. The idea is that we can combine two virtual CBDs into a new CBD, and we can calculate the chunk numbers under this new CBD, when the condition in the following Lemma holds.

Lemma 4 (Merge CBDs)**.** *Let D_1 and D_2 be two CBDs. If $D = D_1 \cup D_2$ is a* non-redundant *and* conflict-free *CBD, then for any string s, $C(s, D) = C(s, D_1) + C(s, D_2)$.*

Therefore, if we materialize the chunk numbers with respect to d virtual CBDs, we can perform $2^d - 1$ chunk filterings using the above merging technique. This greatly enhances the chunk number filtering power. Better yet, we can have an $O(d)$ algorithm to achieve the same pruning power as the one that considers all the $O(2^d)$ merged CBDs. In this linear algorithm, we accumulate virtual CBDs such that one string has more chunks than the other, and then perform chunk number filtering on the CBD that is formed by merging those CBDs. We perform another round of filtering for the CBD formed by merging the rest of the CBDs.

The pseudo-code for the linear-time virtual CBD-based filtering (integrated into the chunk number filtering) is given in Algorithm 4. The algorithm will replace the basic version of chunk number filtering (i.e., $|x.cn - y.cn| \le \tau$) in Line 9 of Algorithm 2.

Example 5. *Assume we use three virtual CBDs D_1, D_2, and D_3, and we know combining any subset of them will result in a non-redundant and conflict-free CBD. Two strings s and t have the following chunk numbers under these CBDs.*

Algorithm 4: ChunkNumberAndVirtualCBDFilters (x, y)

```
Data: $x.cn$ is the chunk number of $x$; $x.cn(D_i)$ is the chunk number of $x$ wrt. the virtual CBD $D_i$
if $|x.cn - y.cn| \leq \tau$ then
    $diff_+ \leftarrow 0$;   $diff_- \leftarrow 0$;
    for each virtual CBD $D_i$ ($1 \leq i \leq d$) do
        $\epsilon \leftarrow x.cn(D_i) - y.cn(D_i)$;
        if $\epsilon > 0$ then
            $diff_+ \leftarrow diff_+ + \epsilon$;
        else
            $diff_- \leftarrow diff_- + |\epsilon|$;

    if $diff_+ > \tau$ or $diff_- > \tau$ then
        return false
    else
        return true

else
    return false
```

String	$C(\cdot, D_1)$	$C(\cdot, D_2)$	$C(\cdot, D_3)$
s	*5*	*8*	*7*
t	*7*	*6*	*9*

Let $\tau = 2$. None of the single virtual CBDs can prune this pair. However, when we combine D_1 and D_3, the chunk numbers for s and t will be 12 and 16, respectively. As a result, this string pair can be pruned. In Algorithm 4, the chunk count number differences of D_1 and D_3 are all collected and accumulated in $diff_-$, which eventually exceeds the threshold τ.

Currently, we use the following simple heuristics to select the virtual CBDs which ensures that the condition in Lemma 4 holds: we randomly partition Σ into multiple disjoint subsets, $S_1, S_2, \ldots, S_d$. We then form CBDs D_i that consists of single-character rules from S_i. This heuristics works well in all the datasets used in the experiment.

5 CBD Selection

CBDs play an important role in the performance of our chunk-based edit similarity join algorithm. Selecting an optimal CBD is hard because (1) it is not easy to determine a cost function to optimize, and (2) the problem is NP-hard under a simplified cost function.

We note that some previous methods resort to heuristics or simply circumvent the problem by using hash functions [21]. Instead, we develop an efficient greedy algorithm to automatically select a good CBD for a given dataset.

5.1 NP-hardness of the Problem

First, we will show the CBD selection problem for even one string is NP-hard in the general case under a simplified cost function.

Consider the worst case scenario: in order for the CBD to be useful for edit similarity join with threshold τ, we need to make sure each string is partitioned into at least $2\tau + 1$ chunks. Therefore, we consider the following cost function: the cost of a CBD D on a string s, $cost(s, D)$, is

$$cost(s, D) = \begin{cases} |C(s, D)| - (2\tau + 1) & \text{, if } |C(s, D)| \geq 2\tau + 1 \\ \infty & \text{, otherwise.} \end{cases}$$

By reduction from the subset sum problem, we have the following theorem.

Theorem 2. *The above optimization problem is NP-hard.*

5.2 A Practical CBD Selection Algorithm

Seeing the difficulty of selecting an optimal CBD in the general case, we design a greedy algorithm to select a good CBD instead. The overall CBD selection algorithm takes as input a suffix character set $\mathcal{S}$ and then performs the following two steps:

- **Step One** Determine a subset of $\mathcal{S}$, named $\mathcal{S}_D$, that can already partition all the strings into at least $2\tau + 1$ partitions.
- **Step Two** Let $\mathcal{S}_R = \mathcal{S} \setminus \mathcal{S}_D$ be the reserved set. We consider adding rules in the form of $[\mathcal{P}]^*[\mathcal{S}_\mathcal{R}]$ which improve the length of the $2\tau + 1$-th longest chunk in the strings.

Intuitively, step one ensures every string is partitioned into at least $2\tau + 1$ chunks, and step two gradually improves the CBD such that the chunks in the prefix of the strings will be longer and hence more selective.

We use the following method to select $\mathcal{S}_D$ in step one. We sort all the characters in Σ in decreasing order of their frequencies. We start with an empty CBD and then greedily select the next most frequent character σ_i and add it to the CBD. This change might render some strings partitioned into more than $2\tau + 1$ chunks — these strings are then removed from further consideration. We repeat the process until all the strings are partitioned into at least $2\tau + 1$ chunks. Due to the zipf-like distribution of characters in most real datasets, this greedy algorithm with the frequency-oriented heuristics can stop very quickly. Therefore, we will assume that all strings in the dataset are partitioned into at least k_1 chunks by $\mathcal{S}_D$.

To implement step two, we start with a CBD made of the single character rules in $\mathcal{S}_D$, and iterate through all the strings in increasing order of their lengths. Assume when considering the i-th string s_i, the CBD obtained so far is D_{i-1}. D_{i-1} determines some chunk boundaries in s_i. There are, however, other *candidate chunk boundaries* which are the occurrence of characters in $\mathcal{S}_R$. Since the current string already has a sufficient number of chunks, we have the luxury to choose a subset of the candidate chunk boundaries if this can increase the length of the $2\tau + 1$-th longest chunk of *most* of the strings[1]. If we choose to use a candidate chunk boundary, we say we *cut* at this position, and we add the longest string that does not contain any character in $\mathcal{S}$ and ends at the cut position as a new rule to the CBD.

For each new candidate rule r induced by a candidate cut, we use the following heuristic to evaluate its benefit. If using the new rule, the $2\tau + 1$-th longest chunk of a string increases in length, we say the rule has a *positive effect* on the string; if the $2\tau + 1$-th longest chunk decreases in length, we say the rule has a *negative effect* on the string; otherwise, the rule has no effect on the string. The benefit of a rule is the number of its positive strings subtracts the number of its negative strings. We then design a greedy algorithm by repeatedly select the rule that has the maximum benefit to the CBD until there is no rule with positive benefit. Note that in each iteration, after adding a rule r to the CBD, the benefits of other candidate rules need to be updated. Fortuneately, only rules that occurs in those strings that is affected by the new rule r need to be updated, and such update can be performed incrementally.

Example 6. *Consider a collection of two strings $s =$ `abcdefghijkl` and $t =$ `abcdefgjhikl`. Assume $S_D = \{\,$`b`$,$`k`$\,\}$ and $S_R = \{\,$`g`$,$`j`$\,\}$. Let $2\tau + 1 = 3$. Since S_D partitions the two strings into at least 3 chunks, the current CBD D_{i-1} is $\{\,$`b`$,$`k`$\,\}$. $C(s, D_{i-1})$ and $C(t, D_{i-1})$ are:*

$$s = \underline{\texttt{ab}}\quad \underline{\texttt{cdef}\mathit{g}\texttt{hi}\mathit{j}\texttt{k}}\quad \underline{\texttt{l}}$$

$$t = \underline{\texttt{ab}}\quad \underline{\texttt{cdef}\mathit{g}\mathit{j}\texttt{hik}}\quad \underline{\texttt{l}}$$

The lengths of the third longest chunk for both strings are 1. There are two candidate cuts in both strings (marked in red, green, respectively), all located in the longest chunk. Cutting at these positions yields three candidate rules: `cdefg`*,* `hij`*, and* `j`*, with benefit values 2, -1, and 0, respectively. For example, the rule* `cdefg` *will increase the length of third longest chunk to 2 for both strings, while the rule* `j` *is has negative effect on s, but positive effect on t. Our CBD selection algorithm will choose to cut at* `g`*, and*

[1]note that a new rule to be added to the CBD will affect not only the current string, but all the strings in the collection.

add the rule `cdefg` *to the CBD. Afterwards, s and t are partitioned into:*

$$s = \underline{\texttt{ab}}\quad \underline{\texttt{cdefg}}\quad \underline{\texttt{hi}j\texttt{k}}\quad \underline{\texttt{l}}$$
$$t = \underline{\texttt{ab}}\quad \underline{\texttt{cdefg}}\quad \underline{j\texttt{hik}}\quad \underline{\texttt{l}}$$

The two candidate rules has the new benefit values as: `hij` *is -1, and* `j` *is -2. Since there is no rule with positive benefit, we finish the CBD selection algorithm. The final CBD is* $\{\texttt{b}, \texttt{k}, \texttt{cdefg}\}$.

Time Complexity Analysis. Assume every string has the same length as $|s|$. The CBD selection algorithm, in each iteration, increases the length of the $2\tau + 1$-th chunk by at least 1 for at least half of the strings. Since the maximum possible length for the $2\tau + 1$-th chunk is $\frac{|s|}{2\tau+1}$, and this is the maximum number of iterations. The number of rules whose benefit values need to be updated in each iteration is at most $n|s|$. Therefore, the overall time complexity of the algorithm is $O(n^2|s|^2/\tau)$. Note that this is a rather pessimistic estimation; empirically, the algorithm is fast and scales almost linear to the size of the collection (See Table 6.3).

6 Experiments

We present our experimental results and analyses in this section.

6.1 Experiment Setup

The following algorithms are used in the experiment.

- **Ed-Join** is a state-of-the-art edit similarity join algorithm based on q-grams [33]. It has been shown to outperform other algorithms, such as the prefix-filtering-based algorithm [3] and PartEnum [2].

- **Winnowing** is a document fingerprinting algorithm to identifying similar documents [26]. It first extracts q-grams from each string and then only keeps q-grams whose hash values are the minimum within a sliding window of width w. Those selected q-grams are called *fingerprints*. It is guaranteed that two strings must share at least one fingerprint if they have a common substring of length at least $w + q - 1$.

  We modify the winnowing method to work for edit similarity join as follows: it is easy to show that if $ed(s, t) \leq \tau$, they must share a substring of length at least $\lfloor \frac{\max(|s|,|t|)}{\tau+1} \rfloor$. Therefore, we set $w_i = \lfloor \frac{|s_i|}{\tau+1} \rfloor - q + 1$ for each string s_i. Even though each string generates fingerprints using sliding windows of different widths, it can be proved that the common substring guarantee still holds. Content-based mismatch filtering is also incorporated into this modified algorithm.

- **VGram-Join** is an algorithm based on variable-length grams for answering edit similarity selection queries [18, 35]. The VGRAM approach is known for its small index sizes and fast speed compared with fixed-length grams. We obtained the binary VGRAM implementation from the original authors and leveraged it to implement a prefix-filtering-based algorithm to support edit similarity joins. Location-based filtering and content-based filtering are not applied.

- **B^{ed}-tree** [36] is a recent index structure for edit similarity searches and joins based on B$^+$-trees. It proposed three different transformations for efficient pruning of candidates during its query processing. We obtained the implementation from the authors.

- **Trie-Join** [31] is a recent trie-based edit similarity join method. We obtained the binary implementation from the authors.

- **PartEnum** [2] is an edit similarity search and join method based on two-level partitioning and enumeration. We used the implementation in the Flamingo project [1]. The Flamingo Project has implemented the algorithm. The original implementation is for string similarity search problem. We modified the implementation to support similarity join as well as adding a few optimizations (such as randomized partitioning).

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

- **NGPP** [32] is an edit similarity search algorithm originally developed for the approximate dictionary matching problem. It is based on a partitioning scheme together with deletion-neighborhood enumeration. We enhanced the implementation to support edit similarity joins.
- **VChunkJoin** and **VChunkJoin-NoLC** VChunkJoin is our proposed algorithm equipped with all the filterings and optimizations. When comparing with the VGRAM-based method, we remove location-based and content-based filterings, and name the resulting algorithm VChunkJoin-NoLC. We use the following parameters for our vchunk-based algorithms: the number of virtual CBDs (d) is 10.

Among the above algorithms, Ed-Join and Winnowing are fixed length gram-based methods, VGram-Join is variable length gram-based methods, B^{ed}-tree and Trie-Join are tree-based methods, and PartEnum and NGPP are enumeration-based methods.

The fast $O(\tau \cdot \min(n, m))$ thresholded edit distance verification algorithm [29] is used for the final verification for all algorithms.

All algorithms are implemented as in-memory algorithms, with all their inputs loaded into memory before running. We choose to hash q-grams/VGRAMs/vchunks to 4-byte integers. We use 2-byte short integers to represent the position of q-grams/VGRAMs/vchunks and *rank* of vchunks.

All experiments were carried out on a PC with Intel Xeon X3220@2.40GHz CPU and 4GB RAM. The operating system is Debian 4.1.1-21. All algorithms were implemented in C++ and compiled using GCC 4.3.2 with `-O3` flag.

We used several publicly available real datasets in the experiment. They were selected to cover a wide range of data distribution s and application domains, and also because they were used in previous approaches.

- **IMDB** is a collection of actor names downloaded from IMDB Web site. [2] It contains about 1.2M actor names.
- **DBLP** is a snapshot of the bibliography records from the DBLP Web site.[3] It contains about 900K records. Each record is a concatenation of author name(s) and the title of a publication.
- **TREC** is from the TREC-9 Filtering Track Collections[4]. It has about 350K references from the MEDLINE database. We extract and concatenate the author, title, and abstract fields.
- **UNIREF** is the UniRef90 protein sequence data from the UniProt project.[5] We extract the first 500K protein sequences; each sequence is an array of amino acids coded in uppercase.
- **ENRON** This dataset is from the Enron email collection with about 390K emails.[6] We extract and concatenate the email title and body.

These datasets are transformed and cleaned as follows: (1) We convert white spaces and punctuations to underscores, letters to their lowercases for TREC and ENRON. UNIREF is already clean data. In order to study the effect of large alphabet sizes, we choose not to perform such conversion on IMDB and DBLP; (2) We remove exact duplicates; (3) We remove strings whose length is smaller than a threshold. This is because the q-gram-based and Winnowing-based methods require strings longer than $q \cdot (\tau + 1)$. We choose minimum string length thresholds as follows: 12 for IMDB, 30 for DBLP, 168 for TREC, UNIREF and ENRON; (4) We sort the strings into increasing order of length.

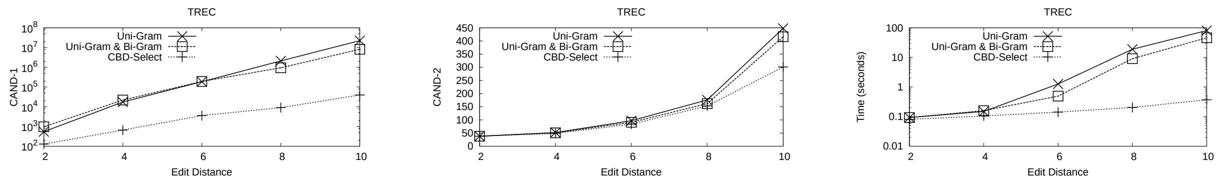
Some important statistics about these datasets after the cleaning are listed in Table 6.1.

We use 3-grams on IMDB, 5-grams on DBLP, and 8-grams on TREC, UNIREF and ENRON, which result in the best performance for both Ed-Join and Winnowing [33]. For VGRAM, we set the sampling ratio to 10%, the number of workload queries to 5,000, and $q_{\min}$ to 4, according to [35]. For PartEnum, we fix $q = 1$ according to [2] and then manually tune its parameters n_1 and n_2. The best performance is achieved by using $n_1 = 3$, $n_2 = 4$ for $\tau = 1$ and using $n_1 = 3$, $n_2 = 7$ for $\tau = 2$ or $\tau = 3$. For NGPP, l_p is set to 7 to obtain best running time.

[2] `http://www.imdb.com`
[3] `http://www.informatik.uni-trier.de/~ley/db`
[4] `http://trec.nist.gov/data/t9_filtering.html`
[5] `http://beta.uniprot.org/`
[6] `http://www.cs.cmu.edu/~enron/`

Table 6.1: Statistics of Cleaned Datasets

Dataset	N	*avg_len*	$\|\Sigma\|$	**Comment**
IMDB	1,060,981	15.7	78	actor names
DBLP	860,751	105.0	93	author, title
TREC	239,580	1227.8	37	author, title, abstract
UNIREF	377,438	463.0	25	protein sequences
ENRON	390,412	2094.1	37	title, body



(a) TREC, CAND-1 (b) TREC, CAND-2 (c) TREC, Running Time

Figure 6.1: Effect of CBD Selection

We take the following measurements: (1) the average length of the prefixes; (2) the total size of the indexes; (3) the number of the candidate pairs formed after probing the inverted index (denoted as **CAND-1**). For VChunkJoin, it is the number of candidate pairs that pass the location-based filtering, and chunk number filtering. For Ed-Join, it is the number of candidate pairs that pass the location-based filtering. For Winnowing, VGram-Join and VChunkJoin-NoLC, it is the number of candidate pairs that pass the prefix filtering; (4) the number of candidate pairs before the final edit distance verification (denoted as **CAND-2**); (5) the running time. The running time does not include preprocessing time or loading time of the q-gram/VGRAM/vchunk arrays unless explicitly specified.[7]

6.2 Evaluating the CBD Selection Algorithm

Our first experiment compares different CBD selection methods for VChunkJoin. To this end, we disable the virtual CBD filtering from the join algorithm, and consider the following approaches:

- **Uni-Grams** We only allow single character rules in the CBD.
- **Uni-Grams & Bi-Grams** We allow both single character and two-character rules in the CBD, provided that they form a tail-restricted CBD.
- **CBD-Select** We select rules using our proposed CBD selection algorithm. We choose the most common characters in the datasets to form $\mathcal{S}$, i.e., $\{\,\mathtt{a},\mathtt{e},\mathtt{h},\mathtt{l},\mathtt{o},\mathtt{r},\mathtt{s},\mathtt{t},\mathtt{u}\,\}$ for IMDB, $\{\,\mathtt{l},\mathtt{a},\mathtt{e},\mathtt{i},\mathtt{n},\mathtt{o},\mathtt{r}\,\}$ for DBLP, $\{\,\mathtt{a},\mathtt{e},\mathtt{i},\mathtt{o},\mathtt{u},\mathtt{r}\,\}$ for TREC, $\{\,\mathtt{a},\mathtt{e},\mathtt{g},\mathtt{l},\mathtt{s},\mathtt{v}\,\}$ for UNIREF, and $\{\,\mathtt{h},\mathtt{o},\mathtt{u},\mathtt{r}\,\}$ for ENRON.

For the first two methods, we wrote a program to search the entire search space with some heuristics and kept the best performing ones found.

We show the results of CAND-1, CAND-2, and running time for different CBDs on the TREC dataset in Fig. 6.1(a)–6.1(c). The general trend is that, our automatically selected CBD outperforms manually selected simple CBDs under all the threshold settings, and the gap is more significant for large edit distance thresholds. This is because when τ is large, a string needs to be partitioned into more chunks, and our algorithm is likely to make a wise choice when selecting a good partitioning scheme.

In the rest of the experiments, we will use the automatically selected CBDs for our VChunkJoin algorithm.

6.3 Data Structure Sizes for gram-based Methods

We show the sizes of various data structures of gram-based algorithms in Table 6.2, since their sizes can all be decomposed into the same three parts. Comparison of index sizes for the other algorithms

[7]The loading time is between 24 to 79 seconds.

Table 6.2: Data Structure Sizes

(a) DBLP, $\tau = 3$

	Index Size	Token Array Size	Data Size	Dict. Entries
VGram-Join	85.5 MB	311.5 MB	91.3 MB	255,667
VChunkJoin	**32.6** MB	**223.9** MB	91.3 MB	815
Ed-Join	42.1 MB	532.5 MB	91.3 MB	n/a
Winnowing	42.5 MB	532.5 MB	91.3 MB	n/a

(b) TREC, $\tau = 10$

VChunkJoin	**26.0** MB	**614.7** MB	284.6 MB	742
Ed-Join	45.1 MB	1,697.3 MB	284.6 MB	n/a
Winnowing	41.1 MB	1,697.3 MB	284.6 MB	n/a

(c) ENRON, $\tau = 10$

VChunkJoin	**36.3** MB	**982.3** MB	780.1 MB	2,826

will be given in Section 6.6. Index sizes are the size of the inverted index (built on the prefixes or the fingerprints). Token array sizes are the sizes of q-grams/VGRAMs and their associated information (e.g., *pos* for q-grams/VGRAMs, and *pos* and *rank* for vchunks) for the strings. It also includes chunk numbers due to virtual CBDs for vchunk-based methods.

Table 6.2(a) shows the data structure sizes for DBLP dataset. It can be observed that (1) VChunkJoin has the smallest index size and token array size. Compared with Ed-Join, its index size is reduced by 22.6% and token array size reduced by 58.0%. (2) VGram-Join has the second smallest token array size but the largest index size. The latter is due to the lack of location-based mismatch filtering. (3) The VGRAM dictionary has far more entries than our CBD.

We do not consider VGram-Join algorithm on TREC and UNIREF because the VGRAM implementation limits the input string to be no longer than 255 characters. The data structure sizes for TREC are shown in Table 6.2(b). The results on UNIREF dataset are similar. Compared with Ed-Join, VChunkJoin reduces the index size by 42.3%, and token array size by 63.8%. The reduction is more significant on TREC than on DBLP.

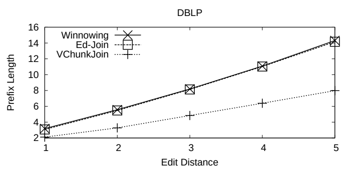
We only perform experiment on ENRON with VChunkJoin algorithm, since the token arrays for Ed-Join and Winnowing are too large and exceed the main memory. The various data structure sizes are shown in Table 6.2(c). The token array size is only slightly larger than the text data size (1.3x), and the index size is quite small.

6.4 Comparing With Ed-Join and Winnowing

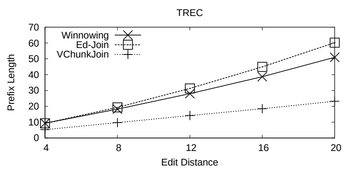
We compare our VChunkJoin algorithm with the Ed-Join and Winnowing algorithms. We can also observe the scalability of those algorithms with respect to the edit distance threshold (τ).

Fig. 6.2(a)–6.2(c) show the prefix lengths for the three algorithms on the three datasets. We abuse the term "prefix length" to denote the number of q-grams selected as fingerprints for the Winnowing algorithm.

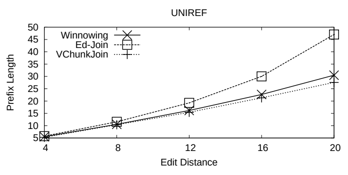
The general trend is that the prefix length grows linearly with the edit distance threshold. We see that Ed-Join and Winnowing exhibit similar prefix lengths on two datasets, while VChunkJoin always has the shortest prefix length in all three datasets. This is expected as (1) the prefix length is $2\tau + 1$ for VChunkJoin, and $q\tau + 1$ for Ed-Join, both in the worst case, and approximately $c(2\tau + 2)$ (where c is a parameter usually slightly larger than 1.0) in the average case; (2) the location-based filtering helps to reduce the prefix length for VChunkJoin and Ed-Join. The prefix length of VChunkJoin is about 60% of Ed-Join on DBLP, and about 40% on TREC. It is interesting to note that, On UNIREF, our prefix length has a less significant reduction compared with the other two algorithms. The reason is that the edit operations are more scattered on protein sequences (UNIREF) than on English texts (DBLP and TREC). For instance, the prefix lengths are 19.3 and 15.4 for Ed-Join and VChunkJoin, respectively, when



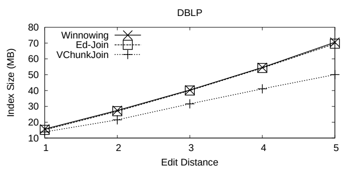
(a) DBLP, Prefix Length



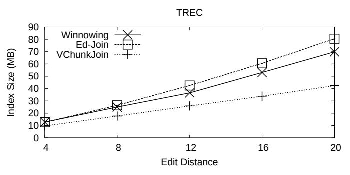
(b) TREC, Prefix Length



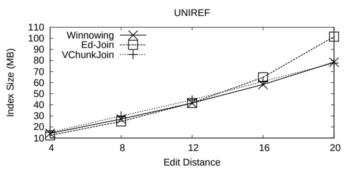
(c) UNIREF, Prefix Length



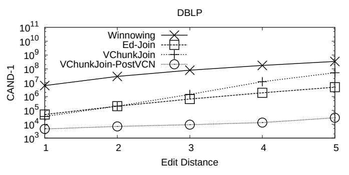
(d) DBLP, Index Size



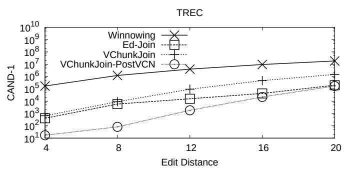
(e) TREC, Index Size



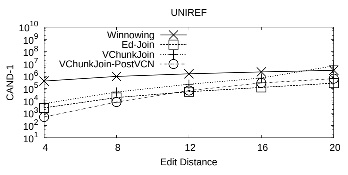
(f) UNIREF, Index Size



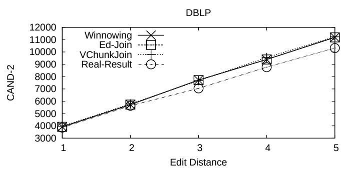
(g) DBLP, CAND-1



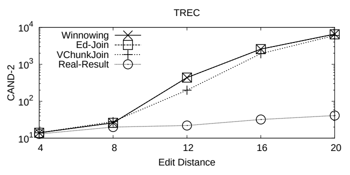
(h) TREC, CAND-1



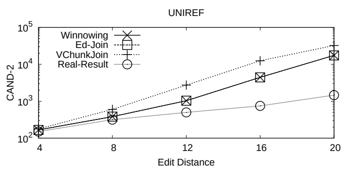
(i) UNIREF, CAND-1



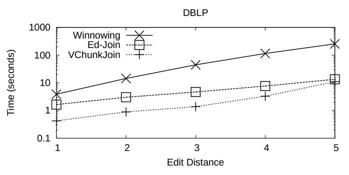
(j) DBLP, CAND-2



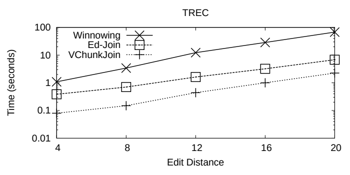
(k) TREC, CAND-2



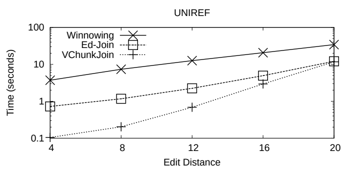
(l) UNIREF, CAND-2



(m) DBLP, Running Time

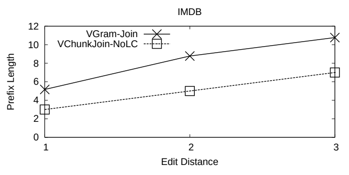


(n) TREC, Running Time

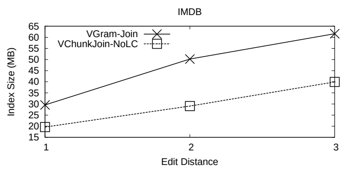


(o) UNIREF, Running Time

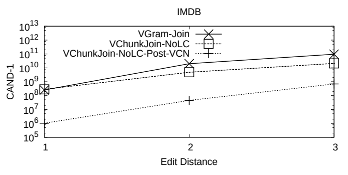
Figure 6.2: Comparison with Ed-Join and Winnowing



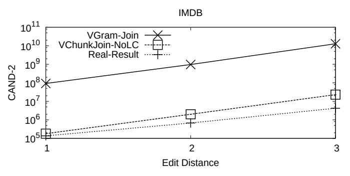
(a) IMDB, Prefix Length



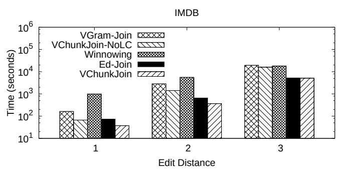
(b) IMDB, Index Size



(c) IMDB, CAND-1

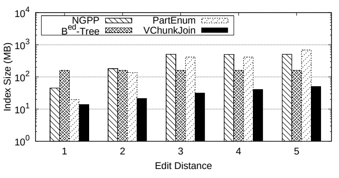


(d) IMDB, CAND-2

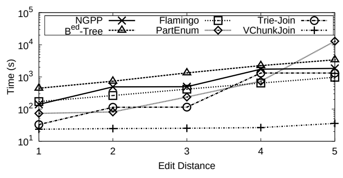


(e) IMDB, Running Time

Figure 6.3: Comparison with VGRAM Method



(a) DBLP, Index Size



(b) DBLP, Running Time

Figure 6.4: Comparison Non-Gram-Based Methods

$\tau = 12$. This showcases the effect of the location-based mismatch filtering for scattered edit errors. Note that the latter value is close to the theoretical lower bound of $\tau + 1 = 13$.

The prefix length has a direct impact on the index size, as shown in Fig. 6.2(d)–6.2(f). It is obvious that VChunkJoin usually has the smallest index. The reason why our index size is slightly larger than that of Ed-Join on the UNIREF dataset is because our index entry has an additional *rank* field than that of Ed-Join.

The CAND-1 sizes for the three algorithms are shown in Fig. 6.2(g)–6.2(i). Winnowing has the largest CAND-1 size because its fingerprinting algorithm ignores the q-gram frequency information and inevitably chooses and indexes some common q-grams. We observe that VChunkJoin generates more CAND-1s than Ed-Join. This is mainly because Ed-Join does a really good job of placing most rare q-grams in the prefixes, while some of the chunks in the prefix of VChunkJoin are not selective enough. Nevertheless, if we consider the candidates that also pass chunk number filtering and virtual CBD filtering (which are unique to our method), the number (named *VChunkJoin-PostVCN* in the fig.) is always the smallest across all datasets.

The CAND-2 sizes produced by the three algorithms are plotted in Fig. 6.2(j)–6.2(l). We also show the size of the join results, which is the lower bound for all algorithms. All three algorithms have similar CAND-2 sizes. This is mainly due to the effectiveness of the content-based mismatch filtering.

Finally, the running times of the algorithms are shown in Fig. 6.2(m)–6.2(o). Winnowing, which produces the largest number of candidates, is the slowest, and its running time grows rapidly with the increase of the edit distance threshold τ. Both VChunkJoin and Ed-Join scale much better with τ. VChunkJoin consistently outperforms Ed-Join on all three datasets, with a speed-up up to 3.9x on DBLP, 4.8x on TREC, and 6.9x on UNIREF, respectively.

6.5 Comparing with the VGRAM Method

We compare our VChunkJoin-NoLC algorithm with the VGRAM algorithm on IMDB dataset. This is done separately because it seems pretty involved to implement the location-based and content-based mismatch filterings for the VGRAM method. Hence, we use the VChunkJoin-NoLC algorithm, which does not include these two filters, in this part of experiment to have a fair comparison.

Fig. 6.3(a)–6.3(b) show the prefix lengths and index sizes for the two algorithms. The prefix length of VChunkJoin-NoLC is exactly $2\tau + 1$, while the prefix length of VGram-Join is obtained from the NAG vectors, which appears to be slightly larger than $q_{\min}\tau + 1$. Our chunk-based method reduces the prefix length by 42%, and the index size by 34%.

Fig. 6.3(c)–6.3(d) show the CAND-1 and CAND-2 sizes produced by the algorithms. Without the content-based filtering, the CAND-2 size is the number of candidates that pass count and position filterings for both algorithms. VChunkJoin-NoLC has smaller candidate sizes for both CAND-1 and CAND-2 measures. The difference in CAND-2 sizes is more significant when τ is large. A major contributing factor is that the lower bound for VChunkJoin-NoLC is tighter than that of VGram-Join. The differences of the candidate sizes have a major impact on the running times of the two algorithms, which are shown in Fig. 6.3(e). VChunkJoin-NoLC is faster than VGram-Join under all the parameter settings, and the speed-up can be up to 7.8x. Two factors contribute to this:

1. VChunkJoin-NoLC has a smaller CAND-1 size. This means fewer inverted index entries are accessed by VChunkJoin-NoLC. In addition, VChunkJoin-NoLC employs the chunk number and virtual CBD filterings which further cut down on the candidate size.

2. With a smaller CAND-2 size, VChunkJoin-NoLC invokes the thresholded edit distance verification routine fewer number of times. This contributes substantially to the difference in running time.

For the sake of completeness, we also plot the running times of other algorithms in Fig. 6.3(e). VChunkJoin outperforms VChunkJoin-NoLC as more filterings are used and is the fastest among all algorithms. It is interesting to see that even VChunkJoin-NoLC outperforms Ed-Join.

6.6 Comparing with the non-gram-based Methods

We compare our VChunkJoin algorithm with four non-gram-based methods: NGPP, PartEnum, Trie-Join, and B^{ed}-tree, and report the results on DBLP dataset, where all these four algorithms implementations can run.

Fig. 6.4(a) shows the index sizes of different algorithms. We observe that VChunkJoin uses the least space. B^{ed}-tree is the runner-up under most parameter settings, and its index size is insensitive to τ as its tree-based index structure is built regardless of τ, agreeing with their theoretical analyses. NGPP and PartEnum display competitive index sizes for small τ settings, yet their index sizes increase rapidly with τ. Note that we were not able to report Trie-Join's index size as such measure is not available from the binary code we obtained from the authors.

Next, we plot the running times of the five algorithms in Fig. 6.4(b). Note that this running time includes the time for preprocessing data. VChunkJoin is always faster than all the non-gram-based methods, and the speed-up can be up to 100x against the runner-up, Trie-Join.

6.7 Preprocessing Time

We measured preprocessing cost for the algorithms. For various algorithms, the preprocessing time includes

- Ed-Join extracting q-grams & sorting by decreasing *idf*.
- Winnowing extracting the q-grams that have the minimum hashed value within each sliding window.
- VChunkJoin selecting CBD, collecting vchunks, and sorting by decreasing *idf*.
- VGram-Join selecting dictionary, computing NAG, extracting VGRAMs, and sorting by decreasing *idf*.
- B^{ed}-tree building B^+-tree index.
- PartEnum generating signatures by partitioning and enumerating.
- NGPP partitioning and generating 1-variants.

We were not able to report Trie-Join's preprocessing time as such measure is not available from the binary code we obtained from the authors.

The preprocessing time for different algorithms is given in Table 6.3. We also show the time for tokenizing strings into q-grams (essentially the approach in [9]) and name it q-gram-base. This can serve as a *baseline* preprocessing time for all q-gram-based methods; note that edit similarity joins using this method will be extremely costly (typically hours). We observe that the preprocessing cost of VChunkJoin is lower than other approaches, and is only moderately higher than q-gram-base. The main reason is that VChunkJoin partition strings into non-overlapping chunks, and thus the number of vchunks in a string is much smaller than that of q-grams. For example, the average number of q-grams in a string is 1,120.8 on TREC, whereas only 403.7 vchunks are collected from a string on average. Among non-gram-based methods, NGPP and PartEnum has little preprocessing time for short strings, but the time increases rapidly for long strings. B^{ed}-tree's proprocessing time is pretty stable across datasets. Nevertheless, VChunkJoin is still the most efficient method.

Summary. Considering the space usage and the runtime performance of all the algorithms compared, we find that VChunkJoin achieves the best performance while occupying the least amount of space.

Table 6.3: Preprocessing Time

	IMDB	DBLP	TREC	UNIREF
q-gram-base	**6.8** s	**16.6** s	**48.8** s	45.8 s
Ed-Join	10.9 s	84.0 s	339.9 s	239.8 s
Winnowing	8.5 s	52.3 s	203.2 s	155.6 s
VGram-Join	224.0 s	5468.0 s	8186.0 s	>10 hrs
NGPP	14.8 s	34.9 s	105.2 s	106.4 s
PartEnum	15.5 s	71.7 s	752.7 s	428.0 s
B^{ed}-tree	59.0 s	67.0 s	98.0 s	79.0 s
VChunkJoin	**7.6** s	**23.5** s	**79.6** s	**41.8** s

7 Related Work

[8] is a recent survey with a section on similarity search and join methods for the task of record linkage and near duplicate object detection. A more comprehensive treatment is in the recent tutorials [15, 11]. [23] is a survey on approximate string matching methods.

Recent progress in the literature that is related to similarity joins includes similarity joins with various similarity or distance functions [25, 6, 3, 34, 33, 18, 35, 19], [7], [31], [36], similarity selection [17, 10], and selectivity estimation [14, 16, 22, 12].

The methods proposed for edit similarity joins or selections can be classified into three categories:

- *Gram-based.* Traditionally, fixed length q-grams are widely used for edit similarity joins or queries, because the count filtering is very effective in pruning candidates [9]. Together with prefix-filtering [6], the count filtering can also be implemented efficiently. Filters based on mismatching q-grams are proposed to further speed up the query processing [33]. Variable-length grams are also proposed [18, 35], which can be easily integrated into other algorithms and help to achieve better performance.

- *Tree-based.* A trie-based approach for edit similarity search has been proposed in [7]. It builds a trie for the dataset and support edit similarity queries by incrementally probing the trie. [31] is also based on the trie data structure to support edit similarity joins with multiple sub-trie pruning techniques. [36] proposes an index structure named B^{ed}-tree to support edit similarity selection and join queries by mapping strings into a linear space which is supported by a standard B^{+}-tree, together with several filtering approaches to prune internal and leaf nodes of the B^{+}-tree.

- *Enumeration-based.* Neighborhood generation-based methods enumerates all possible strings obtained by up to τ edit operations. While naïve enumeration method only works in theory, recent proposals using deletion neighborhood [28] and partitioning [32] can work well with small edit distance thresholds. While the above work is mainly for the edit distance selection queries, PartEnum [2] is tailored for edit similarity joins. It performs two levels of partitioning and then enumerate signatures for each string.

In this paper, our main focus is to improve the performance (index size and join efficiency) of gram-based methods, as they have competative performance, and are applicable for a large range of parameter settings. Comparing with VGRAMs, we consider *disjoint* substrings, while VGRAMs considers *overlapping* ones. While both approaches propose algorithms to select a good (not necessarily optimal) "dictionary", very different techniques are used. As for other categories of approaches, tree-based approaches have large index sizes unless strings are fairly short and share many prefixes. Enumeration-based approaches typically generate an enormous amount of signatures and hence index when τ is large. For example, PartEnum generates $O(\tau^{2.39})$ signatures per record [2], and NGPP generates $O(l_p\tau^2)$ [32].

8 Conclusions

In this paper, we investigate a novel approach to processing edit similarity join efficiently based on the idea of partitioning strings into non-overlapping chunks. We devise a special class of chunking schemes based on the notion of tail-restricted CBDs. Our proposed scheme has the good property that an edit operation destroys at most two chunks. Based on this salient property, we design an efficient edit similarity join algorithm, VChunkJoin, that incorporates all existing filters as well as several novel filters. We also consider tackling the hard problem of finding a good chunking scheme and design an efficient greedy algorithm for it. Experimental results show that our proposed algorithm outperforms existing ones based on fixed or variable length grams yet occupies less space.

Bibliography

[1] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, 2008.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.

[4] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, 2009.

[5] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava. Benchmarking declarative approximate selection predicates. In *SIGMOD Conference*, pages 353–364, 2007.

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

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

[8] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

[9] 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*, 2001.

[10] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.

[11] M. Hadjieleftheriou and C. Li. Efficient approximate search on string collections. *PVLDB*, 2(2):1660–1661, 2009.

[12] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava. Hashed samples: selectivity estimators for set similarity selection queries. *PVLDB*, 1(1):201–212, 2008.

[13] O. A. Hamid, B. Behzadi, S. Christoph, and M. R. Henzinger. Detecting the origin of text segments efficiently. In *WWW*, 2009.

[14] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, pages 397–408, 2005.

[15] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Conference*, pages 802–803, 2006.

[16] H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, pages 195–206, 2007.

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

[18] C. Li, B. Wang, and X. Yang. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, 2007.

[19] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE*, pages 1111–1120, 2008.

[20] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *SIGMOD Conference*, pages 157–168, 2003.

[21] U. Manber. Finding similar files in a large file system. In *USENIX Winter*, pages 1–10, 1994.

[22] A. Mazeika, M. H. Böhlen, N. Koudas, and D. Srivastava. Estimating the selectivity of approximate string queries. *ACM Trans. Database Syst.*, 32(2):12, 2007.

[23] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[24] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB*, pages 351–362, 2000.

[25] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

[26] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD Conference*, pages 76–85, 2003.

[27] J. Seo and W. B. Croft. Local text reuse detection. In *SIGIR*, pages 571–578, 2008.

[28] B. S. T. Bocek, E. Hunt. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007.

[29] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985.

[30] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[31] J. Wang, J. Feng, and G. Li. Trie-join: Efficient trie-based string similarity joins with edit. In *VLDB*, 2010.

[32] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit constraints. In *SIMGOD*, 2009.

[33] 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.

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

[35] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD Conference*, pages 353–364, 2008.

[36] 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 *SIGMOD Conference*, pages 915–926, 2010.