Ed-Join: An Efficient Algorithm
for Similarity Joins With Edit Distance Constraints

Chuan Xiao   Wei Wang   Xuemin Lin
School of Computer Science and Engineering
The University of New South Wales, Australia
{chuanx, weiw, lxue}@cse.unsw.edu.au

ABSTRACT

There has been considerable interest in similarity join in the research community recently. Similarity join is a fundamental operation in many application areas, such as data integration and cleaning, bioinformatics, and pattern recognition. We focus on efficient algorithms for similarity join with edit distance constraints. Existing approaches are mainly based on converting the edit distance constraint to a weaker constraint on the number of *matching q*-grams between pair of strings.

In this paper, we propose the novel perspective of investigating *mismatching q*-grams. Technically, we derive two new edit distance lower bounds by analyzing the locations and contents of mismatching q-grams. A new algorithm, Ed-Join, is proposed that exploits the new mismatch-based filtering methods; it achieves substantial reduction of the candidate sizes and hence saves computation time. We demonstrate experimentally that the new algorithm outperforms alternative methods on large-scale real datasets under a wide range of parameter settings.

1. INTRODUCTION

With the wide availability of data sources on the Web and increasing demands for data integration within enterprises, similarity join has become an essential procedure to provide an effective and efficient way to correlate data together. Similarity join between two sets of objects returns pairs of objects from each set such that similarity values between the pairs are above a given threshold. Due to its importance, similarity join has been studied in many areas, such as data integration and cleaning, bioinformatics, and pattern recognition. Similarity join is also adopted in the industry solutions. For example, Google adopts both approximate and exact similarity join for near duplicate Web page detection [18], query log mining, and collaborative filtering [3]. Microsoft researchers proposed the SSJoin primitive operator [12] to support similarity join and it has been used in the Data Debugger project [11].

In this paper, we focus on similarity joins with edit distance thresholds or *edit similarity join* [12]. Edit distance measures the minimum number of edit operations (insertion, deletion, and substitution) to transform one string to another. Edit distance has two distinctive advantages over alternative distance or similarity measure: (a) it reflects the ordering of tokens in the string; and (b) it allows non-trivial alignment. These properties make edit distance a good measure in many application domains, e.g., to capture typographical errors for text documents, and to capture similarities for Homologous proteins or genes.

Similarity joins with edit distance thresholds pose serious algorithmic challenges. Computing edit distance is more expensive ($O(n^2)$) than alternative distance or similarity measure (usually $O(n)$). Even approximating edit distance is hard. For example, Andoni et al. showed that edit distance cannot be embedded into $\mathcal{L}_1$ with distortion less than $3/2$ [1]. As a result, the naïve algorithm that evaluates edit distance for every pair of strings would incur an prohibiting $O(N^2 \cdot n^2)$ running cost, where N is the number of strings and n is the length of the strings.

Current state-of-the-art approaches to process edit similarity join are mainly based on a *filter-and-verify* approach, operating on the q-gram representation of strings [14]. The q-gram representation of a string is the set of substrings obtained by sliding a window of length q over the string. If there are only few edit errors between two strings, the majority of the q-grams in one string will be preserved and we should be able to find them in approximately the same locations in the other string. q-grams that are preserved are called *matching q-grams*. Several filtering conditions regarding the total number and locations of the matching q-grams and the lengths of the strings were developed [14]. These filtering conditions are only necessary conditions for the edit distance threshold, and therefore, string pairs that survive all the filters still need to be *verified* by the edit distance calculation. Hence, the efficiency of this filtering-based approach critically depends on the pruning power of the filters. A recent progress is the introduction of prefix filtering [30, 12, 3]. When applied to the edit similarity join, it usually reduces the candidate size significantly and hence speeds up the computation.

Unlike all the existing approaches, this paper takes a novel perspective by studying **mismatching q-grams**. Intuitively, traditional filtering methods based on the count of matching q-grams have high computational cost as they have to access *all* the q-grams before rejecting a candidate pair. We argue that q-grams that cannot be matched also provide valuable

information on the similarity of strings. Technically, we derive two novel lower bounds to edit distance by analyzing mismatching q-grams. Two filtering methods, *location-based* mismatch filtering and *content-based* mismatch filtering, are developed that usually reduce the size of the candidates substantially compared with previous methods [14, 12, 3]. We also show how the additional filters are complementary to existing ones and that they are all integrated into a new algorithm, Ed-Join, to attain the maximal pruning power. Our extensive experimental evaluation verifies the superior efficiency of the new algorithm to alternative methods.

We make the following contributions in the paper:

- We propose a novel perspective of analyzing the locations and contents of *mismatching* q-grams to speed up edit similarity join computation. We developed two new filtering methods that are especially effective against *non-clustered edit errors* and *clustered edit errors*, respectively.

- We propose a new algorithm, Ed-Join, that integrates the new mismatch filtering methods with existing filters. Experimental results on several large-scale datasets demonstrate the substantial reduction of candidate sizes and the running time.

- Based on the experimental results, we recommend considering longer q-grams for *stand-alone* implementation of edit similarity join on medium to long strings. We present our analysis on why this deviates from the previous recommendation of $q = 2$ [14]. This finding might impact many data cleaning toolkits that currently rely on bi-grams or tri-grams for edit similarity join.

The rest of the paper is organized as follows: Section 2 introduces preliminaries and backgrounds. Sections 3 and 4 present location-based and content-based mismatch filtering and their integration with the new proposed algorithm Ed-Join. Section 5 discusses several implementation issues. Experimental results and analyses are given in Section 6. Section 7 presents related work and Section 8 concludes the paper.

2. PRELIMINARIES

2.1 Problem Definition and Backgrounds

Let Σ be a finite alphabet of symbols σ_i ($1 \leq i \leq |\Sigma|$). A string s is an ordered array of symbols drawn from Σ. 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 increasing order of string length. $ed(x, y)$ denotes the edit distance between strings x and y, which measures the minimum number of edit operations (insertion, deletion, and substitution) to transform one string to another (and vice versa). It can be computed in $O(n^2)$ time and $O(n)$ space using the standard dynamic programming [32].

Given two sets of strings R and S, a similarity join with edit distance threshold τ (or *edit similarity join* [12]) returns pairs of strings from each set, such that their edit distance is no larger than τ, i.e., $\{ \langle r, s \rangle \mid ed(r, s) \leq \tau, r \in R, s \in S \}$. For the ease of exposition, we will focus on the self-join case in the paper, i.e., $\{ \langle r_i, r_j \rangle \mid ed(r_i, r_j) \leq \tau \wedge r_i.id < r_j.id, r_i \in R, r_j \in R \}$.

A q-gram is a contiguous substring of length q; and its starting position in a string is called its *position* or *location*. A *positional q-gram* is a q-gram together with its position, usually represented in the form of $(token, pos)$ [14]. For simplicity, when there is no ambiguity, we use "positional q-gram" and "q-gram" interchangeably.

Let w be a q-gram and $df(w)$ be the number of strings containing w. The *inverse document frequency* of w, $idf(w)$, is defined as $1/df(w)$. Intuitively, q-grams with high idf values are rare q-grams in the collection.

We can extract all the positional q-grams of a string and order them by decreasing order of their idf values and increasing order of their locations. We call the sorted array the *q-gram array* of the string. Sorting positional q-grams in this order is a good heuristic to speeding up similarity joins [12].

A string s can generate $l = |s| - q + 1$ q-grams.[1] Given a q-gram array x, $str(x)$ denotes its corresponding string. The i-th positional q-gram in x is denoted as $x[i]$; its q-gram and location are denoted as $x[i].token$ and $x[i].loc$, respectively. The k-prefix of x is its first k entries, i.e., $x[1..k]$.

An inverted index for q-grams is a data structure that maps a q-gram w to an array I_w containing entries in the form of (id, loc), where id identifies the sting that contains w and loc is the starting location of w in the string identified by id. The entries in I_w are sorted in the increasing order of id and loc.

EXAMPLE 1. *Consider the string $s = $ abaabab. Let $q = 2$, it has $l = 6$ positional q-grams: (ab, 1), (ba, 2), (aa, 3), (ab, 4), (ba, 5), and (ab, 6). If $idf(\text{aa}) \geq idf(\text{ab}) \geq idf(\text{ba})$, then the q-gram array of s is*

(aa, 3)	(ab, 1)	(ab, 4)	(ab, 6)	(ba, 2)	(ba, 5)

The inverted list for q-gram ba will be

(s, 2)	(s, 5)	...	...

2.2 q-gram-based Filtering

[14] proposed an efficient solution for edit similarity join in a database setting. The method essentially relaxes the edit distance constraint to a *weaker* count constraint on the number of *matching* q-grams. Two q-grams match if they have the same token *and* their locations are within the edit distance threshold τ. Specifically, three filters were proposed as follows: if two strings s and t are within edit distance τ, then

Count Filtering mandates that s and t must share at least $LB_{s;t} = (\max(|s|, |t|) - q + 1) - q \cdot \tau$ common q-grams.

Position Filtering mandates that s and t must share at least $LB_{s;t}$ *matching positional* q-grams.

Length Filtering mandates that $||s| - |t|| \leq \tau$.

Note that a pair of strings that passes the filters (called *candidate pair*) does not necessarily satisfy the edit distance constraint. Therefore, edit distance calculation is needed for every candidate pair that survives all the preceding filters.

Given a pair of string s and t and a threshold τ, q-grams in s that cannot be matched with any q-gram in t are called *mismatching q-grams from s to t*, or simply s's *mismatching q-grams* if there is no ambiguity.

2.3 Prefix Filtering

A major performance bottleneck in [14] is the generation of candidate pairs that share $LB_{s;t}$ matching q-grams. We cannot discard a candidate pair unless all their q-grams have been accessed and compared.

---

[1] Unlike [14], we choose not to add special symbol '#' or '$' to s as prefix or suffix for the ease of illustration. Otherwise, l should be $|s| + q - 1$.

This weakness has been identified by several researchers [30, 12]. A prefix-based filtering is proposed to quickly filter out candidate pairs that are guaranteed not to meet the $LB_{s;t}$ threshold. Figure 1 illustrates the idea of prefix filtering in the context of edit similarity join.
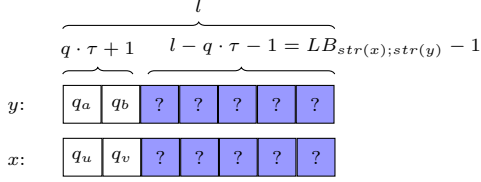

Figure 1: Illustration of the prefix filtering (l is the total number of q-grams for both q-gram arrays; the unshaded cells are prefixes of length $q \cdot \tau$; if x and y has no matching q-gram in their prefixes, their matching q-grams are no more than $LB_{str(x);str(y)} - 1$, as both arrays follow the same ordering).

We formally state the prefix filtering principle for the edit similarity join in Lemma 1. More details and its application to other similarity/distance measures can be found in [12, 3].

LEMMA 1 (PREFIX FILTERING). *Let x and y be two q-gram arrays and $ed(str(x), str(y)) \leq \tau$. Then the $(q \cdot \tau + 1)$-prefix of x and the $(q \cdot \tau + 1)$-prefix of y must have at least one matching q-gram.*

2.4 The All-Pairs-Ed Algorithm

Previous work on edit similarity join algorithm employing prefix filtering is an RDBMS-based implementation that exploits the set-based and group-base processing mechanism by the RDBMS [12].

In this paper, we consider the stand-alone implementation. We modify a state-of-the-art, prefix-filtering-based similarity join algorithm, All-Pairs [3], for edit similarity join.[2] We name it All-Pairs-Ed and its pseudo-code is given in Algorithm 1. All-Pairs-Ed follows an index nested loop join style and maintains an in-memory inverted index on q-grams on-the-fly. Specifically, it assumes the input is a set of q-gram arrays, sorted by in increasing order of their length. It iterates through each q-gram array x; for each q-gram w in the $(q \cdot \tau + 1)$-prefix of x, it probes the inverted index to find other q-gram arrays y that contain matching q-grams to w. Afterwards, x and all its candidates will be further checked by the Verify algorithm.

Within Verify, count and positional filterings will be applied to every candidate pair first. Only those that pass both filters will be further checked by performing the expensive edit distance calculation.

We will see later how additional filters can be integrated to it to gain further efficiency. This algorithm will also serve as a baseline algorithm for comparison in our experiment.

3. LOCATION-BASED MISMATCH FILTERING AND ED-JOIN

Algorithm 1: All-Pairs-Ed (R, τ)

1   $S \leftarrow \emptyset$;
2   $I_i \leftarrow \emptyset$ ($1 \leq i \leq |U|$);
3   **for each** $x \in R$ **do**
4      $A \leftarrow$ empty map from id to `boolean`;
5      $p_x \leftarrow q \cdot \tau + 1$;
6   **for** $i = 1$ **to** p_x **do**
7         $w \leftarrow x[i].token$; $loc_x \leftarrow x[i].loc$;
8   **for each** $(y, loc_y) \in I_w$ such that $|y| \geq |x| - \tau$ **and** $A[y]$ has not been initialized **do**
9   **if** $|loc_x - loc_y| \leq \tau$ **then**
10              $A[y] \leftarrow$ **true**;   /* found a candidate */;
11         $I_w \leftarrow I_w \cup \{ (x, loc_x) \}$;
         /* index the current prefix */;
12   Verify(x, A);
13 **return** S

In this section, we first gives an overview of the proposed Ed-Join algorithm for edit similarity join, followed by the introduction to location-based mismatch filtering. The new filtering method forms the first part of the Ed-Join algorithm.

3.1 Overview of the Ed-Join Algorithm

Most existing algorithms for edit similarity join are based on the accumulation and counting of *matching q*-grams.[3] Even with the prefix-filtering optimization, algorithms still have to access and process *all* the q-grams of strings that share a common q-gram in their respective prefixes.

In this paper, we propose to analyze also the *mismatching q*-grams in order to further speed up the join processing. We design a new algorithm, Ed-Join, that employs two novel filtering techniques obtained from analyzing the *locations* and the *contents* of mismatching q-grams. Like previous algorithms, our Ed-Join algorithm has a candidate-generation phase and then a verification phase. Since our new filters are developed from a new perspective, both filters are orthogonal to existing ones (count and position filterings) and can be used with existing ones in a complementary manner. In our Ed-Join algorithm, the new location-based mismatch filtering is applied to both phases and content-based mismatch filtering is applied to the verification phase.

3.2 Location-based Mismatch Filtering

The count filtering essentially answers the question: *"what is the maximum number of q-grams that can be destroyed by τ edit operations"*. Here, we ask an inverse question: *"what is the minimum number of edit operations that cause the observed mismatching q-grams"*. Answering this question leads us to establish a new lower bound for the edit distance.

EXAMPLE 2. *Let $q = 2$ and $\tau = 1$. Consider two strings:*

$$t = \texttt{abccabcc}.$$
$$s = \texttt{abbcabbc}.$$

Count filtering requires 5 matching q-grams in both strings. Without loss of generality, consider one string t, and it has only two out of seven q-grams that are not matched (two cc

---

[2]Although ppjoin family algorithms [34] have been shown to outperform All-Pairs, we don't consider them as the optimizations used in [34] are either made redundant by the location-based mismatch filtering or not easily extendible to the edit distance function.

[3]PartEnum is an exception, which is based on the pigeon hole principle. We discuss and compare with PartEnum in Section 6.4.

not matched), $\langle s, t \rangle$ will become a candidate pair and will be ultimately verified by edit distance.

However, we can consider the *locations of t's mismatching q-grams and obtain a lower bound directly on the edit distance. Obviously, since the two mismatched q-grams are disjoint in location, it takes at least two edit operations to destroy them. Therefore, we can infer a lower bound of the edit distance between the pair to be 2. Hence the pair can be safely discarded.*

To handle the general case, we also need to consider partially overlapping mismatching q-grams. We named the problem as the *minimum edit errors* problem, which can be stated formally as: *Given a set of q-grams Q, find the minimum number of edit operations that destroy all q-grams in Q.* It seems that there is no closed-form formula to calculate minimum edit errors. Nevertheless, we design a greedy algorithm that calculates the number exactly in $O(|Q|)$ time if Q is already sorted in increasing location, or $O(|Q| \log |Q|)$ otherwise.

Algorithm 2: MinEditErrors (Q)

---

1 Sort q-grams in Q in increasing order of locations if necessary;
2 $cnt \leftarrow 0; \quad loc \leftarrow 0;$
3 **for** $i = 1$ **to** $|Q|$ **do**
4   **if** $Q[i].loc > loc$ **then**
5         $cnt \leftarrow cnt + 1;$
6         $loc \leftarrow Q[i].loc + q - 1;$

7 **return** cnt

The algorithm is shown in Algorithm 2. It *greedily* selects the next unprocessed mismatch q-gram ($Q[i]$), makes an substitution edit operation at its last position (i.e., $Q[i].loc + q - 1$), and then removes all the subsequent q-grams that are destroyed by this substitution. The loop continues until all q-grams are destroyed. Note that the default sorting order of q-gram arrays is by decreasing *idf* values of q-grams rather than their locations.

PROPOSITION 1. *Algorithm 2 correctly solves the minimum edit error problem.*

Let $min\text{-}err(Q)$ be the minimum edit errors for a set of mismatching q-grams Q. We also show two properties with respect to the minimum edit errors, which is important to our Ed-Join algorithm.

PROPOSITION 2 (MONOTONICITY).

$$min\text{-}err(Q) \leq min\text{-}err(Q'), \quad \forall Q \subseteq Q'.$$

PROPOSITION 3. $\lceil |Q|/q \rceil \leq min\text{-}err(Q) \leq |Q|.$

LEMMA 2. *Let x and y be two q-gram arrays, and let Q be the set of mismatching q-grams from x to y, then $ed(str(x), str(y)) \geq min\text{-}err(Q_p), \forall Q_p \subseteq Q.$*

Lemma 2 gives a new edit distance lower bound based on any subset of a string's mismatching q-grams.

3.3 Minimum Prefix for Edit Similarity Join

We introduce how location-based mismatch filtering can be applied to the prefixes and achieve substantial reduction of candidate pairs. The filtering can be used on the entire string as well (See Section 4.3).

Recall that prefix filtering requires each string to generate a fixed-length $q \cdot \tau + 1$ prefix and consider as candidates other strings whose prefix matches one of the q-grams in the prefix. This means that a string in a candidate pair can have up to $q \cdot \tau$ mismatching q-grams in the prefixes. It is highly likely that the minimum edit errors of those mismatching q-grams already exceed τ and the candidate pairs are in fact disqualified with respect to the edit distance constraint due to Lemma 2.

Based on the monotonicity of the minimum edit errors (Proposition 2), we can further strengthen the filtering condition by reducing the prefixes to the *minimum prefixes*. Intuitively, the minimum prefix is the *shortest* prefix of the q-gram array x such that if all the q-grams in the minimum prefix are mismatched, it will incur at least $\tau + 1$ edit errors. We call the length of such minimum prefix *minimum prefix length*.

Algorithm 3: CalcPrefixLen (x)

---

1 left $\leftarrow \tau + 1; \quad$ right $\leftarrow q \cdot \tau + 1;$
2 **while** left $<$ right **do**
3   mid $\leftarrow$ (left + right)$/2;$
4   err $\leftarrow$ MinEditErrors($x[1 .. \text{mid}]$);
5   **if** err $\leq \tau$ **then**
6   left $\leftarrow$ mid + 1;
7   **else**
8   right $\leftarrow$ mid;

9 **return** left

A naïve algorithm to find the minimum prefix length by the definition is to iterate over the range $[\tau + 1, q \cdot \tau + 1]$ (due to Proposition 3) and stop at the first number m such that $x[1 .. m]$ will entail more than τ edit errors (by calling Algorithm 2). This algorithm, however, has an $O(u^2)$ running time, where $u = q \cdot \tau + 1$. We propose a more efficient Algorithm 3 that performs a binary search within the same range of $[\tau + 1, q \cdot \tau + 1]$. The time complexity of the algorithm is $O(u \log^2 u)$.

EXAMPLE 3. *Continuing the example in Example 1, the minimum prefix length for $\tau = 1$ will be 2, since the first two q-grams are disjoint in their locations. In contrast, the standard prefix length required by prefix filtering is 3. Similarly, if the threshold $\tau = 2$, the minimum prefix length is 4 while the standard prefix length is 5.*

We now state the location-based mismatch filtering for edit similarity join as follows.

LEMMA 3 (LOCATION-BASED MISMATCH FILTERING). *Let the minimum prefix length for q-gram arrays x and y be l_x and l_y, respectively. If $ed(str(x), str(y)) \leq \tau$, x's l_x-prefix and y's l_y-prefix must have at least one matching q-gram.*

REMARK 1. *We can interpret prefix filtering as a special case of location-based mismatch filtering. Consider the example in Figure 1. If x and y have no matching q-grams in their prefixes, then all q-grams in the prefix of one of them are mismatching q-grams. Denote these mismatching q-grams as Q, and we know $|Q| = q \cdot \tau + 1$. According to Proposition 3, $min\text{-}err(Q) \geq \tau + 1$, and the pair does not meet the edit distance constraint.*

Algorithm 4: Ed-Join (R, τ)
---
1 Change Line 5 in Algorithm 1 to "$p_x \leftarrow$ CalcPrefixLen(x)" ;
2 Use the Algorithm 7
  as the implementation of Verify (Line 12 in Algorithm 1) ;

3.4 The Ed-Join Algorithm

We introduce a new algorithm, Ed-Join (Algorithm 4), that performs edit similarity join efficiently. Conceptually, the algorithm is similar to the All-Pairs-Ed algorithm with two important modifications:

1. A shorter minimum prefix length is used instead of the standard prefix length required by prefix filtering. We note that this optimization is critical in reducing candidate size, a measure that is highly correlated to the overall join performance, in addition to other obvious benefits (e.g., smaller inverted index).

2. A new implementation of the Verify algorithm that exploits another novel mismatch-based filtering. We will show the details in Section 4. The new filtering will effectively reduce the final number of candidate pairs to be verified by the expensive edit distance function.

Other important optimizations to the algorithm are also given in Section 5.

4. CONTENT-BASED MISMATCH FILTERING

In this section, we introduce another filter that complements the location-based mismatch filter. We then introduce an multiple-filters-based Verify algorithm implementation used in our Ed-Join algorithm.

4.1 Content-based Mismatch Filtering

We define *non-clustered edit errors* to be a set of edit errors such that no two of them are within distance q. It can be shown that location-based mismatch filtering is well suited for detecting this type of errors. However, one weakness in location-based mismatch filtering is that we assume the set of observed mismatching q-grams were caused by the *minimum* number of edit operations. It is possible that several edit errors actually occur within the same mismatching q-gram. We call this type of errors *clustered edit errors*.

Clustered edit errors occur fairly frequently in real datasets. For example, it is often the case that an entire word is deleted/inserted/substituted when editing text documents. In protein sequence alignment, it is well-known that a *gap*, i.e., several contiguous insertions or deletions of amino acid residues, is common and needs special treatment as it is likely to be due to one evolutionary event.

We design the *content-based mismatch filtering* to detect clustered edit errors. Our idea is to select a *probing window* and look into the contents of both strings within the probing window; the content difference in the probing windows, when measured by an appropriate distance measure, will lower bound the edit distance of the pair.

A probing window w is an interval $[w.s \mathinner{.\,.} w.e]$. The content of w on a string s is the substring between location $w.s$ and $w.e$, i.e., $s[w.s \mathinner{.\,.} w.e]$.[4] Given a (sub-)string t, its

---
[4] For the easy of illustration, we will assume the probing window is always within the string boundaries. A special padding scheme is used to deal with the general case in the implementation.

frequency histogram H_t is a vector of size $|\Sigma|$, where $H_t[i]$ records the number of occurrences of a symbol $\sigma_i \in \Sigma$ in t. The $\mathcal{L}_1$ distance between two n-dimensional vectors u and v is defined as $\sum_{1 \le i \le n} |u[i] - v[i]|$.
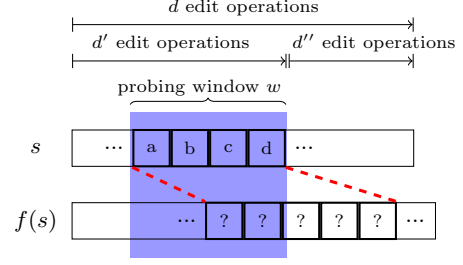

Figure 2: Content-based Mismatch Filtering

Now consider the example in Figure 2 where a string s is transformed into $f(s)$, and the edit distance is d. Suppose there are d' edit operations that occurs before or at the end of probing window w. Consider the $\mathcal{L}_1$ distance (denoted as $d_{\mathcal{L}_1}$) between the frequency histograms for the two strings in the probing window. It is obvious that each edit operation (insertion, deletion, and substitution) will contribute at most 2 to this $\mathcal{L}_1$ distance. Therefore, $d' \ge d_{\mathcal{L}_1}/2$. Finally, since the edit distance of the entire string $d \ge d'$, we have $d \ge d_{\mathcal{L}_1}/2$.

LEMMA 4 (CONTENT-BASED MISMATCH FILTERING). *If the edit distance between the two strings is within τ, there does not exist a probing window such that the $\mathcal{L}_1$ distance between the frequency histograms of two strings within the probing window is larger than 2τ.*

EXAMPLE 4. *Consider $q = 5$, $\tau = 2$, and two strings of length 25:*

$$s = c_1, c_2, c_3, c_4, \underline{c_5, c_6, c_7, c_8, c_9}, c_{10}, \ldots, c_{25}.$$

$$t = c_1, c_2, c_3, c_4, \underline{z_1, z_2, z_3, z_4, z_5}, c_{10}, \ldots, c_{25}.$$

The two strings differ only by the 5 substitutions (from c_i to z_{i-5}). Assume c_i and z_j are all distinct. Then the edit distance between x and y is 5.

The count filtering will require only $25 - 5 + 1 - 5 \cdot 2 = 11$ matching 5-gram. Since there are 12 matching 5-grams between s and t, they will be considered as a candidate pair.

If we take the probing windows $w = [5, 9]$. The $\mathcal{L}_1$ distance between the corresponding contents is 10. This immediately gives us an edit distance lower bound of 5. Hence the pair will be pruned by the content-based mismatch filtering.

Although the above filtering method can be used for any probing window, we found that a good heuristics is to choose a probing window that contains (at least) a mismatching q-grams. The rationale is that a mismatching q-gram indicates that there is no exact match of the q-gram within the τ proximity in the other string, and therefore indicates a high difference in the local contents.

4.2 Tightening the Lower Bound

The above method cannot factor into edit errors made on the right side of the probing window. One way to remedy this is to first collect all the mismatching q-grams of one string in the candidate pair, and compute the minimum

number of edit errors that destroy all the mismatching q-grams on the right side of the probing windows. The latter part can be solved by an algorithm similar to Algorithm 2 with the only difference that it works in the reverse direction on the strings. Since we will perform several probes for a candidate pair, we also build a *condensed suffix sum array* to quickly answer multiple queries.

EXAMPLE 5. *Consider the same string in Example 1. Assume the following q-grams are not matched against a candidate y:*

$$\boxed{(\mathtt{ab}, 1)} \quad \boxed{(\mathtt{ba}, 5)} \quad \boxed{(\mathtt{ab}, 6)}$$

If we run Algorithm 2 in the reverse direction, we can compute the suffix sum array on the left hand side of the figure below:

loc	SumRightErrs		loc	SumRightErrs
6	1		6	1
5	1	$\implies$	1	2
4	1			
3	1			
2	1			
1	2			

SUFFIX SUM ARRAY   CONDENSED SUFFIX SUM ARRAY

The array can be condensed by keep the largest location for a given **SumRightErrs** *value. This results in the* condensed suffix sum array *on the right hand side of the figure above.*

Each entry in the condensed table indicates the total number of errors on the right if the probing window ends before the location field. For example, if our probing window is $[1 \mathinner{.\,.} 2]$, we query the table to find the first entry whose location is larger than 2. If the $\mathcal{L}_1$ distance for the probing windows is 2, then we can infer a lower bound for the edit distance as $2/2 + 1 = 2$.

Algorithm 5: ContentFilter(s, t, Q)

Input : Q is an array
         of mismatching q-grams sorted in increasing order
         of locations; String x and y form a candidate pair.
Output : A lower bound of $2 \cdot ed(s, t)$

1   $j \leftarrow 1; \quad i \leftarrow 2;$
2   **while** $i \le |Q|$ **do**
3   **if** $Q[i].loc - Q[i-1].loc > 1$ **then**
4       $\epsilon = L_1\mathsf{Distance}(s, t, Q[j].loc, Q[i-1].loc + q - 1)$
           $+ \, 2 \cdot \mathsf{SumRightErrs}(Q[i-1].loc + q);$
5   **if** $\epsilon > 2\tau$ **then**
6   **return** $2\tau + 1$ /* early termination here */
7       $j \leftarrow i;$
8     $i \leftarrow i + 1;$
9   **return** $L_1\mathsf{Distance}(s, t, Q[j].loc, Q[i-1].loc + q - 1)$
      $+ \, 2 \cdot \mathsf{SumRightErrs}(Q[i-1].loc + q)$

Algorithm 6: L_1Distance(s, t, lo, hi)

1   $H_s \leftarrow$ frequency histogram for $s[lo \mathinner{.\,.} hi]$;
2   $H_t \leftarrow$ frequency histogram for $t[lo \mathinner{.\,.} hi]$;
3   **return** $\sum_{1 \le i \le |\Sigma|} |H_s[i] - H_t[i]|$

Algorithm 5 implements the tightened content-based mismatch filtering. The heuristic to select the probing window is to use the set of contiguous mismatching q-grams. The algorithm SumRightErrs probes a pre-built condensed suffix sum array to find the minimum number of edit errors to the right of the probing window.

4.3 The Verification Algorithm based on Multiple Filters

Algorithm 7: Verify(x, A)

1   **for each** y *such that* $A[y] = \mathbf{true}$ **do**
2     $(Q, \epsilon_1) \leftarrow \mathsf{CompareQGrams}(x, y);$
      /* count filtering */;
3   **if** $\epsilon_1 \le q \cdot \tau$ **then**
4   Sort Q by the increasing order of locations;
5       $\epsilon_2 \leftarrow \mathsf{MinEditErrors}(Q);$
        /* location-based mismatch filtering */;
6   **if** $\epsilon_2 \le \tau$ **then**
7   Build a condensed suffix sum array for Q;
8         $\epsilon_3 \leftarrow \mathsf{ContentFilter}(str(x), str(y), Q);$
          /* content-based mismatch filtering */;
9   **if** $\epsilon_3 \le 2\tau$ **then**
10           $ed \leftarrow \mathsf{EditDistance}(str(x), str(y));$
11   **if** $ed \le \tau$ **then**
12             $S \leftarrow S \cup (x.id, y.id);$

The complete Verify algorithm used in our Ed-Join algorithm is shown in Algorithm 7. It applies three filtering methods in turn before running the final edit distance calculation: count and position filtering (Lines 2–3), location-based mismatch filtering (Lines 4–6), and content-based mismatch filtering (Lines 7–9).

The CompareQGrams algorithm deserves some discussions. In order to perform count filtering, we need to obtain the total number of mismatching q-grams from the longer string to the shorter one in each candidate pair. We identify a *multiplicity constraint* not mentioned in previous work: each q-gram can have *at most one* match. Ignoring this constraint will lead to double-counting and inflated matching q-gram count.[5] We call q-grams that cannot be matched under the multiplicity constraint *strictly* mismatching q-grams. In contrast, all the other mismatching q-grams we have discussed before are without the multiplicity constraint and are called *loosely* mismatching q-grams. We note that location-based mismatch filtering does *not* work with strictly mismatching q-grams.[6] Therefore, the seemingly complicated CompareQGrams algorithm strives to scan both x and y only once and is still able to compute two outputs: (a) the set of *loosely* mismatching q-grams Q in x, and (b) the total number of strictly mismatching q-grams (i.e., ϵ) in x. ϵ is immediately used in the count filtering (Line 3) and Q is used in the subsequent location-based filtering (Lines 5–6).

5. IMPLEMENTATION DETAILS

---

[5]For example, the SQL implementation in [14] will return a count of 14 for $s = \mathtt{aaaa\underline{w}aaaa\underline{x}}$ and $t = \mathtt{aaaa\underline{y}aaaa\underline{z}}$ and will incorrectly take $\langle x, y \rangle$ as a candidate pair for $\tau = 1, q = 2$.

[6]There are subtle instances where location-based mismatch filtering based on strictly mismatching q-grams will incorrectly prune candidate pairs whose edit distance is within the threshold.

Algorithm 8: CompareQGrams(x, y)

Input : x and y are two sorted q-gram arrays.
Output : Q is the set of *loosely*
        mismatching q-grams from x to y; ϵ is the number
        of *strictly* mismatching q-grams from x to y

1 $i \leftarrow 1;$ $j \leftarrow 1;$
2 $\epsilon = 0;$ $Q \leftarrow \emptyset;$
3 **while** $i \leq |x|$ **and** $j \leq |y|$ **do**
4   **if** $x[i].token = y[j].token$ **then**
5   **if** $|x[i].loc - y[j].loc| \leq \tau$ **then**
6 $i \leftarrow i + 1;$ $j \leftarrow j + 1;$
7   **else**
8   **if** $x[i].loc < y[j].loc$ **then**
9   **if** $x[i].token \neq x[i-1].token$
           or $x[i].token \neq y[j-1].token$
           or $|x[i].loc - y[j-1].loc| > \tau$ **then**
10 $Q \leftarrow Q \cup x[i];$
11 $\epsilon \leftarrow \epsilon + 1;$ $i \leftarrow i + 1;$
12   **else**
13 $j \leftarrow j + 1;$
14   **else**
15   **if** $x[i].token < y[j].token$ **then**
16   **if** $x[i].token \neq x[i-1].token$ **or** $x[i].token \neq$
         $y[j-1].token$ **or** $|x[i].loc - y[j-1].loc| > \tau$ **then**
17 $Q \leftarrow Q \cup x[i];$
18 $\epsilon \leftarrow \epsilon + 1;$ $i \leftarrow i + 1;$
19   **else**
20 $j \leftarrow j + 1;$
21 **while** $i \leq |x|$ **do**
22   **if** $x[i].token \neq x[i-1].token$ **or** $x[i].token \neq y[j-1].token$
     or $|x[i].loc - y[j-1].loc| > \tau$ **then**
23 $Q \leftarrow Q \cup x[i];$
24 $\epsilon \leftarrow \epsilon + 1;$ $i \leftarrow i + 1;$
25 **return** (Q, ϵ)

5.1 Optimizing Index Probing

In Algorithm 4, we probe each q-gram w in x's minimum prefix and try to match w with the positional q-grams returned from the inverted list I_w. This straight-forward implementation is not efficient if the same q-gram appears multiple times in x and/or y (e.g., ba in Example 1). In the worst case, it will probe the inverted index k times and make $O(k^2)$ comparisons, if w appears k times in both x and y. In our implementation, we perform the matching in *blocks*, where each block is a contiguous group of entries for the same q-gram (for x) or a contiguous group of inverted list entries from the same string (for y). The matching is similar to a sort-merge join and we only need to sequentially scan both blocks once. In order not to add the same y to the candidate set more than once, we will skip to the next block *in the inverted list* once a matching q-gram is found.

5.2 q-gram Related Optimizations

A string of length l will generate $l - q + 1$ q-grams. If we store q-grams as it is, i.e., each using q bytes [14], the total size of the q-gram array will be approximately $q + 2$ times of the total size of the string dataset.[7] In addition, the inverted index built during the join will be also large.

We choose to hash q-grams into 4-byte integers[8], an idea

---
[7]We use 2 bytes for each location.

[8]We use the bit-wise hash function designed for strings in [27].

---

also adopted in [2]. Therefore, the total size of the q-grams array is approximately six times of that of the string dataset, *regardless of* the choice of q. Note this does not affect the correctness of the algorithm. Hash collisions only introduce more false positive candidates.

If the edit similarity join is a self join, another simple yet effective optimization is the removal of *widow q-grams*. A widow q-grams has *df* value of 1, i.e., it only appears in one string (even if multiple times), and therefore won't contribute any join result. Widow q-grams can be removed during the preprocessing or simply be ignored when running the Ed-Join algorithm.

5.3 Memory Usage

We measure the memory usage for our Ed-Join algorithm and show the result for the 0.85M-tuple DBLP dataset with $q = 5$ in Table 1.

Table 1: Memory Usage (MB)

Inverted Index	Strings	q-gram Arrays ($= token + loc$)
$\approx 8.5 \cdot \tau$	87.1	$507.9 = 338.6 + 169.3$

We can see that the memory used by the inverted index increases almost linearly with the threshold τ (when τ is small). This is because of the use of minimum prefix length in the Ed-Join algorithm. As will be discussed in Section 6.3, the size of the inverted index for the All-Pairs-Ed algorithm, which does not use minimum prefix, could be up to 10 times higher than that of the Ed-Join algorithm. Note that even though q-gram arrays are large, they do not necessarily need to be loaded entirely into main memory. Thanks to the length filtering (Line 8 in Algorithm 1), we can safely discard q-gram arrays whose lengths are smaller than $l_x - \tau$, where l_x is the length of the current q-gram array (i.e., x). Even in the worst case when all strings are of the same length, the algorithm can switch back to a "block-nested-loop-join" mode, where the input strings are divided into memory-size partitions, and one partition is joined with the entire datasets at a time. This is similar to how the All-Pairs algorithm copes with out-of-core datasets [3].

As a future work, we will explore compression techniques as are often used in search engines [35].

6. EXPERIMENTS

In this section, we report experimental results and our analyses.

6.1 Experiment Setup

The following algorithms are used in the experiment.

All-Pairs-Ed is a prefix-filtering-based algorithm adapted from All-Pairs algorithm for edit similarity join [3]. It serves as a baseline algorithm in the experiment.

PartEnum is a similarity join algorithm based on the Pigeon-Hole principle [2]. It computes and divides a feature vector based on q-grams into n_1 partitions and generates several hash signatures for each partitions using an enumeration scheme; String pairs that share a common signature are recognized as candidate pairs. The Flamingo Project[9]

---
[9]http://flamingo.ics.uci.edu/

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). Section 6.4 is dedicated to the comparison of Ed-Join and PartEnum.

Ed-Join and **Ed-Join-l** . Ed-Join is our proposed algorithm, with both location-based and content-based mismatch filtering. In order to measure the effects of each filter, we remove the content-based mismatch filtering from Ed-Join and we name the resulting algorithm Ed-Join-l.

All algorithms are implemented as in-memory algorithms, with all their inputs loaded into the memory before they were run.

In [3], All-Pairs is shown to consistently outperform alternative algorithms such as ProbeCount-Sort [30], and LSH [13], and therefore we don't consider them.

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 4.1.2 with -O3 flag.

We used several publicly available real datasets in the experiment. They were selected to cover a wide range of data distributions (See Table 2) and application domains.

DBLP This dataset is a snapshot of the bibliography records from the DBLP Web site.[10] It has about 900K records; each record is a concatenation of author name(s) and the title of a publication. DBLP dataset is widely used in similarity join and near-duplicate detection research [2, 3, 21, 20, 34].

TEXAS is a text dump of the Broker and Sales licensees database from the Texas Real Estate Commission.[11] The file contains over 150K records; each is a concatenation of 19 attributes, including person names, addresses, and licence information. This dataset was also used in [21].

TREC is from TREC-9 Filtering Track Collections[12]. It contains 350K references from the MEDLINE database. We extract and concatenate author, title, and abstract fields.

UNIREF denotes the UniRef90 protein sequence data from the UniProt project.[13] We extract the first 500K protein sequences; each sequence is an array of amino acids coded as uppercase letters.

These datasets are transformed and cleaned as follows:

1. We converted white spaces and punctuations into underscores, and letters into their lowercases for TEXAS and TREC. UNIREF is already a clean dataset. In order to study the effect of large alphabet size, we choose not to perform case conversion on DBLP.
2. We then removed exact duplicates.
3. We also removed the strings whose length is smaller than a threshold. This is mainly because count filtering requires strings longer than $q \cdot (\tau + 1)$.[14] Since we use

---

[10] http://www.informatik.uni-trier.de/~ley/db
[11] http://www.trec.state.tx.us/LicenseeDataDownloads/ trecfile.txt (downloaded in Oct, 2007)
[12] http://trec.nist.gov/data/t9_filtering.html
[13] http://beta.uniprot.org/ (downloaded in March, 2008)
[14] See [15] for further discussions.

---

different maximum edit distance thresholds for different datasets, we choose minimum string length thresholds as follows: 20 for DBLP and TEXAS, 100 for TREC, and 180 for UNIREF.

4. We then sort the strings into increasing order of length.

Some important statistics about the cleaned datasets are listed in Table 2. Distributions of q-gram frequency and string length are plotted in Figures 3(a)–3(c). q-gram frequency distributions for all datasets follow approximately the Zipf distribution and thus only the plot on DBLP is shown. It is interesting to observe that the length distributions for the four datasets are all distinct.

Table 2: Datasets Statistics

Dataset	N	avg_len	$|\Sigma|$	Comment
DBLP	863,171	104.8	93	author, title
TEXAS	154,987	112.1	37	name, address, licence#
TREC	271,464	1098.4	37	author, title, abstract
UNIREF	365,996	465.1	25	protein sequences

We mainly measure (a) size of the candidate pairs before calling the Verify algorithm (denoted as **CAND-1**)— these are candidate pairs that pass the location-based mismatch filtering for Ed-Join and Ed-Join-l or candidate pairs that pass the prefix filtering for All-Pairs-Ed; (b) size of candidate pairs before the final edit distance verification (denoted as **CAND-2**); and (c) the running time. The running time does not include preprocessing time or loading time of the q-gram arrays, as they are the same for all algorithms.[15]

6.2 Effect of q-gram Length

We ran Ed-Join algorithm with varying q. Figures 4(a)–4(c) show CAND-1 and CAND-2 sizes and the running time for Ed-Join on the TREC dataset for $\tau \in [2, 10]$. Results on other datasets or algorithms are similar.

With respect to the running time, we observe that (a) the best q under this setting is 8. $q = 10$ is a close runner-up for $\tau \in [2, 8]$, but no longer competitive for larger thresholds. (b) the running time for $q = 2$ is by far the worst for all thresholds. Its running time is usually an order of magnitude slower than the running time with the best q value. (c) for a fixed τ, the general trend is that the running time will first decrease and then increase when we move towards a large q value.

The main reason for the reduction in running time when q increase to its optimal value is due to the reduction of CAND-1 sizes (See Figures 4(a)). E.g., the ratio of candidate sizes between $q = 2$ and $q = 8$ is 164.4 when $\tau = 10$. The effect of CAND-1 size reduction gradually subsides with the increase of q. When $q = 10$, the CAND-1 size actually increases.

There are several factors contributing to the trends of CAND-1 size. First, small q means a small q-gram domain, where the inverted list of the rarest q-gram is still fairly long. Hence a large number of candidate pairs will be generated. This explains the rapid reduction of CAND-1 size when we move away from $q = 2$. Second, a larger q value means the average minimum prefix is longer and hence more inverted lists will be probed and candidates generated. Third, hash collision is more serious when q increases. The last two fac-

---

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

(a) q-gram Frequency Distribution, DBLP (b) String Length Distribution, DBLP (in red) (c) String Length Distribution, TREC (in red) and TEXAS (in blue) and UNIREF (in blue)
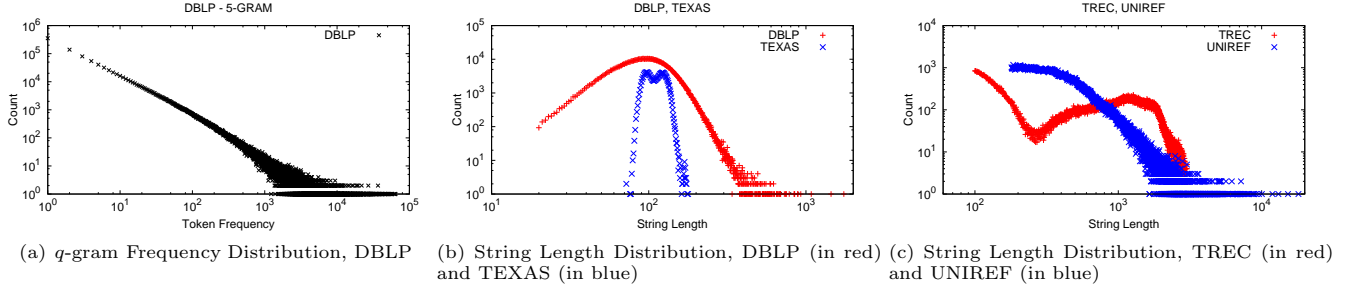
Figure 3: Statistics and Distributions of Datasets

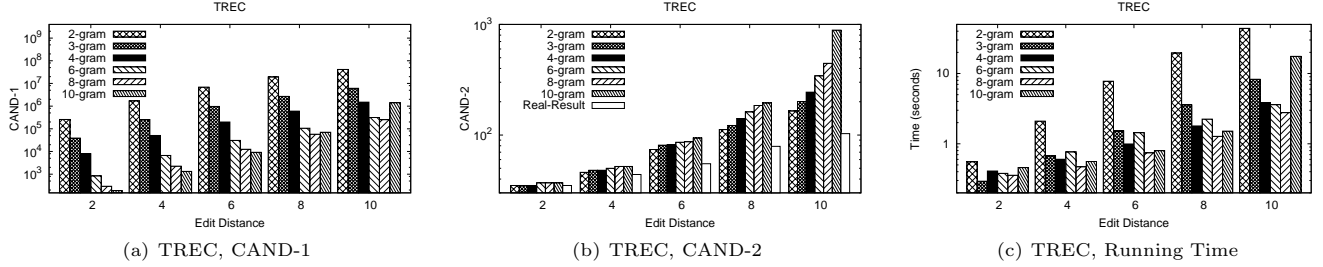(a) TREC, CAND-1 (b) TREC, CAND-2 (c) TREC, Running Time

Figure 4: Varying q-gram Length

tors together contributed to the increase of CAND-1 size for $q = 10$ and $\tau = 10$.

Another contributing factor to the running time is the time spent on the edit distance calculation. Since a large q value reduces the effectiveness of count filtering, we observe a steady increase in the CAND-2 sizes when q increases.

In the rest of the experiment, we use the most time-efficient q for each dataset.

6.3 Effect of Filters

We run the All-Pairs-Ed, Ed-Join-l, and Ed-Join algorithms on DBLP, TEXAS, and TREC datasets to measure the effectiveness of location-based and content-based mismatch filtering techniques.

Location-Based Mismatch Filtering. Figures 5(a)–5(c) show the average prefix lengths for All-Pairs-Ed and Ed-Join-l on three datasets. Note that Ed-Join has the same prefix length as Ed-Join-l. We can see that prefix length is reduced by 1/2 to 2/3 from Ed-Join-l to All-Pairs-Ed due to the use of location-based mismatch filtering. The prefix lengths for both algorithms grow linearly with the threshold τ. For All-Pairs-Ed, this is expected as the length is exactly $q \cdot \tau + 1$. The linear growth of minimum prefix length in Ed-Join-l showcases the inherent redundancy in the prefix and the wide applicability of this optimization.

The reduction of prefix length affects both the index size and the CAND-1 size (See Figures 5(d)–5(i)). The impact is even more significant. For example, a 90% reduction of index size and a 72% reduction of CAND-1 size are achieved on TREC dataset by Ed-Join-l with $\tau = 10$, while the reduction of prefix length is about 66%. This is because of the Zipf distribution of q-grams.

Content-Based Mismatch Filtering. Figures 5(j)–5(l) show the CAND-2 sizes by different algorithms. Note that Ed-Join-l has the same CAND-2 size as All-Pairs-Ed. We also show the size of the real join results, which lower bounds the CAND-2 size.

The following observations can be made.

1. The difference of CAND-2 sizes between All-Pairs-Ed and real result size grows exponentially with the increase of τ. This is primarily because the count filtering is less effective at high error thresholds.

2. Content-base mismatch filtering effectively reduces the CAND-2 size. About 76%, 18%, and 98% of candidate pairs are pruned by content-based mismatch filtering at the largest τ tested on DBLP, TEXAS, and TREC, respectively. The reduction effect is generally more substantial with large τ.

3. The reduction rate of CAND-2 size by Ed-Join on TEXAS is not as significant as that of the other two datasets. This is because the majority (70%) of the candidate pairs returned by count filtering are real join results. In contrast, the numbers are 21% and 0.5% for DBLP and TREC, respectively.
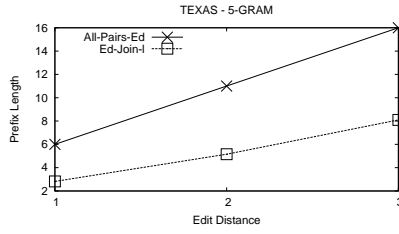
Running Time. We show the running times for all three algorithms on three datasets in Figures 5(m)–5(o). For all settings, Ed-Join is the most efficient algorithm, followed by Ed-Join-l. Both algorithms outperform All-Pairs-Ed algorithm by a large margin and the margin increases rapidly with the increase of error threshold τ. When τ increases, the CAND-1 and CAND-2 sizes for All-Pairs-Ed will both grow due to using longer prefix and the less effectiveness of count filtering. Our location-based and content-based mismatch filtering effectively alleviate each of these issues and thus Ed-Join can still attain good performance.

The speed-ups of Ed-Join and Ed-Join-l against All-Pairs-Ed are in the range of 2.2x to 5x. The speed-up is especially significant on TREC. This can be explained as (a) content-based mismatch filtering helps to remove a 99.5% of the candidate pairs generated by count filtering; and (b) the overhead for edit distance verification is more significant due to the long average length of the strings in the TREC dataset.
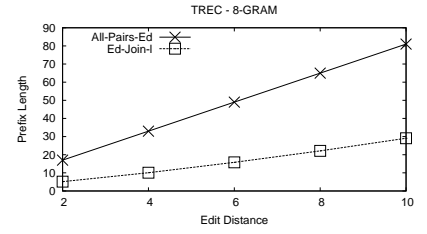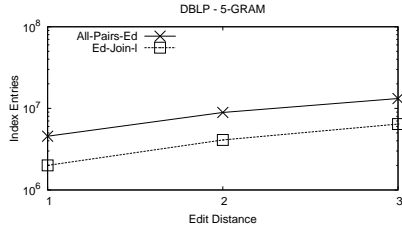
6.4 Comparison with PartEnum
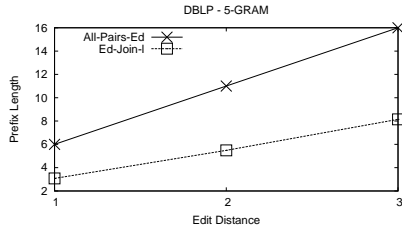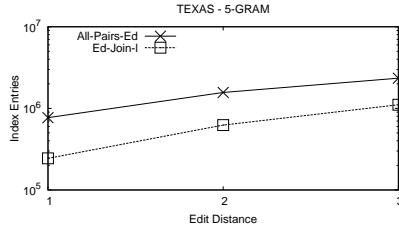
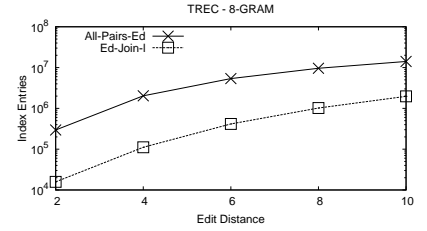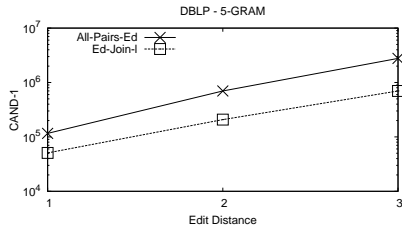(a) DBLP, 5-Gram, Prefix Length (b) TEXAS, 5-Gram, Prefix Length (c) TREC, 8-Gram, Prefix Length
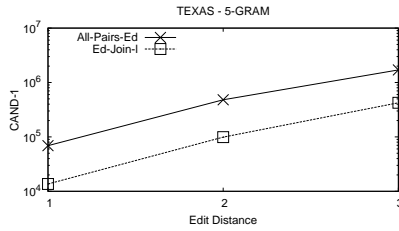
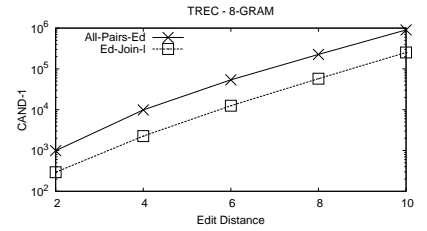(d) DBLP, 5-Gram, Index Entries (e) TEXAS, 5-Gram, Index Entries (f) TREC, 8-Gram, Index Entries
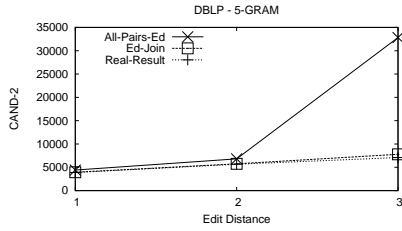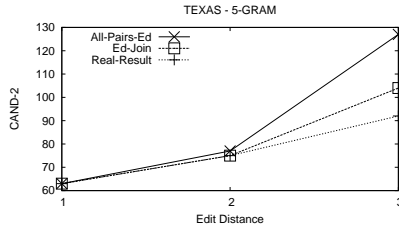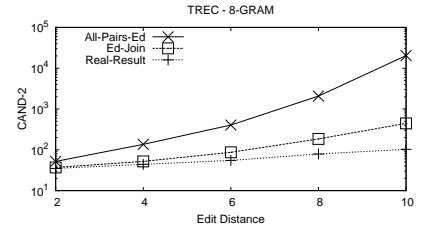
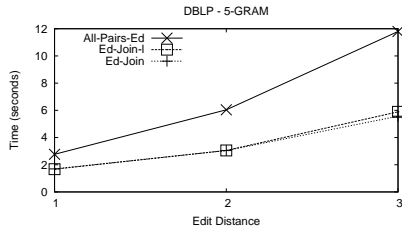(g) DBLP, 5-Gram, CAND-1 (h) TEXAS, 5-Gram, CAND-1 (i) TREC, 8-Gram, CAND-1
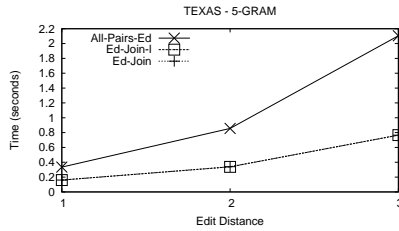
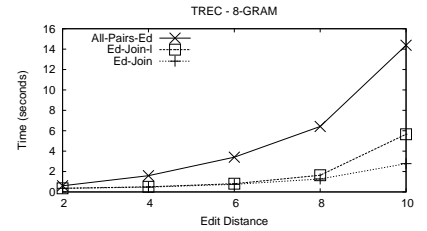(j) DBLP, 5-Gram, CAND-2 (k) TEXAS, 5-Gram, CAND-2 (l) TREC, 8-Gram, CAND-2
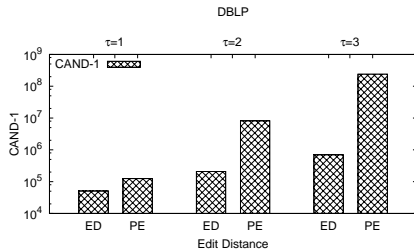
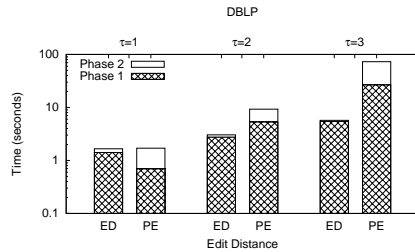(m) DBLP, 5-Gram, Running Time (n) TEXAS, 5-Gram, Running Time (o) TREC, 8-Gram, Running Time
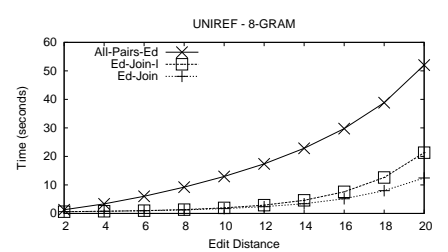
Figure 5: Experiment Results I

(a) DBLP, Candidate Size (b) DBLP, Running Time (c) UNIREF, Running Time

Figure 6: Experiment Results II

We compare our Ed-Join algorithm with PartEnum algorithm on all datasets and report the results of the DBLP dataset here. We run both algorithms in their optimal parameter settings.

- for Ed-Join, we use $q = 5$.
- for PartEnum, we fix $q = 1$ according to [2] and then manually tune its parameters n_1 and n_2. The best performance is obtained by using $n_1 = 3, n_2 = 4$ for $\tau = 1$ and using $n_1 = 3, n_2 = 7$ for $\tau = 2$ or $\tau = 3$.

The running time of PartEnum does not include its signature generation time.

Figure 6(a) shows the candidate size produced by two algorithms. We measure the candidate size generated by location-based mismatch filtering for Ed-Join (i.e., CAND-1). The candidate of PartEnum is generated by signature set intersection. The candidate size of PartEnum grows much more rapidly than Ed-Join when edit distance threshold increases.

Figure 6(b) plots the running times of the algorithms decomposed into two phases. Phase 1 is the time taken to generate candidate pairs and phase 2 is the time taken to remove false positives and obtain the final answers. We can see that PartEnum is only competitive for $\tau = 1$ while Ed-Join outperforms PartEnum for all τ values. For $\tau = 3$, Ed-Join achieves 13x speed-up against PartEnum. There are at least two reasons for the slow down of PartEnum: it generates a much larger candidate set and it directly verifies the edit distance for all candidates without further filtering.

6.5 Large Edit Error Thresholds

We measure the performance of the edit similarity join algorithms with respect to large error threshold on the UNIREF dataset. Note that large threshold is often used when searching or clustering similar protein sequences. The results for τ up to 20 are shown in Figure 6(c).

The running time for all three algorithms increases rapidly with τ. It is clear that Ed-Join outperforms Ed-Join-l. Both are superior to All-Pairs-Ed by a large margin, especially when τ is large. The result shows that Ed-Join is the algorithm of choice for large thresholds.

6.6 Discussion of Our Findings

Summarizing our experimental results, we have the following two findings:

1. The choice of q (i.e., q-gram length) is important to the performance of the edit similarity join. Very small or large q values usually give worst performance.[16]
2. Large-scale edit similarity join with medium range edit distance threshold ($\tau \in [4, 20]$) can be efficiently computed in reasonable amount of time.

The first finding deserves some discussion. [14] tested $q \in [2, 5]$ and recommended $q = 2$ or $q = 3$ for best performance and $q = 2$ for minimum space overhead. In our experiments, small q values, however, always result in the worst performance.

We note that

1. [14]'s implementation is on top of a relational DBMS and implements the three filtering techniques using a non-equijoin followed by HAVING constraints. We conjecture

---

[16]$q \in [4, 8]$ is probably a good choice for datasets of similar nature to those used in our experiments.

that the DBMS might not be able to fully optimize the execution of this complex query and the time to generate all the candidates will be approximately the same for different q values.

2. Count filtering is less effective for large q, which results in more verification by the expensive edit distance UDF function.

Our method prefers a larger q because

1. Our Ed-Join algorithm is based on the prefix filtering, which does not work well for small q as it will result in a small alphabet.[17]
2. The location-based mismatch filtering effectively reduces the growth of prefix length, while the additional content-based mismatch filtering reduces the amount of edit distance verification.

Therefore, we've found the best q for a dataset is usually larger than 3 (e.g., $q = 8$ for TREC and $q = 5$ for DBLP). The best q value also varies with the dataset and edit distance threshold and currently can only be determined experimentally.

We note that this conclusion also holds for All-Pairs-Ed, a standalone implementation of the existing prefix filtering and count filtering. Interesting, the best q for All-Pairs-Ed is usually smaller, e.g., $q = 4$ is best for DBLP. This is mainly because it does not have any additional means to alleviate the adverse effect bought by a large q value.

A possible issue with using long q-gram is that short strings whose length is no greater than $\xi = q \cdot (\tau + 1)$ cannot be processed. We can select the subset of data whose string length is no greater than $\xi + \tau$ and use, say, bi-grams to find out *meaningful* edit similarity join results that involve at least one string whose length is within $[2(\tau + 1), \xi]$ [15].

7. RELATED WORK

Similarity join has been studied by many researchers due to its importance in many applications, including record linkage [33], merge-purge [19], data deduplication [29], and name matching [4].

In the early stage, only binary similarity functions for sets were studied, including set containment [17, 28, 24], set equality and non-empty set intersection [22]. More general similarity/distance functions were later considered [14, 30, 12, 2, 3, 34], including edit distance, set overlap, Jaccard similarity, cosine similarity, Hamming distance and their variants. Similarity join in Euclidean space has also been studied (e.g., [5]). [10] considered similarity joins with multiple conjunctive similarity predicates. [8] compared a large number of similarity functions experimentally with an emphasis on their performance and accuracy.

Another line of work is to solve the similarity join problem approximately. Locality Sensitive Hashing (LSH) [13] is a widely adopted for technique for nearest neighbor search and can be adapted to similarity join [2, 3]. [6] proposed a shingle-based approach to approximately identify near duplicate Web pages. Another synopsis-based algorithm was proposed [9] based on randomized projection.

Similarity join on strings is also closely related to approximate string matching, an extensively studied topic in algorithm and pattern matching communities. Due to the large amount of related literature, we refer readers to [26] and [16].

---

[17]First observed in [2].

Edit distance is a common distance function for strings. It can be computed in $O(n^2)$ time and $O(n)$ space using the standard dynamic programming [32]. The time complexity of calculating edit distance can be reduced from $O(n^2)$ to $O(n^2/\log n)$ using the Four-Russians technique [23] or $O(n^2/w)$ using bit-parallel algorithm [25]. [31] provides a worst case $O(nd)$-time, average case $O(n + d^2)$-time algorithm for edit distance, where d is the edit distance between the two strings.

q-grams are widely used for approximate string match. It is especially useful for edit distance constraints due to its ability to lower bound the edit distance. Gapped q-gram is shown to have better filtering powers than standard q-gram, but is only suitable for edit distance threshold of 1 [7]. Recently, a variable length q-gram was proposed in [21] and was shown to speed up many computation tasks originally based on q-gram.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we study new algorithms for the edit similarity join problem. Unlike previous approaches that pose constraints on the number of matching q-grams, we develop two new filtering methods by analyzing the locations and contents of mismatching q-grams. The two new filters are combined with existing filters in our Ed-Join algorithm. Extensive experiments on large-scale real datasets demonstrate that the new Ed-Join algorithm outperforms alternative methods. We also report several interesting findings, including the recommendation of using longer q-grams for implementing the edit similarity join algorithm.

Our future work includes implementing our proposed algorithm on top of a RDBMS. A key issue is how to optimize our algorithm to leverage the existing set- and group-based query processing capabilities of RDBMSs. Another future work is to explore compression-based techniques to reduce the memory footprint of the algorithm.

REFERENCES

[1] A. Andoni, M. Deza, A. Gupta, P. Indyk, and S. Raskhodnikova. Lower bounds for embedding edit distance into normed spaces. In *SODA*, pages 523–526, 2003.

[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] M. Bilenko, R. J. Mooney, W. W. Cohen, P. Ravikumar, and S. E. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Sys.*, 18(5):16–23, 2003.

[5] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *SIGMOD*, pages 379–388, 2001.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[7] S. Burkhardt and J. Kärkkäinen. One-gapped q-gram filters for Levenshtein distance. In *CPM*, pages 225–234, 2002.

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

[9] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[10] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries.

[11] S. Chaudhuri, V. Ganti, and R. Kaushik. Data debugger: An operator-centric approach for data quality solutions. *IEEE Data Eng. Bull.*, 29(2):60–66, 2006.

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

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[14] 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.

[15] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free (erratum). Technical Report CUCS-011-03, Columbia University, 2003.

[16] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Computer Science and Computational Biology. Cambridge University Press, 1997.

[17] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *VLDB*, pages 386–395, 1997.

[18] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, 2006.

[19] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[20] 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.

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

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

[23] W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[24] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28:56–99, 2003.

[25] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999.

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

[27] M. V. Ramakrishna and J. Zobel. Performance in practice of string hashing functions. In *DASFAA*, pages 215–224, 1997.

[28] 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.

[29] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD*, 2002.

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

[31] E. Ukkonen. On approximate string matching. In *FCT*, 1983.

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

[33] W. E. Winkler. The state of record linkage and current research problems. Technical report, U.S. Bureau of the Census, 1999.

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

[35] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

In *VLDB*, pages 327–338, 2007.