EmbedJoin: Efficient Edit Similarity Joins via Embeddings*

Haoyu Zhang
Indiana University Bloomington
Bloomington, IN 47408, USA
hz30@umail.iu.edu

Qin Zhang
Indiana University Bloomington
Bloomington, IN 47408, USA
qzhangcs@indiana.edu

ABSTRACT

We study the problem of edit similarity joins, where given a set of strings and a threshold value K, we want to output all pairs of strings whose edit distances are at most K. Edit similarity join is a fundamental problem in data cleaning/integration, bioinformatics, collaborative filtering and natural language processing, and has been identified as a primitive operator for database systems. This problem has been studied extensively in the literature. However, we have observed that all the existing algorithms fall short on long strings and large distance thresholds.

In this paper we propose an algorithm named `EmbedJoin` which scales very well with string length and distance threshold. Our algorithm is built on the recent advance of metric embeddings for edit distance, and is very different from all of the previous approaches. We demonstrate via an extensive set of experiments that `EmbedJoin` significantly outperforms the previous best algorithms on long strings and large distance thresholds.

1 INTRODUCTION

Given a collection of strings, the task of similarity join is to find all pairs of strings whose similarities are above a predetermined threshold, where the similarity of two strings is measured by a specific distance function. Similarity join is a fundamental problem in data cleaning and integration (e.g., data deduplication), bioinformatics (e.g., find similar protein/DNA sequences), collaborative filtering (e.g., find user pairs of similar interests), natural language processing (e.g., automatic spelling corrections), etc. It has been studied extensively in the literature (see [12] for a survey), and has been identified as one of the primitive operators for database systems [7].

In this paper we study similarity join under edit distance. The edit distance between two strings x and y, denoted by $\mathrm{ED}(x, y)$, is defined to be the minimum number of edit operations (insertion, deletion and substitution) to transfer x to y. Formally, given a collection of strings $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ over alphabet Σ, a similarity threshold K, edit similarity (self)join outputs

$$\{(s_i, s_j) \mid s_i, s_j \in \mathcal{S}; i \neq j; \mathrm{ED}(s_i, s_j) \leq K\}.$$

*Both authors are supported in part by NSF CCF-1525024 and IIS-1633215.

KDD '17, Halifax, NS, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-4887-4/17/08...$15.00
DOI: 10.1145/3097983.3098003

For example, given strings ACCAT, CCAAT, GCCCT, CACGA, AACGG and $K = 2$, the output pairs will be (ACCAT, CCAAT), (ACCAT, GCCCT), (CACGA, AACGG).

Compared with the Hamming distance and token-based distances such as Cosine, Jaccard, Overlap and Dice, edit distance retains the information of the orderings of characters, and captures the best alignment of the two strings, which is critical to applications in bioinformatics, natural language processing and information retrieval. On the other hand, edit distance is computationally more expensive than Hamming and token-based distances: computing edit distance takes at least quadratic time under the SETH conjecture [2], while Hamming, Cosine, Jaccard, Overlap and Dice can be computed in linear time.

Due to its difficulty and usefulness, a large portion of the similarity join literature has been devoted to edit distance [1, 3, 5, 10, 15, 16, 18, 22–25]. However, we have observed that all the existing approaches fall short on long strings and relatively large thresholds. In the recent string similarity search/join competition, it was reported that "an error rate of 20% $\sim$ 25% pushes today's techniques to the limit" [21]. By 20% errors we mean that the distance threshold is set to be 20% of the string length. In fact the limit is reached much earlier on strings that are longer than those tested in the competition.

However, long strings and large thresholds are critical to many applications. For example, documents can contain hundreds of thousands of characters; the lengths of DNA sequences range from thousands to billions of bases. If we set a threshold that is too small, then we may end up getting zero output pair which is certainly not interesting.

Our Contribution. The main contribution of this paper is a novel approach of computing edit similarity joins that scales very well with the string length and the distance threshold. Different from all previous approaches which directly perform computations on the edit distance, we first embed the input strings from the edit space to the Hamming space, and then perform a filtering in the Hamming space using locality sensitive hashing.

Our algorithm, named `EmbedJoin`, is randomized and may introduce a small number of errors (95% - 99% recall, 100% precision in all of our experiments), but it significantly outperforms all the previous algorithms in both running time and memory usage on long strings and large thresholds. In particular, `EmbedJoin` scales very well up to error rate 20% on large datasets which is beyond the reach of existing algorithms.

Overview of Our Approach. Given two strings $x, y \in \Sigma^N$, the Hamming distance between x and y is defined to be $\mathrm{Ham}(x, y) = \sum_{i=1}^{N} \mathbf{1}(x_i \neq y_i)$. Our approach is built on the recent advance of metric embeddings for edit distance, and is very different from all of the previous approaches. In [6], it has been shown that there exists an embedding function $f : \Sigma^N \to \Sigma^{3N}$ such that given $x, y \in \Sigma^N$, we have with probability $1 - o(1)$ that [1]

$$\mathrm{ED}(x, y) \quad \leq \quad \mathrm{Ham}(f(x), f(y)),$$

and with probability at least 0.999 that

$$\mathrm{Ham}(f(x), f(y)) \quad \leq \quad O\left((\mathrm{ED}(x, y))^2\right).$$

We call this scheme the *CGK-embedding*, named after the initials of the authors in [6]. The details of the embedding algorithm will be illustrated in Section 3.1. We call

$$D(x, y) = \mathrm{Ham}(f(x), f(y))/\mathrm{ED}(x, y)$$

the *distortion* of the CGK-embedding on input (x, y). Note that if $\mathrm{ED}(x, y) \leq K$, then $1 \leq D(x, y) \leq O(K)$ with probability at least 0.99.

The high level idea of our approach is fairly simple: we first embed using CGK all the strings from the edit space to the Hamming space, and then perform a filtering step on the resulting vectors in the Hamming space using locality sensitive hashing (LSH) [9, 11]. LSH has the property that it will map a pair of items of small Hamming distance to the same bucket in the hash table with good probability, and map a pair of items of large Hamming distance to different buckets with good probability. The final step is to verify for each hash bucket B, and for all the strings hashed into B, whether their pairwise edit distances are at most K or not, by an exact dynamic programming based edit distance computation.

One may observe that the worst-case distortion of the CGK-embedding can be fairly large if the threshold K is large. However, we have observed that the practical performance of CGK-embedding is much better. We will give more discussions on this phenomenon in Section 3.1. To further reduce the distortion, we choose to run the embedding multiple times, and then for each pair of strings we choose the run with the minimum Hamming distance for the filtering. This minimization step does not have to be performed explicitly since we do not have to compute $\mathrm{Ham}(f(x), f(y))$ for all pairs of strings which is time consuming. We instead integrate this step with LSH for a fast filtering.

Finally, we note that since LSH is a dimension reduction step, LSH-based filtering naturally fits long strings (e.g., DNA sequences) which are our main interest. For short strings LSH-based filtering may not be the most effective approach and one may want to use different filtering methods. We also note Satuluri et al. [19] used LSH-based filtering for computing similarity joins under the Jaccard distance and the Cosine distance. Unfortunately there is no efficient LSH for edit distance, which is the motivation for us to first embed the strings to vectors in the Hamming space and then perform LSH.

2 RELATED WORK

Similarity Joins for Edit Distance. The edit similarity join problem has been studied extensively in the literature. We refer the readers to [12] for a comprehensive survey. A widely adopted approach to this problem is to first generate for each string a set of signatures/substrings. For example, in the q-gram signature, we generate all substrings of length q (e.g., when $q = 2$, the 2-grams of `ACCAT` is {`AC, CC, CA, AT`}). We then perform a filtering step based on the frequencies, positions and/or the contents of these substrings. The filtering step will give a set of candidate (similar) pairs, for each of which we use a dynamic programming algorithm for edit distance to verify its *exact* similarity. Concrete algorithms of signature-based approach include `GramCount` [10], `AllPair` [3], `FastSS` [5], `ListMerger` [15], `EDJoin` [25], `QChunk` [18], `VChunk` [24], `PassJoin` [16], and `AdaptJoin` [23]. We will briefly describe in Section 4.1 the best ones among these algorithms which we use as competitors to `EmbedJoin` in the experiments.

While different signature-based algorithms use different filtering methods, their common feature is to first compute some upper or lower bounds, and then prune those pairs (x, y) for which $g(sig(x), sig(y))$ is above or below the predetermined upper/lower bounds, where g is a predefined function, and $sig(x), sig(y)$ are signatures of x and y respectively. The main drawback of signature-based approach is that the information about the sequence ordering is somewhat lost when converting strings to a set of substrings. Another issue is that the precomputed upper/lower bounds may be too loose for effective pruning.

There are a few other approaches for computing edit similarity joins, such as trie-based algorithm `TrieJoin` [22], tree-based algorithm `M-Tree` [8], enumeration-based algorithm `PartEnum` [1]. However, as reported in [12], these algorithms are not very effective on datasets of long strings.

Similarity Joins for Other Metrics. Similarity joins have been studied for a number of other metrics [1, 3, 10, 15, 16, 23, 26, 27], including Cosine, Jaccard, Overlap and Dice. A survey of these works is beyond the scope of this paper, and we again refer reader to [12] for an overview.

Other Related Work on Edit Distance. Edit distance is also a notoriously difficult metric for sketching and embeddings, and very little is known in these frontiers. As mentioned, embedding enables us to study the similarity join problem in an easier metric space. On the other hand, if we can efficiently obtain small sketches of the input strings, then we can solve the similarity join problem on smaller inputs. Ostrovsky and Rabani proposed an embedding from the edit metric to the ℓ_1 metric with an $\exp(O(\sqrt{\log N \log\log N}))$ distortion [17] where N is the length of the string. A corresponding distortion lower bound of $\Omega(\log N)$ has been obtained by Kraugh-gamer and Rabani [13]. Recently Chakraborty et al. gives a weak embedding to the Hamming space [6] with an $O(K)$ distortion,[2] which serves as the main tool in our algorithm. For sketching, very recently Belazzougui and Zhang [4] proposed the first almost linear time sketching algorithm that gives a sketch of sublinear size (more precisely, $O(K^8 \log^5 N)$), which, unfortunately, is still too large to be useful in practice in its current form.

Computing edit distance in the RAM model has been studied for decades. It is well-known that the edit distance verification problem under distance threshold K can be solved in time $O(NK)$ by dynamic programming [20]. It has been further improved to

[1]The analysis in [6] in fact only gives $\mathrm{ED}(x, y)/2 \leq \mathrm{Ham}(f(x), f(y))$. However, as we shall describe in Algorithm 1, if we pad the embedded strings using a character that is not in the dictionary, then it is easy to show that $\mathrm{ED}(x, y) \leq \mathrm{Ham}(f(x), f(y))$ with probability $1 - o(1)$.

[2]In a "weak" embedding, the distortion holds for *each* pair of strings with constant probability, say, 0.99. In contrast, in a "strong" embedding, with probability 0.99 the distortion holds for *all* pairs of strings simultaneously.

Notation	Definition
$[n]$	$[n] = \{1, 2, \ldots, n\}$
K	Edit distance threshold
$\mathcal{S}$	The set of input strings
s_i	The i-th string in $\mathcal{S}$
$\|x\|$	Length of string x
n	Number of input strings, i.e., $n = \|\mathcal{S}\|$
N	Maximum length of strings in $\mathcal{S}$
Σ	Alphabet of strings in $\mathcal{S}$
r	Number of CGK-embeddings for each input string
t_i^ℓ	The output string generated by the ℓ-th CGK-embedding of s_i
z	Number of hash functions used in LSH for each string generated by CGK-embedding
m	Length of the LSH signature
f_j^ℓ	$f_j^\ell : \Sigma^N \to \Sigma^m$, the j-th ($j \in [z]$) LSH function for each string generated by the ℓ-th CGK-embedding
$\mathcal{D}_j^\ell$	The hash table corresponding to the LSH function f_j^ℓ

Table 1: Summary of Notations

$O(N + K^2)$ [14], but the algorithm in [14] employs suffix trees and thus may not have advantage in practice.

3 THE ALGORITHM

In this section we present our algorithm EmbedJoin. We will first illustrate the CGK-embedding and the LSH for the Hamming distance; these are the main tools that we shall use in EmbedJoin. We list in Table 1 a set of notations that will be used in the presentation.

3.1 The CGK-Embedding

We describe the CGK-embedding in Algorithm 1. Below we illustrate the main idea behind the CGK-embedding, which we believe is useful and important to understand the intuition of EmbedJoin. We note that the original algorithm in [6] was only described for binary strings, and it was mentioned that we can encode an alphabet Σ into binary codes using $\log |\Sigma|$ bits for each character. In our rewrite (Algorithm 1) we choose to use the alphabet Σ directly without the encoding. This may give some performance gain when the size of the alphabet is small.

Let N be the maximum length of all input strings in $\mathcal{S}$. The CGK-embedding maps a string $x \in \mathcal{S}$ to an output string $x' \in \Sigma^{3N}$ using a random bit string $R \in \{0,1\}^{3N|\Sigma|}$. We maintain a counter $i \in [1..|x|]$ pointing to the input string x, initialized to be 1. The embedding proceeds by steps $j = 1, \ldots, 3N$. At the j-th step, we first copy $x[i]$ to $x'[j]$. Next, with probability 1/2, we increase i by 1, and with the rest of the probability we keep i to be the same. At the point when $i > |x|$, if j is still no more than $3N$, we simply pad an arbitrary character outside the dictionary Σ (denoted by "$\perp$" in Algorithm 1) to make the length of x' to be $3N$. In practice this may introduce quite some overhead for short strings in the case that the string lengths vary significantly. We will discuss in Section 3.4 how to efficiently deal with input strings of very different lengths.

Algorithm 1 CGK-Embedding(s, R) [6]

```
Input: A string $x \in \Sigma^\eta$ for some $\eta \le N$, and a random string $R \in \{0,1\}^{3N|\Sigma|}$
Output: A string $x' \in \Sigma^{3N}$
1: Interpret $R$ as a set of functions
       $\pi_1, \ldots, \pi_{3N} : \Sigma \to \{0,1\}$; for the $k$-th char $\sigma_k$ in $\Sigma$,
       $\pi_j(\sigma_k) = R[(j-1) \cdot |\Sigma| + k]$
2: $i \leftarrow 1$
3: $x' \leftarrow \emptyset$
4: for $j \in [3N]$ do
5:     if $i \le |x|$ then
6:         $x' \leftarrow x' \odot x[i]$        ▷ the "$\odot$" denotes concatenation
7:         $i \leftarrow i + \pi_j(x[i])$
8:     else
9:         $x' \leftarrow x' \odot \perp$        ▷ "$\perp$" can be an arbitrary character outside $\Sigma$
10:    end if
11: end for
```

Now consider two input stings x and y. We use i_0 and i_1 as two counters pointing to x and y respectively. At the j-th step, we first copy $x[i_0]$ to $x'[j]$, and $y[i_1]$ to $y'[j]$, and then decide whether to increment i_0 and i_1 using the random bit string R. There are four possibilities: (1) only i_0 increments; (2) only i_1 increments; (3) both i_0 and i_1 increment; and (4) neither i_0 nor i_1 increments. Let $d = i_0 - i_1$ be the position *shift* of the two counters/pointers on the two strings. Note that if $x[i_0] = y[i_1]$, then only the cases (3) and (4) can happen, so that d will remain the same. Otherwise if $x[i_0] \ne y[i_1]$, then each case can happen with probability 1/4 – whether i_0 or i_1 will increment depends on the two random hash values $\pi_j(x[i_0])$ and $\pi_j(y[i_1])$. Thus with probability 1/4, 1/2 and 1/4, the value d will increment, remain the same, or decrement, respectively. Ignoring the case when the value d remains the same, we can view d as a (different) *simple random walk* on the integer line with 0 as the origin.

We now try to illustrate the high level idea of why CGK-embedding gives an $O(K)$ distortion. Let $u = |x|$ and $v = |y|$. Suppose that at some step j, letting $p = i_0(j)$ (the value of i_0 at step j) and $q = i_1(j)$, we have two tails $x[p..u] = \alpha \circ \tau$ and $y = y[q..v] = \tau$ where α, τ are two substrings and $|\alpha| = k \le K$. That is, we have k consecutive deletions in the optimal alignment of the two tails. Now if after a few random walk steps, at step $j' > j$, we have $p' = i_0(j') \ge p + k$, $q' = i_1(j') \ge q$ and $p' - q' = (p - q) + k$, then the two tails $x[p'..u]$ and $y[q'..v]$ can be perfectly aligned, and consequently the pairs of characters in the output strings x', y' will always be the same; in other words, they will *not* contribute to the Hamming distance from step j'.

Now observe that since the value of d changes according to a simple random walk, by the theory of random walk, with probability 0.999 it takes at most $O(k^2)$ steps for d to go from $(p - q)$ to $(p' - q')$ where $|(p - q) - (p' - q')| = k$. Therefore the number of steps j where $x'[j] \ne y'[j]$ is bounded by $O(k^2)$. This is roughly why $\mathrm{Ham}(x', y')$ can be bounded by $O(K^2)$ if $\mathrm{ED}(x, y) \le K$, and consequently the distortion can be bounded by $O(K)$.

Small Distortion is Good for Edit Similarity Join. We now explain why the distortion of the embedding matters. If we have an embedding f such that for any pair of input strings (x, y), the distortion of the embedding is upper bounded by D, then the set $\{(x, y) \mid \text{Ham}(f(x), f(y)) \leq D \cdot K\}$ will include all pairs (x, y) such that $\text{ED}(x, y) \leq K$. Therefore a small D can help to reduce the number of false positives, and consequently reduce the verification time which typically dominates the total running time.

Why CGK-embedding Does Better in Practice? Although the worst-case distortion of CGK-embedding can be large when $\text{ED}(x, y)$ is large, we have observed that its practical performance on the datasets that we have tested is much better. While it is difficult to fully understand this phenomenon without a thorough investigation of the actual properties of the datasets, we can think of the following reasons.

First, if a set of z edits fall into an interval of length $O(z)$, *and* the difference between the numbers of insertions and deletions among the z edits is at most $O(\sqrt{z})$ (substitutions do not matter), then with probability 0.999 after $O(z)$ walk steps the random walk will re-synchronize. In other words, the distortion of the embedding is $O(1)$ with probability 0.999 on this cluster of edits. We have observed that in our protein/genome datasets (Section 4.1) the edits are often clustered into small intervals; in each cluster most edits are substitutions, and consequently the difference between the numbers of insertions and deletions is small.

Second, in the task of differentiating similar pairs of strings and dissimilar pairs of strings, as long as the distance gap between strings is preserved after the embedding, the distortion of CGK-embedding will not affect the performance by much. In particular, when the distortion of CGK-embedding is $\Theta(k)$ (which is very likely when edits are well separated), the embedding actually *amplifies* the distance gap between similar and dissimilar pairs, which makes the next LSH step easier.

To further improve the effectiveness of the CGK-embedding, we run the embedding multiple times and then take the one with the minimum Hamming distance. That is, we choose the run with the best distortion. This is just a heuristic, and cannot improve the distortion by much in theory, but we have observed that for the real-world datasets that we have tested, repeating and then taking the minimum does help to reduce the distortion. In Figure 1 we depicted the best distortions under different numbers of runs of the CGK-embedding on a real-world genome dataset.

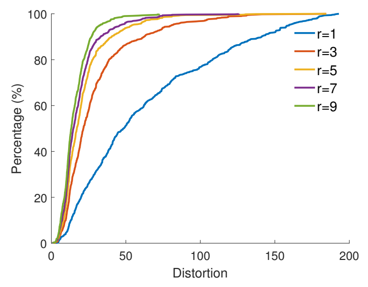


Figure 1: The CDF of the best distortions of 1000 random pairs strings from the GEN50kS dataset, under different numbers of CGK-embeddings (value r)

3.2 LSH for the Hamming Distance

Our second tool is the LSH for the Hamming distance, introduced in [9, 11] for solving nearest neighbor problems. We first give the definition of LSH. By $h \in_r \mathcal{H}$ we mean sampling a hash function h randomly from a hash family $\mathcal{H}$.

Definition 3.1. (Locality Sensitive Hashing [9]) Let U be the item universe, and $d(\cdot, \cdot)$ be a distance function. We say a hash family $\mathcal{H}$ is (l, u, p_1, p_2)-sensitive if for any $x, y \in U$

- if $d(x, y) \leq l$, then $\mathbf{Pr}_{h \in_r \mathcal{H}}[h(x) = h(y)] \geq p_1$,
- if $d(x, y) \geq u$, then $\mathbf{Pr}_{h \in_r \mathcal{H}}[h(x) = h(y)] \leq p_2$.

We have the following LSH for the Hamming distance.

Theorem 3.2. *(Bit-sampling LSH for Hamming [9]) For the Hamming distance over vectors in Σ^N, for any $d > 0, c > 1$, the family*

$$\mathcal{H}_N = \{v_i : v_i(b_1, \ldots, b_N) = b_i \mid i \in [N]\}$$

is $(d, cd, 1 - d/N, 1 - cd/N)$-sensitive.

We can use the standard AND-OR amplification method[3] to amplify the gap between p_1 and p_2. We first concatenate m (m is a parameter) hash functions, and define

$$f = h_1 \circ h_2 \circ \ldots \circ h_m \text{ where } \forall i \in [m], h_i \in_r \mathcal{H},$$

such that for $x \in U$, $f(x) = (h_1(x), h_2(x), \ldots, h_m(x))$ is a vector of m bits. Let $\mathcal{F}(m)$ be the set of all such hash functions f. We then define (for a parameter z)

$$g = f_1 \vee f_2 \vee \ldots \vee f_z, \text{ where } \forall j \in [z], f_j \in_r \mathcal{F}(m),$$

such that for $x, y \in U$ $g(x) = g(y)$ if and only if there is at least one $j \in [z]$ for which $f_j(x) = f_j(y)$. Easy calculation shows that g is

$$\left(d, cd, 1 - \left(1 - (d/N)^m\right)^z, 1 - \left(1 - (cd/N)^m\right)^z\right)$$

sensitive. By appropriately choosing the parameters m, z we can amplify the gap between p_1 and p_2.

We comment that this vanilla version of LSH is enough for our applications, and its fast time performance fits our needs well.

3.3 Our EmbedJoin Algorithm

Now we are ready to describe EmbedJoin, which is presented in Algorithm 3 using Algorithm 2 as a subroutine. We explain them in words below; a running example for EmbedJoin can be found in the Appendix of the full version [28].

In the preprocessing we generate $r \times z$ hash tables $\mathcal{D}_j^\ell$ ($\ell \in [r], j \in [z]$) implicitly by sampling $r \times z$ random hash functions f_j^ℓ ($\ell \in [r], j \in [z]$) from $\mathcal{F}(m)$ (defined in Section 3.2). We then CGK-embed each string $s_i \in \mathcal{S}$ for r times, getting t_i^ℓ ($\ell \in [r]$).

We now describe our main algorithm. As previous algorithms, EmbedJoin has two stages: it first finds a small set of candidate pairs, and then verifies each of them using exact edit-distance computation via dynamic programming. We use the algorithm for computing edit distance in [16] for the second step. In the rest of this section we explain the first filtering step.

[3]See, for example, https://en.wikipedia.org/wiki/Locality-sensitive_hashing.

Algorithm 2 Preprocessing ($\mathcal{S}$, r, z, m)

```
Input: Set of input strings $\mathcal{S} = \{s_1, \ldots, s_n\}$, and parameters $r$, $z$ and $m$ described in Table 1
Output: Strings in $\mathcal{S}$ in the sorted order, strings after CGK-embedding $\{t_i^\ell \mid \ell \in [r], i \in [n]\}$, and hash tables $\{\mathcal{D}_j^\ell \mid \ell \in [r], j \in [z]\}$.
 1: Sort $\mathcal{S}$ first by string length increasingly, and second by the alphabetical order.
 2: for each $\ell \in [r]$ do
 3:     for each $j \in [z]$ do
 4:         Initialize hash table $\mathcal{D}_j^\ell$ by generating a random hash function $f_j^\ell \in \mathcal{F}(m)$
 5:     end for
 6: end for
 7: for each $\ell \in [r]$ do
 8:     Generate a random string $R^\ell \in \{0,1\}^{3N|\Sigma|}$
 9:     for each $s_i \in \mathcal{S}$ do
10:         $t_i^\ell \leftarrow$ CGK-Embedding($s_i$, $R^\ell$)
11:     end for
12: end for
```

Algorithm 3 EmbedJoin ($\mathcal{S}$, K, r, z, m)

```
Input: Set of input strings $\mathcal{S} = \{s_1, \ldots, s_n\}$, distance threshold $K$, and parameters $r$, $z$ and $m$ described in Table 1
Output: $O \leftarrow \{(s_i, s_j) \mid s_i, s_j \in \mathcal{S}; i \neq j; \mathrm{ED}(s_i, s_j) \leq K\}$
 1: Preprocessing($\mathcal{S}, r, z, m$)                         ▷ Using Algorithm 2
 2: $C \leftarrow \emptyset$                                        ▷ Collection of candidate pairs
 3: for each $s_i \in \mathcal{S}$ (in the sorted order) do
 4:     for each $\ell \in [r]$ do
 5:         for each $j \in [z]$ do
 6:             for each string $s$ stored in the $f_j^\ell(t_i^\ell)$-th bucket of table $\mathcal{D}_j^\ell$ do
 7:                 if $|s_i| - |s| \leq K$ then
 8:                     $C \leftarrow C \cup (s, s_i)$
 9:                 else
10:                     Remove $s$ from $\mathcal{D}_j^\ell$
11:                 end if
12:             end for
13:             Store $s_i$ in the $f_j^\ell(t_i^\ell)$-th bucket of $\mathcal{D}_j^\ell$
14:         end for
15:     end for
16:     Remove duplicate pairs in $C$
17: end for
18: for each $(x, y) \in C$ do
19:     if $\mathrm{ED}(x, y) \leq K$ then                ▷ Using the algorithm in [16]
20:         $O \leftarrow O \cup (x, y)$
21:     end if
22: end for
```

The main idea of the filtering step is fairly straightforward. We use LSH to find all pairs (s_i, s_j) for which there exists an $\ell \in [r]$ such that t_i^ℓ and t_j^ℓ are hashed into the same bucket by at least one of the hash functions $f_j^\ell \in \mathcal{F}(m)$ $(j \in [z])$. In other words, for at least one of the r CGK-embeddings, the output pairs corresponding to s_i and s_j are identified to be similar by at least one of the z LSH functions. Recall that we do r repetitions of CGK-embedding to achieve a good distortion ratio (see the discussion in Section 3.1), and we use z LSH functions from $\mathcal{F}(m)$ to amplify the gap between p_1 and p_2 in the definition of LSH to reduce false positives/negatives (see the discussion in Section 3.2).

In the actual implementation, we use a sliding window to speed-up the filtering: We first sort the input strings in $\mathcal{S}$ according to their lengths increasingly (breaking ties by the alphabetical orders of the strings). We then process them one by one. If $s_i \in S$ is hashed into some bucket B in the hash table, when fetching each string s in B we first test whether $|s_i| - |s| \leq K$ (Line 7). If not, we can immediately conclude $\mathrm{ED}(s, s_i) > K$, and consequently $\mathrm{ED}(s, s_{i'}) > K$ $(i' > i)$ for all the future strings $s_{i'} \in \mathcal{S}$, since we know for sure that $|s_{i'}| - |s| > K$ due to the sorted order. We thus can safely delete s from bucket B (Line 10). Otherwise we add (s, s_i) to our candidate set C. After these we store s_i in bucket B for future comparisons. Note that each pair (s_i, s_j) can potentially be added into C multiple times by different LSH collisions, we thus do a deduplication at Line 16.

There are two implementation details that we shall mention. First, in the preprocessing we do not need to generate the whole t_i^ℓ, but just those m bits that will be used by each of the z LSH functions. This reduces the space usage from $3N \cdot r \cdot n$ to $z \cdot m \cdot r \cdot n$. Second, It is time/space prohibited to generate the hash table $\mathcal{D}_j^\ell$ whose size is $|\Sigma|^m$. We adopt the standard two-level hashing implementation of LSH: For a signature in Σ^m, we first convert it into a vector $u \in \{1, \ldots, |\Sigma|\}^m$ in the natural way. We then generate a random vector $v \in \{0, \ldots, P-1\}^m$ where $P > 1,000,000$ is a prime we choose that fits our datasets in experiments. Finally, the second level hash function returns $\langle u, v \rangle \bmod P$, where $\langle \cdot, \cdot \rangle$ denotes the inner product.

Choices of parameters. There are three parameters m, z, r in `EmbedJoin` that we need to specify. Recall that m is the length of the LSH signature, or, the number of primitive hash functions $h \in \mathcal{H}$ we use in each $f \in \mathcal{F}(m)$; and z is the number of LSHs we use for each string generated by CGK-embedding. The larger z and m are, the better LSH performs in terms of accuracy and filtering effectiveness. The product $m \cdot z$ will contribute to the total running time of the algorithm. On the other hand, r is number of CGK-embeddings we perform for each input string. The larger r we use, the smaller distortion we will get (see Figure 1).

The concrete choices of m, z and r depend on the data size, distance thresholds, computation time/space budget and accuracy requirements. For our datasets we have tested a number of parameter combinations. We refer readers to Section 4.2 for some statistics. We have observed that $r = z = 7$, and $m = 15 - \lfloor \log_2 \frac{100K}{N} \rfloor$ are good choices to balance the resource usage and the accuracy.

Running time. The preprocessing step takes time $O(r \cdot z \cdot P + r \cdot n \cdot 3N\,|\Sigma|)$. The time cost of LSH-based filtering depends on the usefulness of the sliding window pruning; in the worst case it is $O(nrzm)$ where m counts the cost of evaluating a hash function $f \in \mathcal{F}(m)$. Finally, the verification step costs $O(NK \cdot Z)$ where Z is the number of candidate pairs after LSH-based filtering.

3.4 Further Speed-up

Note that in the CGK-Embedding (Algorithm 1), we always pad the output strings x' up to length $3N$, where $N = \max_{i \in [n]}\{|s_i|\}$. This approach is not very efficient for datasets containing strings with very different lengths (for example, our datasets UNIREF and TREC; see Section 4.1), since we need to pad a large number of "$\perp$" to the output strings which can be a waste of time. For example, for two strings s_1 and s_2 where $|s_1|, |s_2| \ll N$, if we map them to bit vectors s_1' and s_2' of size $3N$, then most of the aligned pairs in s_1' and s_2' are $(\perp, \perp)$s which carry almost no information. Then if we use bit-sampling LSH for the Hamming distance we need a lot of samples in order to hit the interesting region, that is, the coordinates of strings in s_1' and s_2' where at least one of the two characters is *not* "$\perp$". This is time and space expensive. We propose two ways to handle this issue.

Grouping. We first partition the set of strings of $\mathcal{S}$ to $(N'/K - 1)$ groups where $N' = \lceil N/K \rceil \cdot K$. The i-th group contains all the strings of lengths $((i-1)K, (i+1)K]$. Note that each string will be included in two groups (the redundancy), and every pair of strings with distance at most k will both be included in at least one of the groups. We then apply EmbedJoin on each group, and union the outputs at the end. Due to the redundancy this approach may end up evaluating at most twice of the total number of candidates.

Truncation. The second method is to use truncation, that is, we truncate each embedded string of size $3N$ to $3 \cdot avg(\mathcal{S})$, where $avg(\mathcal{S})$ is the average length of the strings in $\mathcal{S}$. We then apply EmbedJoin on all the truncated strings. Note that after truncation we essentially assume that all the bits after the $(3avg(\mathcal{S}))$-th position in the embedded strings are the same, and thus truncation will *not* increase the Hamming distance of any pair of strings, and consequently will not introduce any false negative. It can introduce some false positives but this is not a problem since we have a verification step at the end to remove all the false positives.

Our experimental results (see the full version [28]) show that truncation always has the better performance than grouping on our tested datasets. Therefore in the rest of the paper we always use truncation.

4 EXPERIMENTS

In this section we present our experimental studies. After listing the datasets and tested algorithms, we first give an overview of the performance of EmbedJoin. We then compare it with the existing best algorithms. Finally, we show the scalability of EmbedJoin in the ranges that the existing best algorithms cannot reach.

4.1 The Setup

Datasets. We tested the algorithms in three publicly available real world datasets.

UNIREF: a dataset of UniRef90 protein sequence data from UniProt project.[4] Each sequence is an array of amino acids coded in uppercase letters. We first remove sequences whose lengths are smaller than 200, and then extract the first 400,000 protein sequences.

[4]Available in http://www.uniprot.org/

Datasets	n	Avg Len	Min Len	Max Len	$\|\Sigma\|$
UNIREF	400000	445	200	35213	25
TREC	233435	1217	80	3947	37
GEN50kS	50000	5000	4844	5109	4
GEN20kS	20000	5000	4847	5109	4
GEN20kM	20000	10000	9849	10090	4
GEN20kL	20000	20000	19838	20098	4
GEN80kS	80000	5000	4844	5109	4
GEN320kS	320000	5000	4841	5152	4

Table 2: Statistics of tested datasets

TREC: a dataset of references from Medline (an online medical information database) consisting of titles and abstracts from 270 medical journals.[5] We first extract and concatenate title, author, and abstract fields, and then convert punctuations into white spaces and letters into their upper cases.

GEN50kS GEN20kS GEN20kM GEN20kL GEN80kS GEN320kS: datasets of human genome sequences of 50 individuals obtained from the *personal genomes project*,[6] and the reference sequence is obtained from GRCh37 assembly. We choose to use Chromosome 20. We partition the long DNA sequences into shorter substrings according to the indices of the reference sequence to construct the datasets listed above. The names of datasets can be read as 'GEN $\circ$ number of strings ($20k$ to $320k$) $\circ$ string length (S $\approx$ 5k, M $\approx$ 10k, L $\approx$ 20k)'.

We summarize the statistics of our datasets in Table 2.

Tested Algorithms. We implemented our algorithm EmbedJoin in C++, and complied using GCC 5.4.0 with O3 flag. We compared our algorithms with PassJoin[16], EDJoin[25], AdaptJoin[23], and QChunk[18], whose binary codes were downloaded from the authors' project websites. We choose these competing algorithms based on the recommendations of the experimental study [12] and the similarity search/join competition [21]. We believe that these are the best existing algorithms for edit similarity joins.

Measurements. We report three types of measurements in our experiments: *accuracy*, *memory usage* and *running time*. Recall that EmbedJoin only have false negatives; the *accuracy* we report is number of output pairs returned by EmbedJoin divided by the ground truth returned by other exact competing algorithms. The memory usage we report is the maximum memory usage of a program during its execution. As mentioned, the competing algorithms may use different filtering methods or different parameters. We always choose the *best* combinations for comparisons. To make the comparison fair we have counted the time used for all the preprocessing steps.

Computing Environment. All experiments were conducted on a Dell PowerEdge T630 server with 2 Intel Xeon E5-2667 v4 3.2GHz CPU with 8 cores each, and 256GB memory.

4.2 Performance Overview of EmbedJoin

In this section we present an overview of the performance of EmbedJoin. All the results are the average of five independent runs.

[5]Available in http://trec.nist.gov/data/t9_filtering.html
[6]Available in http://personalgenomes.org/

	$r = 5$			$r = 7$			$r = 9$		
Accuracy	$z = 3$	$z = 5$	$z = 7$	$z = 3$	$z = 5$	$z = 7$	$z = 3$	$z = 5$	$z = 7$
$m = 5$	94.5%	97.4%	98.6%	96.9%	99.0%	99.5%	98.5%	99.4%	99.7%
$m = 7$	91.6%	94.0%	95.6%	95.2%	97.2%	98.4%	96.4%	98.4%	99.1%
$m = 9$	90.1%	90.9%	92.9%	90.7%	94.7%	96.1%	92.9%	96.2%	97.6%

Table 3: Accuracy of EmbedJoin, UNIREF **dataset,** $K = 20$

	$r = 5$			$r = 7$			$r = 9$		
Accuracy	$z = 3$	$z = 5$	$z = 7$	$z = 3$	$z = 5$	$z = 7$	$z = 3$	$z = 5$	$z = 7$
$m = 8$	91.3%	94.2%	95.6%	91.3%	94.2%	95.6%	95.6%	95.6%	98.6%
$m = 10$	90.0%	92.8%	92.8%	91.3%	94.2%	94.2%	92.8%	94.2%	95.6%
$m = 12$	90.0%	90.0%	91.3%	90.0%	90.0%	91.3%	91.3%	92.8%	94.2%

Table 4: Accuracy of EmbedJoin, TREC **dataset,** $K = 40$

	$r = 5$			$r = 7$			$r = 9$		
Accuracy	$z = 3$	$z = 5$	$z = 7$	$z = 3$	$z = 5$	$z = 7$	$z = 3$	$z = 5$	$z = 7$
$m = 11$	99.1%	99.5%	99.6%	99.5%	99.6%	99.8%	99.6%	99.8%	99.8%
$m = 13$	99.0%	99.3%	99.4%	99.3%	99.5%	99.6%	99.5%	99.7%	99.8%
$m = 15$	98.8%	99.1%	99.3%	99.1%	99.4%	99.5%	99.4%	99.6%	99.7%

Table 5: Accuracy of EmbedJoin, GEN50kS **dataset,** $K = 100$

Due to the space constraints we delay some of the experimental results (e.g., Figures and Tables) of this section to the full version [28], and only summarize the performance of our algorithms.

Accuracy. In Table 3, 4 and 5 we study how different parameters (r, z, m) influence the accuracy of EmbedJoin. We vary r in $\{5, 7, 9\}$, z in $\{3, 5, 7\}$, and choose slightly different values for m on different datasets (the choices of m largely depend on the string length and the distance threshold K). We observe that the accuracy of EmbedJoin is $90.1 \sim 99.7\%$ on UNIREF, $90.0 \sim 98.6\%$ on TREC, and $98.8 \sim 99.8\%$ in GEN50kS.

We note that the accuracy of EmbedJoin increases with r and z, and decreases with m. This is consistent with the theory. When r and z increase, we use more hash functions (recall that the total number of hash functions used is $r \cdot z$), and thus each pair of strings have more chance to be hashed into the same bucket in at least one of the hash tables. Similarly, when m decreases, each LSH function has larger collision probability. Of course, the increase of the collision probability will always introduce more false positives, and consequently increase the verification time. Using more hash functions/tables will also increase the space usage.

Time and Space. In Table 6 we study how different parameters (r, z, m) influence the running time of EmbedJoin in the GEN50kS dataset. We note that the running time increases when r and z increase, decreases when m increases. This is just the opposite to what we have observed for accuracy, and is consistent to the theory that increasing the collision probability will introduce more false positives/candidates and thus increase the verification time.

Due to the space constraints we refer readers to the full version for the statistics of memory usage of EmbedJoin under different parameters (r, z, m). We observe that the memory usage increases when r and z increases. This is because when r and z increase we need to store more hash tables *and* we will have more candidate pairs to verify. When m increases, the memory usage stays the same or slightly increases. There are two kinds of mutually exclusive forces that affect this. On the one hand, when m increases the size of each hash signature increases. On the other hand, when m increases the number of candidate pairs decreases. From what we have observed, the first force generally dominates the second.

More Studies. We also did the followings. (1) Recorded the change of accuracy of EmbedJoin under different thresholds K and parameters (r, z, m). (2) Recorded and compared the running time of EmbedJoin on different modules of the algorithms: reading the input and CGK-embedding, performing LSH, and verification. (3) Studied how different parameters (r, z, m) influence the number of candidates generated by EmbedJoin. Due to the space constraints we refer readers to the full version [28] for details.

4.3 A Comparison with Existing Algorithms

In this section we compare EmbedJoin with the existing best algorithms introduced in Section 4.1. We note that in some figures some data points for competing algorithms are missing, which is either because these algorithms have implementation limitations (returned wrong answers or triggered memory overflow) or they cannot finish in 24 hours.

Scalability on the Threshold Distance. Figure 2 shows the running time of different algorithms when varying the distance threshold K on UNIREF, TREC and GEN50kS. In all experiments we always guarantee the accuracy of EmbedJoin is above 95% on UNIREF and TREC, and is above 99% on GEN50kS.

	$r = 5$			$r = 7$			$r = 9$		
Time(s)	$z = 3$	$z = 5$	$z = 7$	$z = 3$	$z = 5$	$z = 7$	$z = 3$	$z = 5$	$z = 7$
$m = 11$	143.9	149.1	159.2	152.4	159.8	172.3	158.8	174.4	191.4
$m = 13$	132.6	134.1	137.3	138.4	140.9	145.5	143.4	147.0	154.0
$m = 15$	131.6	133.5	134.0	134.2	138.3	139.0	137.6	142.6	147.6

Table 6: Running time of `EmbedJoin`, `GEN50kS` dataset, $K = 100$

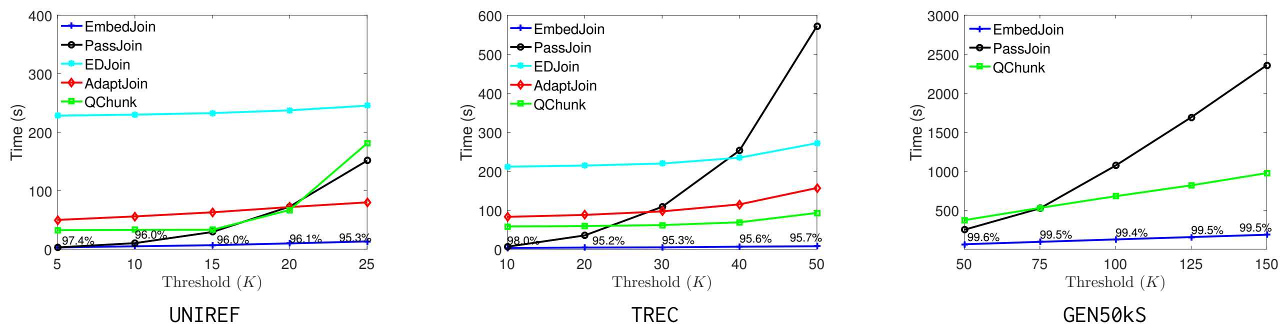


Figure 2: Running time, varying K. Percentages on the curves for `EmbedJoin` are its accuracy

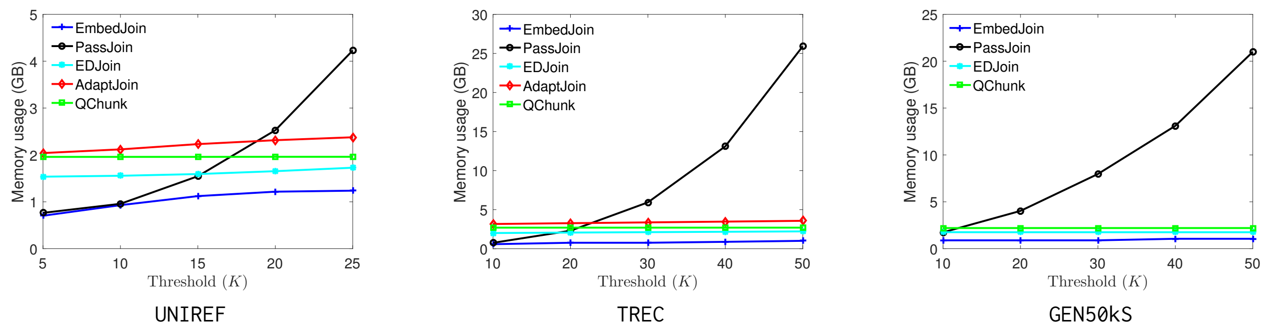
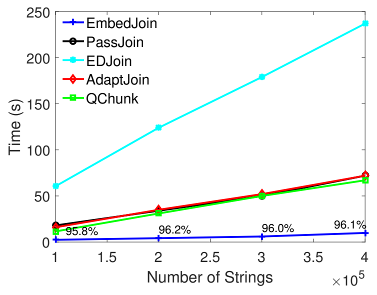


Figure 3: Memory usage, varying K

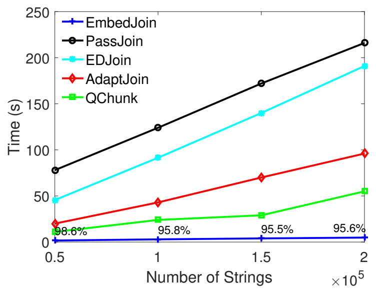
We observe that `EmbedJoin` always has the best performance: the running time of `EmbedJoin` is better than the best existing algorithm by a factor of 11.3 on `UNIREF` ($K = 25$), 12.1 in `TREC` ($K = 50$), and 5.2 on `GEN50kS` ($K = 150$). The `PassJoin` algorithm does not scale well on K: when K increases, the running time jumps sharply. This may due to the fact that the time complexity in the filtering step of `PassJoin` is $O(nK^3)$ – a cubic dependence on K. The other three algorithms, `EDJoin`, `AdaptJoin` and `QChunk`, are all based on q-gram or its variants; they generally have similar running time curves, which rise much slower compared with `PassJoin` when K increases. One exception is that on the `UNIREF` dataset the running time of `QChunk` increases sharply when K passes 20, which may due to the sudden increase of the number of candidate pairs that `QChunk` produces. On `GEN50kS`, the running time of `EDJoin` is too large (> 10000s when $K = 50$) and thus does not fit the figure, and `AdaptJoin` reports erroneous results.

We would like to comment that the *output size* has little to do with the running time of the algorithms. For example, for the five distance thresholds on the `UNIREF` dataset in Figure 2, the corresponding output sizes are 707, 1377, 2154, 3184, 5864; the numbers increase quickly. On the other hand, for the five distance thresholds on the `TREC` dataset the corresponding output sizes are 52, 63, 64, 69, 70; the numbers increase slowly. But the running time curves of the test algorithms on the two datasets generally follow the similar trends. The major components that affect the running time are the time spent on the filtering and the number of *candidate pairs* after the filtering step for verification. We refer readers to the full version of this paper for statistics on the running time of `EmbedJoin` on different modules.

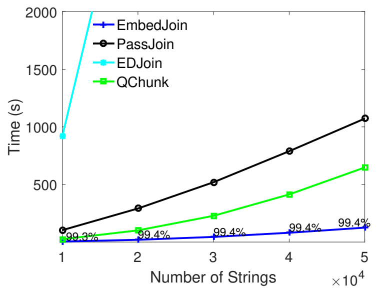
Figure 3 shows the memory usages of different algorithms in the same settings as Figure 2. We note that the memory used by `EmbedJoin` is always the smallest among all. This is largely because the hash tables used by `EmbedJoin` take less space than the signatures and indexes built by other algorithms. The memory usage of `PassJoin` is also small at the beginning, but deteriorates fast when K increases. The three q-gram based algorithms have similar trends in memory usage.



UNIREF ($K = 20$)

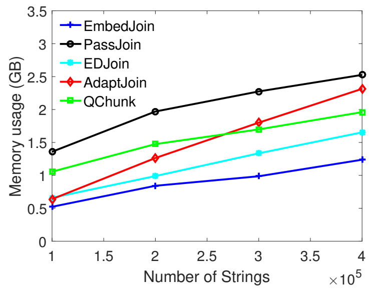


TREC ($K = 40$)

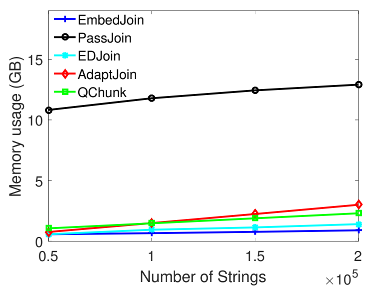


GEN50kS ($K = 100$)

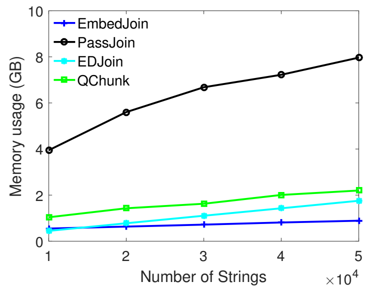
Figure 4: Running time, varying n. Percentages on the curves for EmbedJoin are its accuracy



UNIREF ($K = 20$)



TREC ($K = 40$)



GEN50kS ($K = 100$)

Figure 5: Memory usage, varying the input size n

Scalability on the Input Size. Figure 4 shows the running time of different algorithms on the UNIREF, TREC and GEN50kS datasets when varying input size n. The trends of the running time of all algorithms are similar; they increase almost linearly with respect to n. It is clear that EmbedJoin performs much better than all the other algorithms: EmbedJoin performs better than the best existing algorithm by a factor of 6.8 on UNIREF ($N = 4 \times 10^5$), 11.5 on TREC ($N = 2 \times 10^5$), and 5.1 on GEN50kS ($N = 5 \times 10^4$).

Figure 5 shows the memory usages of different algorithms in the same settings as Figure 4. The trends of the memory usages of all algorithms are similar; they increase almost linearly with respect to n. It is clear that EmbedJoin always performs the best.

The Ultimate Scalability of EmbedJoin. Finally, we present a set of experiments that distinguish EmbedJoin from all the competing algorithms. We test all the algorithms on longer strings (length ranges from 5,000 to 20,000) with larger distance thresholds (1% ∼ 20% of the corresponding string length). The numbers of strings in the datasets range from 20,000 to 320,000. For EmbedJoin we fix $r = z = 7$, and set $m = 15 - \lfloor \log_2 x \rfloor$ where $x\%$ is the threshold. Result points are only depicted for those that can finish in 24 hours, *and* return correct answers.

When varying the string length N (see Figure 6), there are three algorithms that can produce data points in the GEN20kS dataset: EDJoin can report answer up to the 2% distance threshold, and PassJoin and QChunk can go up to 8%. We observe a sharp time jump of QChunk from 4% to 8% – at the 8% distance threshold QChunk barely finished within 24 hours. On GEN20kL, unfortunately, the program for QChunk that we have used cannot produce any data point due to memory overflow. PassJoin only succeeds at the 1% distance threshold.

When varying the number of input strings n (see Figure 7; the first subfigure of Figure 7 is simply a repeat of the first subfigure of Figure 6), all the other computing algorithms cannot produce anything on GEN320kS. PassJoin manages to produce results on GEN80kS up to 4% distance threshold. On the other hand, EmbedJoin scales smoothly on all the datasets.

To summarize, it is clear that on large datasets with long string, EmbedJoin performs much better than all the competing algorithms, and scales well up to distance threshold 20%. Unfortunately, we do not know the exact accuracy of EmbedJoin in many points since other exact computation algorithms cannot finish, but from what we have observed on shorter strings and smaller distance thresholds, we would expect that its accuracy will be consistently high.

5 CONCLUSION

We propose an algorithm named EmbedJoin for computing edit similarity join, one of the most important operations in database systems. Different from all previous approaches, we first embed the input strings from the edit space to the Hamming space, and then try to perform a filtering (for reducing candidate pairs) in the Hamming space where efficient tools like locality sensitive hashing are available. Our experiments have shown that EmbedJoin significantly outperforms, at a very small cost of accuracy, all previous best algorithms on long strings and large thresholds.

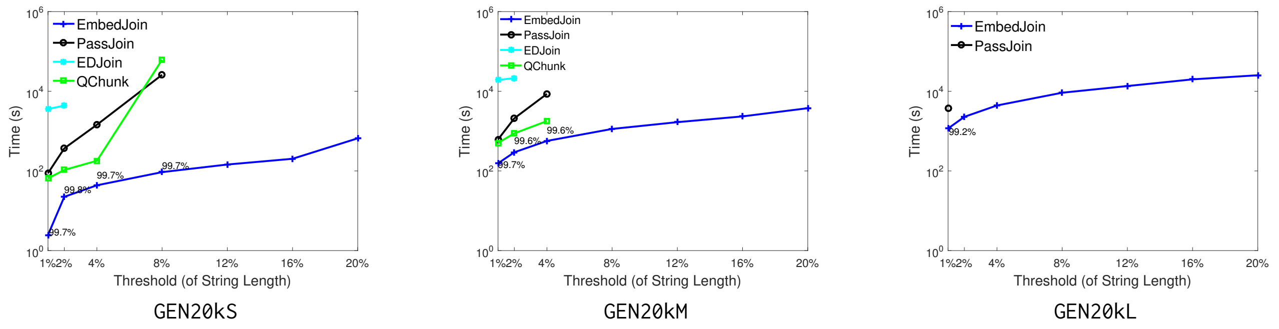


Figure 6: Scalability on string length. Percentages on the curves for `EmbedJoin` are its accuracy

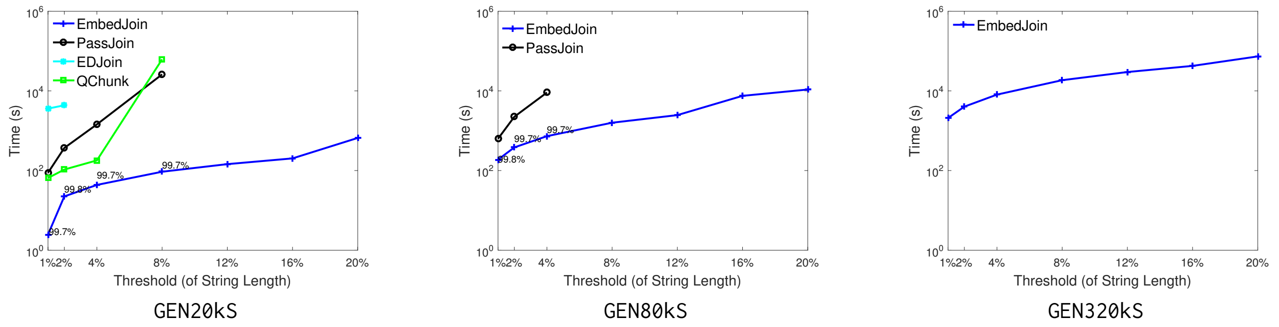


Figure 7: Scalability on number of strings. Percentages on the curves for `EmbedJoin` are its accuracy

6 ACKNOWLEDGMENT

The authors would like to thank Djamal Belazzougui for many helpful discussions, and Haixu Tang and Diyue Bu for their help on preparing the human genome datasets.

REFERENCES

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *VLDB*. 918–929.

[2] Arturs Backurs and Piotr Indyk. 2015. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false). In *STOC*. 51–58.

[3] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *WWW*. 131–140.

[4] Djamal Belazzougui and Qin Zhang. 2016. Edit Distance: Sketching, Streaming and Document Exchange. In *FOCS*. to appear.

[5] Thomas Bocek, Ela Hunt, Burkhard Stiller, and Fabio Hecht. 2007. *Fast similarity search in large dictionaries*. University.

[6] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. 2016. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *STOC*. 712–725.

[7] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*. 5.

[8] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB*. 426–435.

[9] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.

[10] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate String Joins in a Database (Almost) for Free. In *VLDB*. 491–500.

[11] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.

[12] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String Similarity Joins: An Experimental Evaluation. *PVLDB* 7, 8 (2014), 625–636.

[13] Robert Krauthgamer and Yuval Rabani. 2009. Improved Lower Bounds for Embeddings into L_1. *SIAM J. Comput.* 38, 6 (2009), 2487–2498.

[14] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. 1998. Incremental String Comparison. *SIAM J. Comput.* 27, 2 (1998), 557–582.

[15] Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *ICDE*. 257–266.

[16] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. PASS-JOIN: A Partition-based Method for Similarity Joins. *PVLDB* 5, 3 (2011), 253–264.

[17] Rafail Ostrovsky and Yuval Rabani. 2007. Low distortion embeddings for edit distance. *J. ACM* 54, 5 (2007).

[18] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*. 1033–1044.

[19] Venu Satuluri and Srinivasan Parthasarathy. 2012. Bayesian Locality Sensitive Hashing for Fast Similarity Search. *PVLDB* 5, 5 (2012), 430–441.

[20] Esko Ukkonen. 1985. Algorithms for Approximate String Matching. *Information and Control* 64, 1-3 (1985), 100–118.

[21] Sebastian Wandelt, Dong Deng, Stefan Gerdjikov, Shashwat Mishra, Petar Mitankin, Manish Patil, Enrico Siragusa, Alexander Tiskin, Wei Wang, Jiaying Wang, and Ulf Leser. 2014. State-of-the-art in string similarity search and join. *SIGMOD Record* 43, 1 (2014), 64–76.

[22] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2010. Trie-Join: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints. *PVLDB* 3, 1 (2010), 1219–1230.

[23] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*. 85–96.

[24] Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. 2013. VChunkJoin: An Efficient Algorithm for Edit Similarity Joins. *IEEE Trans. Knowl. Data Eng.* 25, 8 (2013), 1916–1929.

[25] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.

[26] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Efficient similarity joins for near duplicate detection. In *WWW*. 131–140.

[27] Jiaqi Zhai, Yin Lou, and Johannes Gehrke. 2011. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *SIGMOD*. 997–1008.

[28] Haoyu Zhang and Qin Zhang. 2017. EmbedJoin: Efficient Edit Similarity Joins via Embeddings. *CoRR* abs/1702.00093 (2017).