

# MinJoin: Efficient Edit Similarity Joins via Local Hash Minima\*

Haoyu Zhang<sup>†</sup> and Qin Zhang<sup>‡</sup>

Department of Computer Science, Indiana University

## Abstract

We study the problem of computing similarity joins under edit distance on a set of strings. Edit similarity joins is a fundamental problem in databases, data mining and bioinformatics. It finds important applications in data cleaning and integration, collaborative filtering, genome sequence assembly, etc. This problem has attracted significant attention in the past two decades. However, all previous algorithms either cannot scale well to long strings and large similarity thresholds, or suffer from imperfect accuracy.

In this paper we propose a new algorithm for edit similarity joins using a novel string partition based approach. We show mathematically that with high probability our algorithm achieves a perfect accuracy, and runs in linear time plus a data-dependent verification step. Experiments on real world datasets show that our algorithm significantly outperforms the state-of-the-art algorithms for edit similarity joins, and achieves perfect accuracy on all the datasets that we have tested.

## 1 Introduction

Edit similarity joins is a fundamental problem in the database and data mining literature, and finds numerous applications in data cleaning and integration, collaborative filtering, genome sequence assembly, etc. In this problem we are given a set of strings  $\{s_1, \dots, s_n\}$  and a distance threshold  $K$ , and asked to output all pairs of strings  $(s_i, s_j)$  such that  $\text{ED}(s_i, s_j) \leq K$ , where  $\text{ED}(\cdot, \cdot)$  is the edit distance function, which is defined to be the minimum number of insertions, deletions and substitutions to transfer one string to another. There is a long line of research on edit similarity joins [5, 1, 2, 3, 7, 16, 13, 9, 15, 8, 14].

A major challenge for most existing algorithms, as pointed out by the recent work [17], is that they do not scale well to long strings and large edit thresholds. Long strings and large thresholds are critical for applications involving long sequence data such as big documents and DNA sequences, where a small threshold  $K$  may just give zero output. For example, in the genome sequence assembly, in which the first step is to find all pairs of similar reads under edit distance, the third generation sequencing technology such as single molecule real time sequencing (SMRT) [10] generates reads of 1,000-100,000 bps long with 12-18% sequencing errors (i.e., percentage of insertions,

---

\*Authors are supported in part by NSF CCF-1525024, IIS-1633215 and CCF-1844234.

<sup>†</sup>Email: hz30@umail.iu.edu

<sup>‡</sup>Email: qzhangcs@indiana.edu

deletions and substitutions). Large threshold is also identified as the main challenge in a recent string similarity search/join competition [12], where it was reported that “an error rate of 20%-25% pushes todays techniques to the limit”.

Different from previous algorithms which are deterministic and return the exact answers, in [17] the authors proposed a randomized algorithm named `EmbedJoin` which is more efficient on long strings and large thresholds. However, the accuracy (more precisely, the *recall*, i.e., the number of pairs found by the algorithm divided by the total number of similar pairs; the *precision* of all algorithms discussed in this paper is always 100%) of `EmbedJoin` is only 95% - 99% on a number of real-world datasets tested in [17]. The imperfect accuracy is inherent to `EmbedJoin` which we shall explain shortly. The main question we are going to address in this paper is:

*Can we solve edit similarity joins efficiently on long string and large edit threshold while achieving perfect accuracy with a good probability?*

**Our Contribution.** We propose a novel randomized algorithm named `MinJoin` to address the above question. The high level framework of `MinJoin` is simple: it first partitions each string into a set of substrings, and then uses hash join on these substrings to find all pairs of strings that share at least one common substring. At the end a verification step is used to remove all false positives. Our string partition scheme works as follows: We first assign each letter  $\alpha$  in the string  $s$  a value, which is a random hash value of the  $q$ -gram ( $q$  is a value determined by the string length, the threshold  $K$ , and the size of the alphabet) starting from  $\alpha$ . We then determine the *anchors* of string  $s$  using the following strategy: a letter  $\alpha$  is an anchor if and only if its value is the smallest among all letters in a certain neighborhood of  $\alpha$ . At the end we simply partition  $s$  at all of its anchors.

Via a rigorous mathematical analysis we can show that under our partition scheme, with a good probability, any pair of strings with edit distance at most  $K$  will share at least one common partition. We can also show that this partition procedure runs in *linear* time.

We have verified the effectiveness of `MinJoin` by an extensive set of experiments. Though in our experiments we do not include a parallel repetition step which is for the purpose of guaranteeing that our algorithm achieves perfect accuracy with high probability in theory (see the discussion in Section 2.2), our experimental results show that `MinJoin` is able to achieve perfect accuracy on all datasets that were used in [17]. Moreover, `MinJoin` is faster than all existing exact (deterministic) algorithms by orders of magnitudes on datasets of long strings and large edit thresholds, and is also faster than `EmbedJoin` by a good margin.

**Previous Work and Comparisons.** Many of the existing algorithms on edit similarity joins also follow the string partition framework. The performance of the algorithm is largely determined by the number of partitions generated for each string, and the number of queries made to the indices (e.g., hash tables) to search for similar strings.

We discuss several state-of-the-art algorithms according to the experimental studies in [6].

`QChunk` [9] is an exact edit similarity join algorithm based on string partition. `QChunk` first obtains a global order  $\sigma$  of  $q$ -grams. It then partitions each string into a set of chunks with starting positions  $1, q + 1, 2q + 1, \dots$ , and stores the first  $K + 1$  chunks (according to the order  $\sigma$ ) in a hash table. Next, for each string the algorithm queries the hash table with the string’s first  $N - (\lceil (N - K)/q \rceil - K) + 1$   $q$ -grams according to  $\sigma$  to check if there is any match, where  $N$  is the

string length.<sup>1</sup>

**PassJoin** [8] is another exact algorithm based on string partition. The algorithm partitions each string  $s$  into  $K + 1$  equal-length segments, and records the  $i$ -th segment into an inverted index  $L_{|s|}^i$ . Next, for each string the algorithm queries some of the inverted indices to find similar strings; the number of queries made for each string is  $\Theta(K^3)$ , which is  $\Theta(N^3)$  when  $K$  is a fixed percentage of  $N$ .

**VChunk** [15] is the one that is closest to **MinJoin** among all algorithms that we are aware of. In **VChunk** each string is partitioned into at least  $2K + 1$  chunks of possibly different lengths,

determined by a *chunk boundary dictionary (CBD)*. More precisely, each string is cut at positions of appearances of each word in CBD to obtain its chunks. The CBD is data dependent and the optimal one is NP-hard to compute. In [15] the authors proposed a greedy algorithm for computing a CBD in time  $O(n^2N^2/K)$ , where  $n$  is the number of input strings, and  $N$  is the maximum string length.

The recently proposed algorithm **EmbedJoin** [17] uses a very different approach. **EmbedJoin** first embeds each string from the edit distance metric space to the Hamming distance metric space, translating the original problem to finding all pairs of strings that are close under Hamming distance. It then uses Locality Sensitive Hashing to compute (approximate) similarity joins in the Hamming space. However, the embedding algorithm employed by **EmbedJoin** has a worst case distance distortion  $K$ , which can be very large. Although in practice the distortion is much smaller, it still contributes a non-negligible percentage of false negatives which prevent a perfect accuracy.

Compared with these existing algorithms, **MinJoin** has the following major advantages.

- For each string **MinJoin** only generates  $O(K)$  partitions, and makes the same amount of queries (for searching similar strings), which are significantly smaller than **QChunk** and **PassJoin**.
- **MinJoin** can compute partitions of all strings in time  $O(nN)$ , i.e., linear in the input size, which is even faster than the computation of CBD in **VChunk**.
- **MinJoin** is able to reach perfect accuracy on tested datasets, compared with 95%-99% of **EmbedJoin**.

**A Comparison with MinHash Based Approach.** We would like to note that **MinJoin** is quite different from the folklore algorithm using MinHash, in which for each string we collect all its  $q$ -grams and hash them to numbers, and then pick the one with the smallest hash value as the signature for the subsequent hash join; to increase the accuracy we can pick multiple signatures using different hash functions for each string.

To see the difference, in **MinJoin** the hash values of the  $q$ -grams are used to partition a string to substrings/signatures, while in the MinHash based approach the  $q$ -grams are the signatures themselves. In **MinJoin** we set  $q$  to be a small number (more precisely,  $q = \Theta(\log_{|\Sigma|}(N/K))$  where  $\Sigma$  is the alphabet of the string) in order to make all  $q$ -grams distinct in every small neighborhood of the string. And one partition will give us all the signatures of the string. While in the MinHash based approach, it is not clear how to find the best combination of the value  $q$  and the number of signatures (or, hash functions) to use, for the purpose of achieving a perfect accuracy under a small running time. We are not aware of any theory for guiding the choices of  $q$  and the number

---

<sup>1</sup>Alternatively, for each string we can store the first  $N - (\lceil (N - K)/q \rceil - K) + 1$   $q$ -grams in the hash table, and make queries with the first  $K + 1$  chunks.

Notation	Definition
$[n]$	$[n] = \{1, 2, \dots, n\}$
$K$	edit distance threshold
$\mathcal{S}$	set of input strings
$s_i$	$i$ -th string in $\mathcal{S}$
$n$	number of input strings, i.e., $n =  \mathcal{S} $
$ s $	length of string $s$
$s_{i..j}$	substring of $s$ starting from the $i$ -th letter to the $j$ -th letter
$N$	maximum string length
$\Sigma$	alphabet of strings in $\mathcal{S}$
$q$	length of $q$ -gram
$\Pi$	random hash function $\Sigma^q \rightarrow (0, 1)$
$T$	number of targeted partitions; $T = \Theta(K)$
$r$	radius for computing local minimum

Table 1: Summary of Notations

of signatures in the MinHash based approach for edit similarity joins. In Section 4.3 we will show experimentally that **MinJoin** significantly performs the MinHash based approach in both accuracy and running time.

**More Related Work.** There is a large body of work on similarity joins under edit distance. A large number of the existing algorithms fall into the category called the *signature-based* approach, in which we compute for each string a set of signatures, and then apply various filtering methods to those signatures to select a set of candidate pairs for verification. All the string partition based algorithms that we have discussed can be thought as special cases of the signature-based approach. Other algorithms in this category include **GramCount** [5], **AllPair** [2], **FastSS** [3], **ListMerger** [7], **EDJoin** [16], and **AdaptJoin** [14].

There are a few algorithms that use different approaches, including the embedding-based algorithm **EmbedJoin** discussed previously, the tree-based algorithm **M-Tree** [4], the enumeration-based algorithm **PartEnum** [1], and the trie-based algorithm **TrieJoin** [13]. However, except **EmbedJoin**, others' performance is not as good as the best partition-based approaches.

**Notations.** We have listed a set of notations to be used in this paper in Table 1.

## 2 A String Partition Scheme Using Local Hash Minima

In this section we present the string partition algorithm and analyze its properties.

### 2.1 The Algorithm

We start by giving some high level ideas of our partition scheme. As mentioned, in **MinJoin** we first partition each string to a set of substrings, and then find pairs of strings that share at least one common partition as candidates for verification. Consider a pair of strings  $x$  and  $y$  ( $|x| = |y| = N$ ) with edit distance  $k$ . Let  $\rho : [N] \rightarrow [N] \cup \{\perp\}$  be the *optimal* alignment between  $x$  and  $y$ ,

---

**Algorithm 1** Partition-String ( $s, T, \Pi$ )

---

**Input:** Input string  $s$ , number of targeted partitions  $T$ , random hash function  $\Pi : \Sigma^q \rightarrow (0, 1)$   
**Output:** Partitions of  $s$ :  $\mathcal{P} = \{(pos, len)\}$ , where  $(pos, len)$  refers a substring of  $s$  starting at the  $pos$ -th position with length  $len$

- 1:  $\mathcal{P} \leftarrow \emptyset$
- 2:  $A = \{a_1, \dots, a_p\} \leftarrow \text{Find-Anchor}(s, T, \Pi)$
- 3: **for each**  $i \in [1, p - 1]$  **do**
- 4:    $\mathcal{P} \leftarrow \mathcal{P} \cup (a_p, a_{p+1} - a_p)$
- 5: **end for**

---

where  $\rho(i) = j \in [N]$  means that either  $x[i] = y[j]$  or  $x[i]$  is substituted by  $y[j]$  in the optimal transformation, and  $\rho(i) = \perp$  means that  $x[i]$  is deleted in the optimal transformation. If we pick any  $k$  indices  $1 < i_1 < \dots < i_k < N$  such that  $\rho(i_\ell) \neq \perp$  ( $\ell \in [k]$ ), partition  $x$  at indices  $i_1, \dots, i_k$  to  $k + 1$  substrings, and partition  $y$  at indices  $\rho(i_1), \dots, \rho(i_k)$  to  $k + 1$  substrings, then by the pigeonhole principle  $x$  and  $y$  must share at least one common partition.

Of course obtaining an optimal alignment between  $x$  and  $y$  before the partition is unrealistic. Our goal is to partition each string independently, while still guarantee that with a good probability, any pair of similar strings will share at least one common partition.

We present our partition algorithm in Algorithm 1 and Algorithm 2. Let us briefly describe them in words. Algorithm 1 first calls Algorithm 2 to obtain all *anchors* (to be defined shortly) of the input string  $s$ , and then cuts  $s$  at each anchor into a set of substrings. To compute all anchors, Algorithm 2 first hashes all the substrings of  $s$  of length  $q$  (i.e.,  $s[1..q], s[2..q+1], \dots$ ) into values in  $(0, 1)$ . Now we have effectively transferred  $s$  to an array  $h[]$  of size  $|s| - q + 1$ , with each coordinate taking a value in  $(0, 1)$ . We call a coordinate  $i$  in  $h[]$  a *local minimum* if its value is strictly smaller than all other coordinates within a distance  $r$  of  $i$  (for a pre-specified parameter  $r$ , call it the *neighborhood size*). Algorithm 2 outputs the corresponding  $i$ -th letter in string  $s$  as an anchor. For convenience, in the rest of the paper we also call a local minimum coordinate in  $h[]$  an anchor.

We will show that for a pair of strings  $x, y$ , if they share a common substring  $\sigma$  that is long enough, then there must be at least two letters  $u, v$  in  $\sigma$  such that  $u$  and  $v$  are two adjacent anchors in both  $x$  and  $y$ , which means that if we use anchors to partition  $x$  and  $y$ , then they must share at least one common partition. On the other hand, we know that for two strings of length  $N$  and edit distance at most  $K$ , they must share at least one common substring of length  $(N - K)/(K + 1)$ . Thus by properly choosing the neighborhood size  $r$  (as a function of the string length and the number of targeted substrings  $T$ ), we can guarantee that two similar strings will share at least one common partition.

**A Running Example..** Before analyzing Algorithm 1 we first give a running example. Table 2 presents the hash values of all 3-grams in  $\mathcal{S}$  under the hash function  $\Pi$ . Table 3 presents a collection of input strings  $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5\}$  and their lengths. We want to find all pairs of strings with edit distance less than or equal to  $K = 4$ . Table 4 presents the partitions of strings obtained by Algorithm 2 under parameter  $T = 3$ . We also calculate the neighborhood size  $r$  for each string based on its string length and the parameter  $T$ .

Considering string  $s_1$  as an example, its 6-th 3-gram “CTA” has a smaller hash value than all its neighbors within distance  $r = 2$  (i.e., “TGC”, “GCT”, “TAA”, “AAC”). Thus “CTA” is selected as

---

**Algorithm 2** Find-Anchor( $s, T, \Pi$ )

---

**Input:** Input string  $s$ , number of targeted substrings  $T$ , random hash function  $\Pi : \Sigma^q \rightarrow (0, 1)$

**Output:** The set of anchors  $A$  on  $s$

```

1:  $A \leftarrow \{1\}$ 
2:  $r \leftarrow \lfloor \frac{|s|-q+1-T}{2T+2} \rfloor$ 
3: Initialize an empty array  $h$  with  $|s| - q + 1$  elements
4: for each  $i \in [|s| - q + 1]$  do
5:    $h[i] \leftarrow \Pi(s_{i..i+q-1})$ 
6: end for
7: for each  $i \in [1 + r, |s| - q + 1 - r]$  do
8:    $Label \leftarrow 1$ 
9:   for each  $j \in [i - r, i + r]$  and  $j \neq i$  do
10:    if  $h[i] \geq h[j]$  then
11:       $Label \leftarrow 0$ 
12:    Exit the for loop
13:    end if
14:   end for
15:   if  $Label = 1$  then
16:      $A \leftarrow A \cup \{i\}$ 
17:   end if
18: end for
19:  $A \leftarrow A \cup \{|s|\}$ 

```

---

an anchor of  $s_1$ . Same to the 14-th 3-gram ‘‘CTA’’. We then partition  $s_1$  to  $\{\text{ACGTG}, \text{CTAACGTG}, \text{CTAACGTA}\}$ . We next find that the strings  $s_1, s_2$  share a common partition ‘‘CTAACGTG’’,  $s_3, s_4$  share a common partition ‘‘TCGAAT’’, and  $s_3, s_4, s_5$  share a common partition ‘‘CGTCGAAT’’, which give the following candidate pairs:  $(s_1, s_2), (s_3, s_4), (s_3, s_5), (s_4, s_5)$ . After computing the exact edit distance of each pair, we output  $(s_1, s_2), (s_3, s_4), (s_3, s_5)$  as the final answer (i.e., those whose edit distances are no more than  $K = 4$ ).

**Discussions.** We would like to discuss two items in more detail. First, we require the value of an anchor in the hash array  $h[]$  to be *strictly* smaller than its  $2r$  neighbors. The purpose of this is to reduce the number of false positives generated by periodic substrings with short periods; false positives will increase the running time of the verification step of the **MinJoin** algorithm. In real world datasets, periodic substrings are often caused by systematic errors, and may be shared among different strings. For example, consider the following periodic substring on genome data ‘‘...AAAAAAA ...’’ produced by sequencing errors, if we allow the value of an anchor to be equal to its neighbors, then we may have many anchors in this substring. Consequently, two strings both containing such a substring will be considered as a candidate pair even that they are very different elsewhere.

Second, we use different neighborhood size  $r$  for strings of different lengths. More precisely, we set  $r = \lfloor \frac{|s|-q+1-T}{2T+2} \rfloor$  where  $T = \Theta(K)$  is an input parameter standing for the number of targeted partitions. The purpose of doing this, instead of choosing a fixed  $r$  for all strings, is again to reduce false positives. Indeed, if we choose the same  $r$  for all strings, then long strings will generate many partitions, since in order to achieve perfect accuracy we cannot set  $r$  to be too large at the

3-gram	Value	3-gram	Value	3-gram	Value
CTA	0.01	ACG	0.39	GAA	0.69
GCT	0.05	AAA	0.42	AAT	0.74
TGC	0.12	AAC	0.46	ATC	0.77
TAA	0.21	CCT	0.53	GTC	0.83
ACC	0.25	TCG	0.58	TGG	0.89
CGT	0.31	ATC	0.62	GGA	0.91
GTG	0.33	CGA	0.64	GCG	0.97

Table 2: Hash values of 3-grams

ID	String	Length
$s_1$	ACGTGCTAACGTGCTAACGTG	21
$s_2$	AAACGTGCTAACGTGCTAACCT	22
$s_3$	TCGAATCGTCGAATCGTCGAA	21
$s_4$	TCGAATCGTCGAATCGTGGAA	21
$s_5$	GTGCGAATCGTCGAATCGTCG	21

Table 3: Input strings

presence of short strings. Consequently, the large number of partitions generated by long strings will contribute to many false positives.

This is in contrast to **VChunk**, who cuts the string whenever it finds a word in CBD appearing on the string. Consequently two strings of very different length but sharing a relatively long substring are likely to be considered as a candidate pair, producing a false positive for the verification.

## 2.2 The Analysis

We now analyze the properties of Algorithm 1. Our goal is to understand how many partitions Algorithm 1 will generate (which will contribute to the running time of **MinJoin** as we shall see in Section 3), and what is the probability for two similar strings to share a common partition.

To keep the analysis clean, we assume that in any  $r$ -neighborhood of the array  $h[]$  all the coordinates are distinct, which is true if (1) we assume that all corresponding  $q$ -grams are different, and (2) the hash function  $\Pi : \Sigma^q \rightarrow (0, 1)$  does not produce a collision when applying to  $q$ -grams. The later can be easily satisfied if we keep an  $O(\log N)$ -bit precision ( $N$  is the maximum string length) in the range of  $\Pi$ , in which case there is no hash collision with probability  $1 - 1/N^{\Omega(1)}$ . For the former, we set  $q = 3 \log_{|\Sigma|}(N/T)$ . Note that by our choice of  $r$  we have  $r \approx N/(2T)$ . If all letters in a substring of size  $r$  are random, then the probability that two  $q$ -grams in this substring are the same is  $1/|\Sigma|^q = (\frac{T}{N})^3$ . By a union bound with probability  $1 - o(1)$  all  $q$ -grams in a substring of size  $2r$  are different. We emphasize that this assumption is only used for the convenience of the analysis, and Algorithm 1 works without this constraint.

The following lemma states that the number of anchors produced by Algorithm 2 is concentrated around  $T$ , the number of targeted partitions.

**Lemma 1** *Given an input string and a parameter  $T$ , for any  $c > 0$ , the number of anchors generated by Algorithm 2, denoted by  $X$ , satisfies  $\Pr[|X - T| \geq \sqrt{cT}] < 1/c$ .*

ID	Partitions of string	$r$
$s_1$	ACGTG, CTAACGTG, CTAACGTA	2
$s_2$	AAACGTG, CTAACGTG, CTAACCT	2
$s_3$	TCGAAT, CGTCGAAT, CGTCGAA	2
$s_4$	TCGAAT, CGTCGAAT, CGTGGAA	2
$s_5$	GTGCGAAT, CGTCGAAT, CGTCG	2

Table 4: Partitions of strings by Algorithm 1 ( $T = 3$ )

**Proof:** Consider the array  $h[1..|s| - q + 1]$  constructed in Algorithm 2;  $h[i]$  is the hash value of the  $i$ -th  $q$ -gram of  $s$ . Let  $w = |s| - q + 1 - 2r$ . For  $i = 1, \dots, w$ , define a random variable  $X_i$  whose value is 1 if  $h[i+r]$  is the smallest coordinate in the window  $h[i..i+2r]$ , and 0 otherwise. Let  $X = \sum_{i \in [w]} X_i$ , which is the total number of anchors generated by Algorithm 2. We now analyze the random variable  $X$ .

We start by computing its expectation. Recall that we have set  $r$  to be  $\lfloor \frac{|s|-q+1-T}{2T+2} \rfloor$  at Line 2 of Algorithm 2. For simplicity we ignore the floor operation whose effect is negligible to the analysis.

$$\mathbf{E}[X] = \sum_{i \in [w]} \mathbf{E}[X_i] = \sum_{i \in [w]} \mathbf{Pr}[X_i = 1] = \frac{w}{2r+1} = T. \quad (1)$$

We next compute the variance.

$$\begin{aligned} \mathbf{Var}[X] &= \sum_{i \in [w]} \mathbf{Var}[X_i] + \sum_{i \neq j} \mathbf{Cov}[X_i, X_j] \\ &= \sum_{i \in [w]} \mathbf{Var}[X_i] + \frac{1}{2} \sum_i \sum_{j \neq i} \mathbf{Cov}[X_i, X_j]. \end{aligned} \quad (2)$$

We compute the two terms of (2) separately. For the first term,

$$\begin{aligned} \sum_{i \in [w]} \mathbf{Var}[X_i] &= \sum_{i \in [w]} \left( \mathbf{E}[X_i^2] - (\mathbf{E}[X_i])^2 \right) \\ &= w \times \left( \frac{1}{2r+1} - \frac{1}{(2r+1)^2} \right) \\ &\leq \frac{w}{2r+1}. \end{aligned} \quad (3)$$

For the second term of (2), by the definition of the covariance,

$$\begin{aligned} \mathbf{Cov}[X_i, X_j] &= \mathbf{E}[X_i X_j] - \mathbf{E}[X_i] \mathbf{E}[X_j] \\ &= \mathbf{E}[X_i X_j] - \frac{1}{(2r+1)^2}. \end{aligned}$$

We analyze  $\mathbf{E}[X_i X_j]$  in three cases.

**Case I.**  $|i - j| \geq 2r + 1$ . It is easy to see that in this case  $X_i$  and  $X_j$  are independent, since their corresponding windows  $h[i..i+2r]$  and  $h[j..j+2r]$  are disjoint. We thus have  $\mathbf{E}[X_i X_j] = \mathbf{E}[X_i] \mathbf{E}[X_j]$ , and consequently  $\mathbf{Cov}[X_i, X_j] = 0$ .

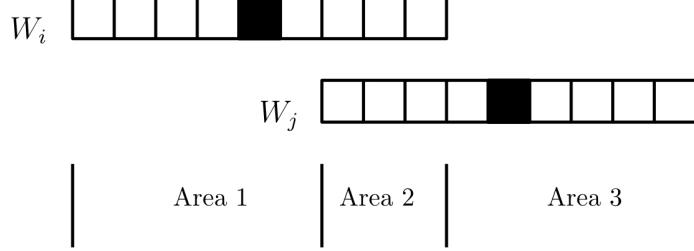


Figure 1: Illustration of windows  $W_i, W_j$  when  $r < |i - j| < 2r + 1$ . Black square represents the central coordinate of the window. The squares in same column correspond to same coordinate in the array  $h[]$ ; we duplicate them for the illustration purpose.

**Case II.**  $|i - j| \leq r$ . In this case,  $h[i+r]$  is inside the window  $h[j..j+2r]$ , and symmetrically  $h[j+r]$  is inside the window  $h[i..i+2r]$ . Thus if  $X_i = 1$  then we must have  $X_j = 0$ , and if  $X_j = 1$  then we must have  $X_i = 0$ . Therefore  $\mathbf{E}[X_i X_j] = 0$ , and consequently  $\mathbf{Cov}[X_i, X_j] = -\frac{1}{(2r+1)^2}$ .

**Case III.**  $r < |i - j| < 2r + 1$ . The analysis for this case is a bit more complicated. Consider two windows  $W_i = h[i..i+2r]$  and  $W_j = h[j..j+2r]$  which overlap. We divide their union into three areas; see Figure 1 for an illustration. Area 2 denotes the intersection of the two windows, and Area 1 and Area 3 denote the coordinates that are only in  $W_i$  and  $W_j$  respectively. It is easy to see that the number of coordinates in Area 1 and Area 3 are equal; let  $\alpha$  ( $r < \alpha < 2r + 1$ ) denote this number.

We write

$$\begin{aligned}\mathbf{E}[X_i X_j] &= \mathbf{Pr}[X_i = 1, X_j = 1] \\ &= \mathbf{Pr}[X_j = 1 \mid X_i = 1] \cdot \mathbf{Pr}[X_i = 1] \\ &= \mathbf{Pr}[X_j = 1 \mid X_i = 1] \cdot \frac{1}{2r+1}.\end{aligned}$$

We thus only need to analyze  $\mathbf{Pr}[X_j = 1 \mid X_i = 1]$ . Define a random variable  $Y$  such that  $Y = 1$  if the central coordinate of  $W_i$  (i.e.,  $h[i+r]$ ) is smaller than all coordinates in Area 3. We have

$$\begin{aligned}\mathbf{Pr}[X_j = 1 \mid X_i = 1] &= \mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 1] \cdot \mathbf{Pr}[Y = 1 \mid X_i = 1] + \\ &\quad \mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 0] \cdot \mathbf{Pr}[Y = 0 \mid X_i = 1].\end{aligned}\tag{4}$$

Note that  $(X_i = 1) \wedge (Y = 1)$  implies that the central coordinate of  $W_i$  is smaller than all coordinates in  $W_j$ , which, however, does not give any information about the relationship between all coordinates in  $W_j$ . We thus have

$$\mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 1] = \mathbf{Pr}[X_j = 1] = \frac{1}{2r+1}.\tag{5}$$

On the other hand,  $(X_i = 1) \wedge (Y = 0)$  implies that the central coordinate of  $W_i$  is smaller than all coordinates in Area 2, and is larger than some coordinate in Area 3. We thus know that the minimum coordinate of  $W_j$  must lie in Area 3. Therefore  $X_j = 1$  if and only if the central coordinate of  $W_j$  is larger than all other coordinates in Area 3. We get

$$\mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 0] = 1/\alpha.\tag{6}$$

Plugging in (5) and (6) to (4), we have

$$\begin{aligned} & \Pr[X_j = 1 \mid X_i = 1] \\ &= \frac{1}{2r+1} \cdot \Pr[Y = 1 \mid X_i = 1] + \frac{1}{\alpha} \cdot \Pr[Y = 0 \mid X_i = 1] \\ &\leq \frac{1}{\alpha} \leq \frac{1}{r+1}. \end{aligned}$$

Consequently we have

$$\mathbf{Cov}[X_i, X_j] \leq \frac{1}{2r+1} \cdot \frac{1}{r+1} - \frac{1}{(2r+1)^2} < \frac{1}{(2r+1)^2}.$$

Summing up, we have

$$\mathbf{Cov}[X_i, X_j] \begin{cases} = -\frac{1}{(2r+1)^2}, & |i-j| \leq r \\ < \frac{1}{(2r+1)^2}, & r < |i-j| < 2r+1 \\ = 0, & |i-j| \geq 2r+1 \end{cases} \quad (7)$$

Plugging (3) and (7) to (2), we get

$$\begin{aligned} \mathbf{Var}[X] &< \frac{w}{2r+1} + \frac{1}{2} \cdot w \cdot 2r \cdot \left( \frac{1}{(2r+1)^2} - \frac{1}{(2r+1)^2} \right) \\ &= \frac{w}{2r+1} = T. \end{aligned} \quad (8)$$

By (1), (8), and the Chebyshev's inequality, we have that for any constant  $c > 0$ ,

$$\Pr[|X - T| \geq \sqrt{cT}] < 1/c.$$

□

We have empirically verified the concentration result in Lemma 1 on two real world datasets (to be introduced in Section 4); see Figure 2. It is clear that the number of partitions Algorithm 1 generates are tightly concentrated around the number of target partitions  $T$ .

We next analyze another key property of our local minimum based partition: Given two similar strings, what is the probability that they share a common partition? We give the following lemma.

**Lemma 2** *For two strings  $s, t$  with  $\text{ED}(s, t) \leq K$ , let  $\mathcal{P}_s$  and  $\mathcal{P}_t$  be the partitions outputted by Algorithm 1 (setting  $T = 120K$ ) on  $s$  and  $t$  respectively. Assume  $|s| = \omega(Kq)$ . The probability that  $\mathcal{P}_s$  and  $\mathcal{P}_t$  share a common partition is at least 0.98.*

**Proof:**

Since  $\text{ED}(s, t) \leq K$ , we have  $|t| \in [|s| - K, |s| + K]$ , and  $s$  and  $t$  must share a common substring of length at least  $L = (|s| - K)/(K + 1)$  in the optimal alignment.

Let  $\gamma$  be such a common substring. Let  $r_s = \lfloor \frac{|s|-q+1-T}{2T+2} \rfloor$ , and let  $\eta = \frac{L-q+1-2r_s}{2r_s+1}$ . When running Algorithm 2 on  $s$ , by an almost identical argument as that for the proof of Lemma 1, we have that the number of anchors  $X$  on  $\gamma$  satisfies

$$\Pr[|X - \eta| \geq \sqrt{c\eta}] < 1/c. \quad (9)$$

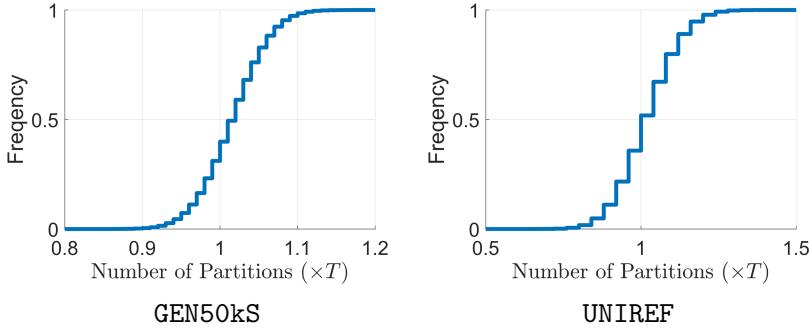


Figure 2: The CDFs of numbers of partitions on each string returned by Algorithm 1 on GEN50kS and UNIREF datasets, with parameters  $T = 100$  and  $T = 25$  respectively.

For  $T = 120K$  and  $|s| = \omega(Kq)$ , we have

$$\begin{aligned}
\eta &= \frac{L - q + 1 - 2r_s}{2r_s + 1} \\
&\geq \left( \frac{|s| - K}{K + 1} - q + 1 - 2r_s \right) \cdot \frac{T + 1}{|s| - q + 2} \\
&\geq 115.
\end{aligned} \tag{10}$$

Plugging (10) to (9), we have with probability at least  $(1 - 1/100) = 0.99$  that

$$X \geq \eta - \sqrt{100\eta} > 4, \tag{11}$$

which means that with probability 0.99 there are at least four anchors on  $\gamma$ .

Let  $a_1, a_2, a_3, a_4$  be four anchors on  $\gamma$  when processing  $s$  using Algorithm 2. Let  $r_t = \lfloor \frac{|t|-q+1-T}{2T+2} \rfloor$ . Since  $\text{ED}(s, t) \leq K$  and  $T = 120K$ , it holds that  $|r_t - r_s| \leq 1$ . In the case that  $r_t = r_s = r$ ,  $a_2$  and  $a_3$  must also be anchors when processing  $t$  using Algorithm 2, since an anchor is fully determined by a neighborhood of size  $r$ .

For the case when  $|r_t - r_s| = 1$ , w.l.o.g., assume that  $r_s = r$  and  $r_t = r + 1$ . Now the probability that  $a_2$  is still an anchor when processing  $t$ , given the fact that  $a_2$  is an anchor when processing  $s$ , is at least  $1 - 1/(r + 1)$ . Same argument holds for  $a_3$ . Thus with probability  $0.99 - 2/(r + 1) \geq 0.98$  (note that  $r = r_s = \lfloor \frac{|s|-q+1-T}{2T+2} \rfloor = \omega(1)$  given  $|s| = \omega(qK)$  and  $T = 120K$ ),  $a_2$  and  $a_3$  are also anchors when processing  $t$ .

Finally, observe that once  $s$  and  $t$  share two adjacent anchors  $a_2$  and  $a_3$ , they must share at least one common partition.  $\square$

**Remark 1 (Choice of  $T$ )** We note that the choice of  $T$  ( $= 120K$ ) in Lemma 2 is overly ‘‘pessimistic’’ – it is just for the convenience of analysis. Moreover, we only considered one pair of common substring of length  $L \approx |s|/K$ , while the average length of the (at most)  $K + 1$  pairs of common substrings between  $s$  and  $t$  in the optimal alignment is at least  $\frac{s-K}{K+1} \approx |s|/K$ . A finer analysis which considers all pairs of common substrings in the optimal alignment can reduce the value of  $T$  all the way down to a value close to  $K$ , while still guarantee that  $\mathcal{P}_s$  and  $\mathcal{P}_t$  share a common partition with a good probability. However, the analysis is a bit cumbersome and we will leave it to the full version of this paper. The main point of this remark is that in practice we can

just set  $T \approx K$ , or even smaller since in real-world datasets multiple edits may occur in the same location, which effectively increases the average length of common substrings. In our experiments we find that  $T \in [K/5, K]$  are good choices for all the datasets we have tested.

*Parallel repetitions for boosting the success probability.* Though the success probability in Lemma 2 is only 0.98, and it is only for each pair of similar strings, we can easily boost it to high probability for all pairs of similar strings using parallel repetitions. We can repeat the partition process for each string for  $\log n$  times using independent randomness, and then union all the partitions of the string. Now for each pair of similar strings, the probability that they share a common partition is at least  $1 - 0.02^{\log n} \geq 1 - 1/n^5$ . We then use a union bound on the at most  $n^2$  pairs of similar strings, and get that the probability that all pairs of similar strings share at least one common partition is at least  $1 - 1/n^3$ . We note in our experiments that we do not need this boosting procedure since a single run of the partition process already achieves perfect accuracy.

**Theorem 1** *If we apply Algorithm 1 augmented by the parallel repetition discussed above on all input strings, then with probability  $1 - n^{-\Omega(1)}$ , all pair of strings with edit distances at most  $K$  will share at least one common partition. The expected running time of the algorithm is  $\log n$  times the input size, and the space needed is also  $\log n$  times the input size.*

**Proof:** The correctness follows directly from Lemma 2 and the discussion of parallel repetition above. In the rest of the proof we focus on the time and space. In fact, to show the claimed time and space usage we can just show that the time and space for partitioning one string  $s$  (by Algorithm 1) is linear in terms of the string length  $|s|$ .

The running time of Algorithm 1 is dominated by that of its subroutine Algorithm 2. The hash values of all  $q$ -grams of  $s$  can be computed by the Rabin-Karp algorithm (the rolling hash) in  $O(|s|)$  time. For Line 7-18 of Algorithm 2, since each number in  $h[]$  is a random hash value, the inner for-loop (Line 9-14) runs in  $O(1)$  time in expectation. Therefore the total running time of Algorithm 1 is  $O(|s|)$  in expectation.

Clearly, the space usage of Algorithm 1 is also  $O(|s|)$ . □

### 3 The MinJoin Algorithm

We now present our main algorithm **MinJoin**, depicted in Algorithm 3. We briefly explain it in words below.

The **MinJoin** algorithm has three stages: initialization (Line 1 - 4), join and filtering (Line 5 - 20) and verification (Line 21 - 25). In the first stage, we initialize an empty set  $\mathcal{C}$  for candidate pairs and an empty hash table  $\mathcal{D}$ , generate a random hash function  $\Pi$ , and sort all strings according to their lengths for the pruning.

In the join and filtering stage, we compute the partitions for each input string using Algorithm 1. For each partition  $(pos, len)$ , which refers the substring of  $s_i$  with length  $len$  and  $pos$  is the index of its first character on  $s_i$ , we find all tuples  $(j, pos_j, len_j)$  in  $f((s_i)_{pos..pos+len-1})$ -th bucket of hash table  $\mathcal{D}$  (that is, we perform a hash join). We use two rules to prune the candidate pairs we have found. The first condition (Line 9) says that if the lengths of  $s_i$  and  $s_j$  differ by larger than  $K$ , then it is impossible to have  $ED(s_i, s_j) \leq K$ . Consequently it is impossible to have  $ED(s_j, s_{i'}) \leq K$  for any  $i' > i$ .

---

**Algorithm 3** MinJoin ( $\mathcal{S}, K, T$ )

---

**Input:** Set of input strings  $\mathcal{S} = \{s_1, \dots, s_n\}$ , distance threshold  $K$ , number of targeted partitions  $T$

**Output:**  $\mathcal{O} \leftarrow \{(s_i, s_j) \mid s_i, s_j \in \mathcal{S}; i \neq j; \text{ED}(s_i, s_j) \leq K\}$

- 1:  $\mathcal{O} \leftarrow \emptyset, \mathcal{C} \leftarrow \emptyset$  ▷  $\mathcal{C}$  : collection of candidate pairs
- 2: Pick a hash function  $f : \Sigma^* \rightarrow \mathbb{N}$  and initialize an empty hash table  $\mathcal{D}$
- 3: Generate a random hash function  $\Pi : \Sigma^q \rightarrow (0, 1)$
- 4: Sort strings in  $\mathcal{S}$  first by string length increasingly, and second by the alphabetical order
- 5: **for each**  $s_i \in \mathcal{S}$  (in the sorted order) **do**
- 6:      $\mathcal{P} \leftarrow \text{Partition-String}(s_i, T, \Pi)$
- 7:     **for each**  $(pos, len) \in \mathcal{P}$  **do**
- 8:         **for each**  $(j, pos_j, len_j)$  in the  $f((s_i)_{pos..pos+len-1})$ -th bucket of  $\mathcal{D}$  **do** ▷  $f(\cdot)$  is the hash function picked at Line 2
- 9:             **if**  $||s_i| - |s_j|| \leq K$  **then**
- 10:                 **if**  $|pos - pos_j| + |(|s_i| - pos) - (|s_j| - pos_j)| \leq K$  **then**
- 11:                      $\mathcal{C} \leftarrow \mathcal{C} \cup (s_i, s_j)$
- 12:                 **end if**
- 13:             **else**
- 14:                 Remove  $(j, pos_j, len_j)$  from  $\mathcal{D}$
- 15:             **end if**
- 16:         **end for**
- 17:         Store  $(i, pos, len)$  in the  $f((s_i)_{pos..pos+len-1})$ -th bucket of  $\mathcal{D}$
- 18:     **end for**
- 19: **end for**
- 20: Remove duplicate pairs in  $\mathcal{C}$
- 21: **for each**  $(x, y) \in \mathcal{C}$  **do**
- 22:     **if**  $\text{ED}(x, y) \leq K$  **then**
- 23:          $\mathcal{O} \leftarrow \mathcal{O} \cup (x, y)$
- 24:     **end if**
- 25: **end for**

---

The second condition (Line 10) concerns the following scenario: if  $s_i$  and  $s_j$  match at indices  $pos$  and  $pos_j$ , which divides both strings into two substrings  $\nu_1 = (s_i)_{1..pos-1}, \nu_2 = (s_i)_{pos..|s_i|}$ , and  $\mu_1 = (s_j)_{1..pos_j-1}, \mu_2 = (s_j)_{pos_j..|s_j|}$ . If  $pos$  and  $pos_j$  are indeed matched in the optimal alignment, then we must have  $\text{ED}(\nu_1, \mu_1) + \text{ED}(\nu_2, \mu_2) \leq K$ , in which case we have  $|(|s_i| - pos) - (|s_j| - pos_j)| + |pos - pos_j| \leq K$ .

We add all pairs of strings that pass the two filtering conditions to the candidate set  $\mathcal{C}$ , and then perform a deduplication step at the end since each pair can potentially be added into  $\mathcal{C}$  multiple times.

In the verification stage, we verify whether each pair of strings in  $\mathcal{C}$  indeed have edit distance at most  $K$ , using the standard dynamic programming algorithm by Ukkonen [11]. Due to this verification step our algorithm will never output any false positive. On the other hand, by Theorem 1, if we augment the string partition scheme with parallel repetition, then **MinJoin** will not produce any false negative with probability  $1 - 1/n^{\Omega(1)}$ . Therefore **MinJoin** will achieve perfect accuracy

with probability  $1 - 1/n^{\Omega(1)}$ .

**Time and Space Analysis.** Let  $N$  be the maximum string length in the set of input strings  $\mathcal{S}$ , and  $n = |\mathcal{S}|$ . By Theorem 1 the running time of the partition (without the parallel repetition) is bounded by  $O(nN)$ .

The total number of pairs that are fed into the filtering steps (Line 9, 10) inherently depends on the concrete dataset. Suppose partitions of all strings are evenly distributed into  $|\mathcal{D}|$  buckets of the hash table  $\mathcal{D}$  (this is indeed what we have observed in our experiments), then we can upper bound this number by  $O\left(\frac{nK}{|\mathcal{D}|}\right)^2$  with probability 0.99. To see this, by the proof in Lemma 1 we know that the expected number of partitions of each string is  $T = \Theta(K)$ . By linearity of expectation, the expected number of partitions of all  $n$  strings is  $nT$ . Therefore the total number of actual partitions is bounded by  $O(nK)$  with probability 0.99 by a Markov inequality. The verification step can be done in  $O(|\mathcal{C}| NK)$  where  $\mathcal{C}$  is the set of the candidate pairs.

The space usage is clearly bounded by  $O(nN)$ , that is, the size of the input.

**Theorem 2** *The MinJoin algorithm has the following theoretical properties. Consider the case that we augment the string partition procedure at Line 6 with  $\log n$  parallel repetitions.*

- *It achieves 100% accuracy with probability  $1 - 1/n^{\Omega(1)}$ .*
- *Assuming that the partitions of all strings are evenly distributed into the buckets of the hash table, the running time of MinJoin is bounded by*

$$O\left(nN \log n + \left(\frac{nK}{|\mathcal{D}|}\right)^2 + |\mathcal{C}| NK\right)$$

*with probability 0.99, where  $\mathcal{C}$  is the set of the candidate pairs MinJoin produces before the verification step.*

- *The space usage of MinJoin is  $\log n$  times the size of input.*

## 4 Experiments

In this section we present our experimental studies. We start by describing the datasets and algorithms used in our experiments. We then provide a detailed study of the performance of MinJoin. Finally, we compare MinJoin with the state-of-the-art algorithms for edit similarity joins.

### 4.1 Setup of Experiments

We implemented our algorithms in C++ and performed experiments on a Dell PowerEdge T630 server with 2 Intel Xeon E5-2667 v4 3.2GHz CPU with 8 cores each, and 256GB memory.

**Datasets.** We use the datasets in [17] which are publicly available.<sup>2</sup> Table 5 describes the statistics of tested datasets.

---

<sup>2</sup>See the documentation from the project website of [17]: <https://github.com/kedayuge/Embedjoin>

Datasets	$n$	Avg Len	Min Len	Max Len	$ \Sigma $
UNIREF	400000	445	200	35213	25
TREC	233435	1217	80	3947	37
GEN50kS	50000	5000	4829	5152	4
GEN20kS	20000	5000	4829	5109	4
GEN20kM	20000	10000	9843	10154	4
GEN20kL	20000	20000	19821	20109	4
GEN80kS	80000	5000	4814	5109	4
GEN320kS	320000	5000	4811	5154	4

Table 5: Statistics of tested datasets (from [17])

**UNIREF:** A dataset consists of UniRef90 protein sequence data obtained from UniProt Project.<sup>3</sup> The sequences whose lengths are smaller than 200 are removed, and the first 400,000 protein sequences are extracted.

**TREC:** A dataset consists of titles and abstracts from 270 medical journals. The title, author, and abstract fields are extracted and concatenated. Punctuation marks are converted into white space and all letters are in uppercase.

**GEN-X-Y's:** Datasets contain 50 human genomes obtained from the Personal Genomes Project,<sup>4</sup> where X denotes the number of strings (range from 20k to 320k), and Y denotes the string length ( $S \approx 5k$ ,  $M \approx 10k$ ,  $L \approx 20k$ ). Each string is a substring randomly sampled from the Chromosome 20 of human genome.

**Algorithms.** We compare `MinJoin` with the state-of-the-art algorithms for edit similarity joins discussed in the introduction, including `PassJoin`[8], `QChunk`[9], `VChunk`[15], `EmbedJoin`[17]. All codes are downloaded from the corresponding project websites.

**Measurements and Choices of Parameters.** We use three metrics to measure the performance of tested algorithms: time, space, and accuracy.

We note that except `MinJoin` and `EmbedJoin` which are randomized and may have false negatives, all other tested algorithms are deterministic and output the exact number of similar pairs, and thus their accuracy is always 100%. According to our theoretical analysis (Theorem 1 and Remark 1), by setting  $T$  appropriately and using  $\log n$  repetitions of the string partition procedure (Algorithm 1), `MinJoin` can output all similar pairs with a high probability. In practice, we found that a single execution of Algorithm 1 with  $T \in [K/5, K]$  can already achieve 100% accuracy on all tested datasets.<sup>5</sup> In fact, as we shall see in Figure 3 and Figure 4, varying  $T$  in this range will not change the accuracy by much, but it does slightly affect the running time since larger  $T$  will introduce more false positives for verification.

In the rest of this section we will always write the accuracy for `EmbedJoin` on the plots, and omit that for `MinJoin` if it is 100%.

<sup>3</sup><http://www.uniprot.org/>

<sup>4</sup><https://www.personalgenomes.org/us>

<sup>5</sup>Whenever there is an exact algorithm that finishes in a reasonable amount of time so that we get to know the ground truth.

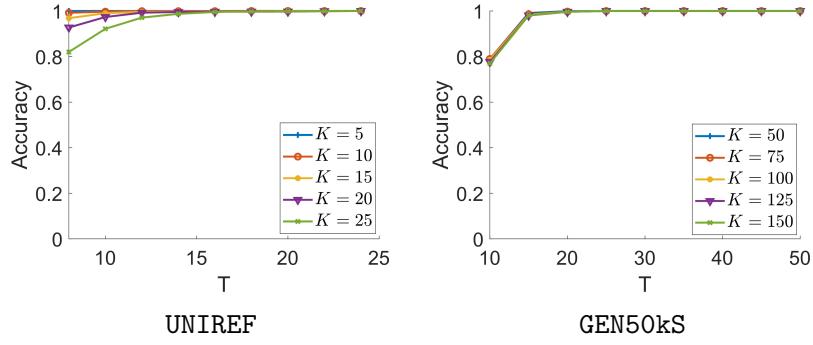


Figure 3: Influence of  $T$  on accuracy

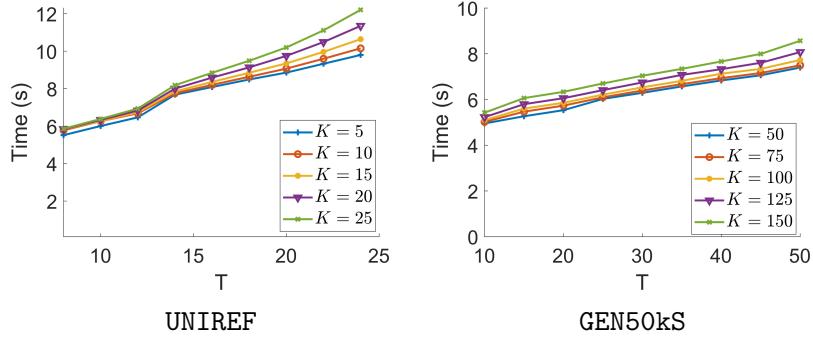


Figure 4: Influence of  $T$  on running time

We always choose the *best* parameters of other tested algorithms. **QChunk** has two parameters:  $q$  (the size of  $q$ -gram) and indexing method. We found that the *indexchunk* always performs better than *indexgram* on all datasets, and we always choose the best  $q$  for each experiment. **VChunk** has a parameter *scale* to tune. **PassJoin** has no parameter. **EmbedJoin** has three parameters  $m, r, z$ . We choose the parameters based on the recommendation of [17]: We select the best combinations of parameters to achieve at least 95% accuracy on UNIREF and TREC datasets, and at least 99% accuracy on GEN50kS dataset; and we select  $r = z = 7, m = 15 - \lfloor \log_2 x \rfloor$  on the rest of datasets, where  $x\%$  is the edit threshold.

Each result is an average of 5 independent runs. For **MinJoin** we fix the randomness at the beginning so that all runs return the same result on the same dataset.

## 4.2 Experiments for MinJoin

We first show the performance of **MinJoin**. We will start by investigating the influence of parameter  $T$  on running time and accuracy, and then present the running time of different stages of **MinJoin**.

**Influence of Parameter  $T$ .** We study empirically how parameter  $T$  influences the accuracy and the running time of **MinJoin**. We present the influence of  $T$  on the accuracy and running time in Figure 3 and 4 respectively. As predicted by theory, both time and accuracy increase when  $T$  increase. We also tested different edit thresholds  $K$ . We observe that when  $K$  is larger, we need a larger  $T$  to maintain the 100% accuracy, which is also consistent with the theory where we need

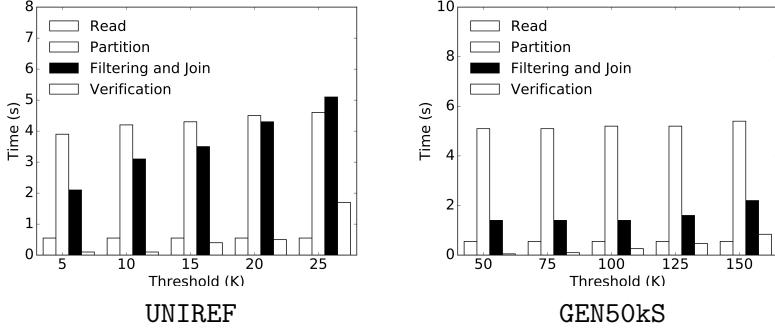


Figure 5: Running time of different parts of **MinJoin**, varying  $K$ .

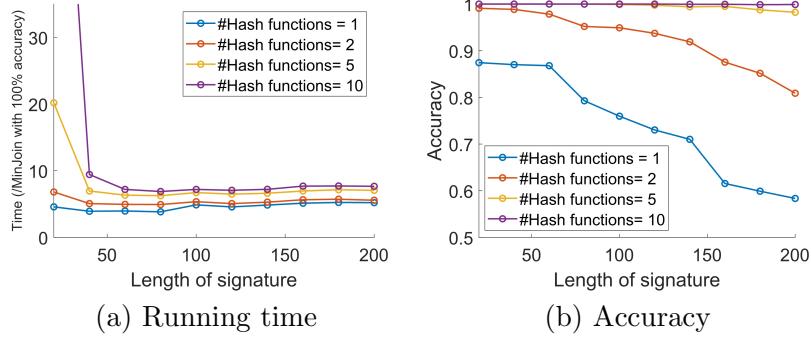


Figure 6: Performance of the MinHash based algorithm on GEN50kS dataset with  $K = 100$ . (a) The running time of the MinHash based algorithm as a multiple of that of **MinJoin** at 100% accuracy. (b) The accuracy of the MinHash based algorithm.

to pick  $T = \Theta(K)$ . As mentioned in Section 4.1, we found that setting  $T$  in the range  $[K/5, K]$  is good for all the tested datasets.

**Running Time of Different Parts of MinJoin.** We have also measured the running time of different parts of **MinJoin**, including input read, string partition, hash join and filtering, and verification. We present in Figure 5 the running time of **MinJoin** on (1) reading the input strings, (2) partitioning strings, (3) performing the hash join and filtering, and (4) verification varying the edit threshold  $K$ . Certainly, the input read time will not change for different  $K$ . We observe that the time for join and filtering increases slightly when  $K$  increases, that for partition is stable, and that for verification increases considerably when  $K$  increases. On UNIREF dataset, the string partitioning as well as join and filtering steps are bottleneck, and on GEN50kS dataset, the string partition step is bottleneck. The verification step takes the smallest amount of time in most cases.

### 4.3 A Comparison with MinHash

Before going to the main body of the experimental study, we try to argue that the folklore MinHash based algorithm is not competitive with **MinJoin**. The reason that we discuss it separately is that this folklore algorithm has two parameters for which we do not have any guideline for the tuning. We thus try to present its performance by testing different combinations of these parameters.

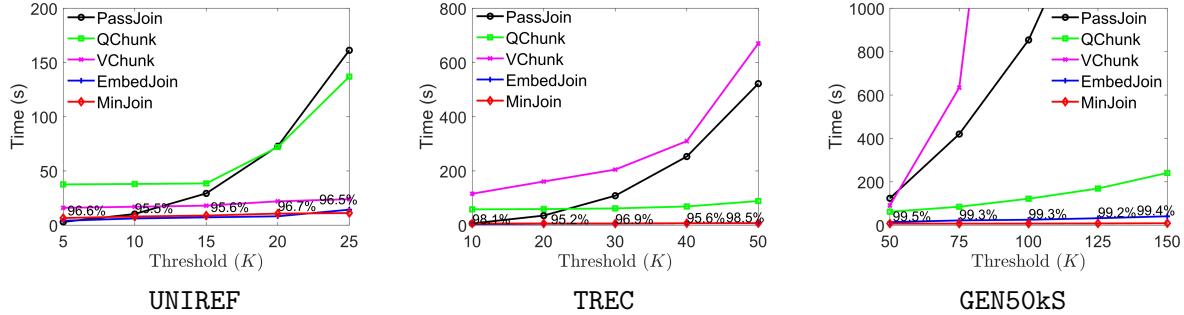


Figure 7: A comparison on running time, varying  $K$ . The percentages on plots stand for accuracy of `EmbedJoin`.

As mentioned in the introduction, the MinHash based algorithm is straightforward: we convert each string into a set which consists of the hash values of all  $q$ -grams of the string, and then pick the smallest value as the signature of the string for the subsequent hash join. To boost the accuracy, we can use  $\ell$  such MinHash functions, and get  $\ell$  signatures for each string. Applying  $\ell$  hash functions to get the signatures is expensive. A standard optimization method is to use only one hash function, and then select the top- $\ell$  smallest hash values as the signatures. This is what we use in our experiments.

Figure 6 shows the running time and accuracy of the MinHash based algorithm when varying the number of hash signatures  $\ell$  and the length of signature  $q$ . The running time is shown as a multiple of `MinJoin` at 100% accuracy. We find that the running time and accuracy of the MinHash based algorithm depend on the two parameters  $q$  and  $\ell$ : When increasing parameter  $\ell$ , both running time and accuracy increase; when increasing parameter  $q$ , the running time first decreases and then increases a little bit, and the accuracy decreases. We observe the accuracy and running time are sensitive to parameters, and there is no principle on how to select them for edit similarity joins. This is in contrast to `MinJoin` where the only parameter is  $T$  (the targeted number of partitions), and we have already discussed how to choose  $T$  both theoretically and practically. Moreover, even we choose the best combination of  $\ell$  and  $q$ , the running time of the MinHash based algorithm is still at least 5 times of that of `MinJoin` at 100% accuracy. We thus conclude that `MinJoin` outperforms the MinHash based algorithm in all aspects.

#### 4.4 A Comparison with the State-of-the-Art

We now compare `MinJoin` with the state-of-the-art algorithms for edit similarity joins (`QChunk`, `PassJoin`, `VChunk` and `EmbedJoin`). We will make use of UNIREF, TREC and GEN50kS for a basic comparison. These datasets are of modest size so that all algorithms can finish within 24 hours. We then use larger genome datasets to test the scalability of all algorithms.

**Effects of the Edit Threshold  $K$ .** Figure 7 presents the running time of different algorithms on UNIREF, TREC and GEN50kS when varying the edit threshold  $K$ . Compared with `EmbedJoin`, `MinJoin` clearly has the advantage on the accuracy (100% versus 95-99%). The running time of `MinJoin` is similar to `EmbedJoin` on UNIREF and TREC, and is better than `EmbedJoin` by a factor of 4.5 on GEN50kS ( $K = 150$ ). We observe that `MinJoin` has a significant advantage over all the exact algorithms on running time: `MinJoin` outperforms the best exact algorithm by a factor of 2.3 in

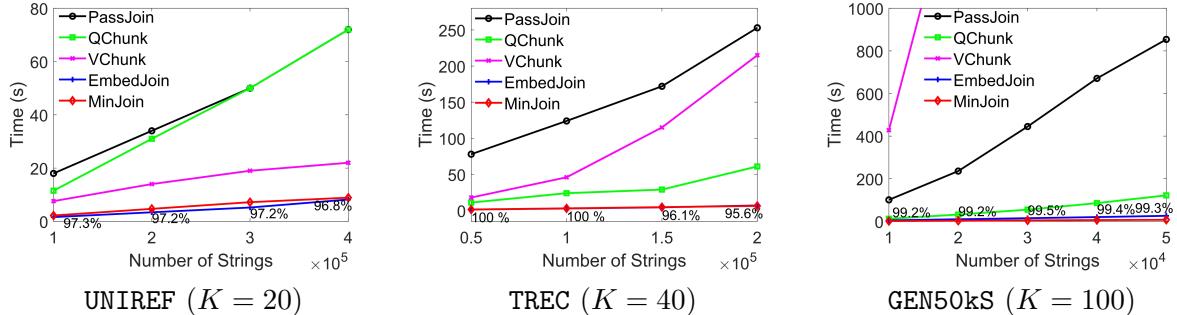


Figure 8: A comparison on running time, varying  $n$ . The percentages on plots stand for accuracy of `EmbedJoin`.

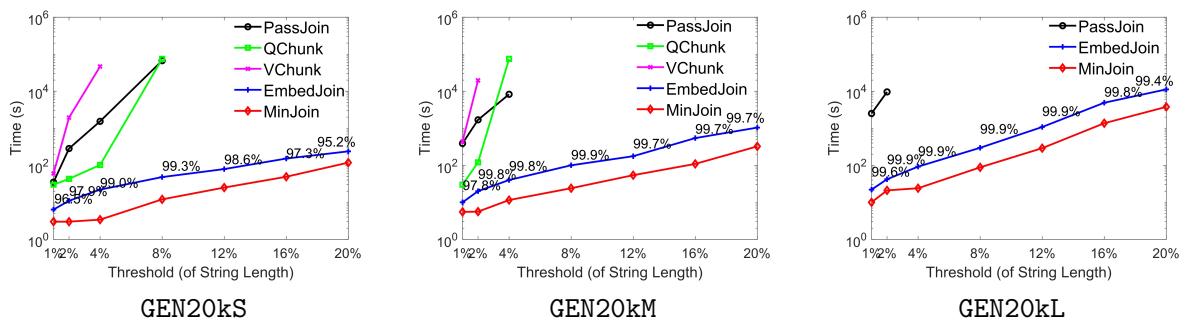


Figure 9: Scalability of different algorithms, varying  $N$ . The percentages on plots are accuracies of `EmbedJoin`.

UNIREF ( $K = 25$ ), 12.3 on TREC ( $K = 50$ ), and 26.7 on GEN50kS ( $K = 150$ ). The running time of `PassJoin` increases quickly when  $K$  becomes large; this is consistent to the theory that the query time in `PassJoin` for each string is proportional to  $K^3$ . `VChunk` performs relatively well on UNIREF, but much worse on TREC and GEN50kS. This may be because the preprocessing time of `VChunk` has a quadratic dependence on string length  $N$ , which is larger in TREC and GEN50kS than UNIREF.

**Effects of the Input Size  $n$ .** Figure 8 presents the running time of different algorithms on UNIREF, TREC and GEN50kS when varying the number of input strings  $n$ . `MinJoin` again has similar running time as `EmbedJoin` on UNIREF and TREC, and much better on GEN50kS (plus the accuracy advantage). The running time of `MinJoin` is better than the best exact algorithm by a factor of 2.2 on UNIREF ( $n = 400,000$ ), 9.5 on TREC ( $n = 200,000$ ), and 16.2 on GEN50kS ( $n = 50,000$ ). The trends of running time of all algorithms increase near linearly in terms of  $n$ , except `VChunk` whose performance deteriorates significantly when  $n$  increases on TREC and GEN50kS, which may again due to the expensive preprocessing step.

**Scalability of the Algorithms.** Finally we test all algorithms on larger datasets. Figure 9 presents the results of the running time when we scale string length up to 20,000 and the edit threshold  $K$  up to 20% of the string length. Figure 10 presents the results when we scale the number of strings up to 320,000, and  $K$  up to 20% of the string length. The first plot of Figure 10 is just a repeat of that of Figure 9. For `MinJoin` we always set the number of targeted partition  $T$  to be  $K/5$ , which already makes the accuracy of `MinJoin` to be 100% on those points where there

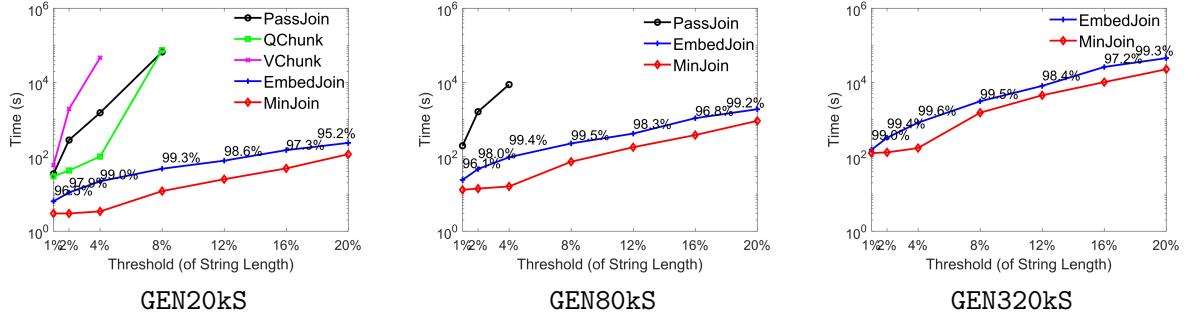


Figure 10: Scalability of different algorithms, varying  $n$ . The percentages on plots are accuracies of `EmbedJoin`.

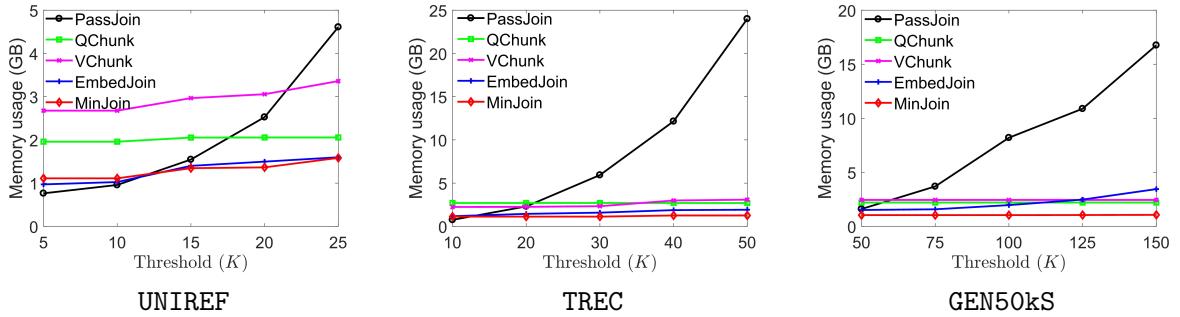


Figure 11: A comparison on memory usage, varying  $K$ .

is at least one exact algorithm that can finish.

We note that some algorithms cannot produce some of the points, which may be because they cannot finish within 24 hours, or there are some implementation issues (e.g., memory overflow). In cases when there is no exact algorithm that can finish in time, the accuracy of `EmbedJoin` is computed using the result returned by `MinJoin` as the ground truth.

We observe that `MinJoin` generally outperforms `EmbedJoin` by 2 ~ 5 times on the running time. The advantage slightly decreases when the number of strings  $n$  or the string length  $N$  increases. This is because when  $n$  or  $N$  increases, the verification time ( $O(NK)$  per pair where  $K$  is also proportional to  $N$  in our plots) will increase faster than other parts of the algorithm. On the other hand, the accuracy of `EmbedJoin`, using `MinJoin` as the baseline, is about 96%-99%.

All the exact algorithms do not scale well on these large datasets. On the smallest dataset `GEN20kS`, `PassJoin` and `QChunk` can run up to the 8% edit threshold, while `VChunk` can only go up to the 4% threshold. Their running times deteriorate significantly when  $K$  increases. Only `PassJoin` can produce some points on `GEN20kL` and `GEN80kS`. On `GEN320kS` none of the exact algorithms can finish within 24 hours.

**Memory Usage.** We have also compared the memory usage of all tested algorithms. Figure 11 and Figure 12 present the memory usage of different algorithms on `UNIREF`, `TREC` and `GEN50kS` when varying edit threshold  $K$  and the number of input strings  $n$ . While the difference on the memory usage is not as large as running time, `MinJoin` still performs the best among all algorithms. The performance of `PassJoin` is clearly worse than others on `TREC` and `GEN50kS`.

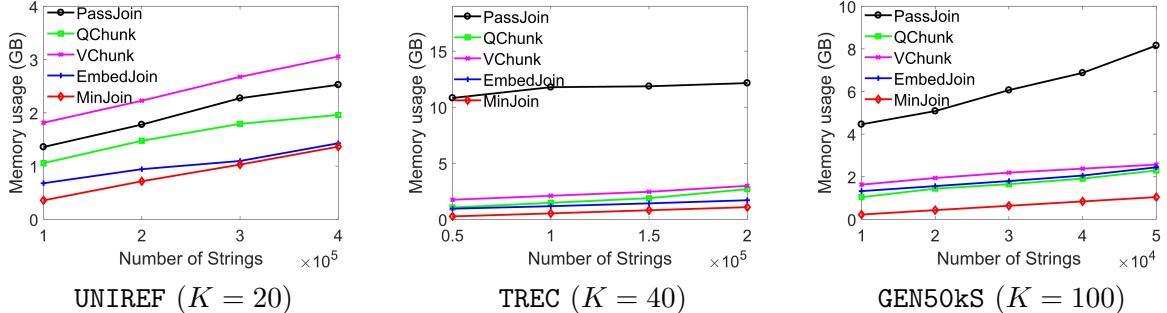


Figure 12: A comparison on memory usage, varying  $n$ .

## 5 Conclusion

In this paper we have presented **MinJoin**, an algorithm for edit similarity joins based on string partition using local hash minima. **MinJoin** has rigorous mathematical properties, and significantly outperforms previous methods on long strings with large edit thresholds. We feel that local hash minima based string partition is a natural and elegant way for solving the edit similarity join problem: it can be applied to each string independently by a linear scan, without any synchronization between strings or global statistics of the datasets. It also works very well with a simple hash join data structure for computing the candidate string pairs. Moreover, even **MinJoin** is a randomized algorithm, it can easily achieve perfect accuracy on all of the datasets that we have tested. We believe **MinJoin** is the right choice for edit similarity joins in many applications.

## References

- [1] ARASU, A., GANTI, V., AND KAUSHIK, R. Efficient exact set-similarity joins. In *VLDB* (2006), pp. 918–929.
- [2] BAYARDO, R. J., MA, Y., AND SRIKANT, R. Scaling up all pairs similarity search. In *WWW* (2007), pp. 131–140.
- [3] BOCEK, T., HUNT, E., STILLER, B., AND HECHT, F. *Fast similarity search in large dictionaries*. University, 2007.
- [4] CIACCIA, P., PATELLA, M., AND ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB* (1997), pp. 426–435.
- [5] GRAVANO, L., IPEIROTIS, P. G., JAGADISH, H. V., KOUDAS, N., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. Approximate string joins in a database (almost) for free. In *VLDB* (2001), pp. 491–500.
- [6] JIANG, Y., LI, G., FENG, J., AND LI, W. String similarity joins: An experimental evaluation. *PVLDB* 7, 8 (2014), 625–636.
- [7] LI, C., LU, J., AND LU, Y. Efficient merging and filtering algorithms for approximate string searches. In *ICDE* (2008), pp. 257–266.

- [8] LI, G., DENG, D., WANG, J., AND FENG, J. PASS-JOIN: A partition-based method for similarity joins. *PVLDB* 5, 3 (2011), 253–264.
- [9] QIN, J., WANG, W., LU, Y., XIAO, C., AND LIN, X. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD* (2011), pp. 1033–1044.
- [10] ROBERTS, R. J., CARNEIRO, M. O., AND SCHATZ, M. C. The advantages of smrt sequencing. *Genome biology* 14, 6 (2013), 405.
- [11] UKKONEN, E. Algorithms for approximate string matching. *Information and Control* 64, 1-3 (1985), 100–118.
- [12] WANDELT, S., DENG, D., GERDJIKOV, S., MISHRA, S., MITANKIN, P., PATIL, M., SIRAGUSA, E., TISKIN, A., WANG, W., WANG, J., AND LESER, U. State-of-the-art in string similarity search and join. *SIGMOD Record* 43, 1 (2014), 64–76.
- [13] WANG, J., LI, G., AND FENG, J. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB* 3, 1 (2010), 1219–1230.
- [14] WANG, J., LI, G., AND FENG, J. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD* (2012), pp. 85–96.
- [15] WANG, W., QIN, J., XIAO, C., LIN, X., AND SHEN, H. T. Vchunkjoin: An efficient algorithm for edit similarity joins. *IEEE Trans. Knowl. Data Eng.* 25, 8 (2013), 1916–1929.
- [16] XIAO, C., WANG, W., AND LIN, X. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.
- [17] ZHANG, H., AND ZHANG, Q. Embedjoin: Efficient edit similarity joins via embeddings. *KDD* (2017), 585–594.