String Similarity Joins: An Experimental Evaluation

Yu Jiang[†] Guoliang Li[†] Jianhua Feng[†] Wen-Syan Li[‡]
[†]Department of Computer Science, Tsinghua University, Beijing, China
[‡]SAP Lab, Shanghai, China
y-jiang12@mails.tsinghua.edu.cn; {liguoliang, fengjh}@tsinghua.edu.cn; wen-syan.li@sap.com

ABSTRACT

String similarity join is an important operation in data integration and cleansing that finds similar string pairs from two collections of strings. More than ten algorithms have been proposed to address this problem in the recent two decades. However, existing algorithms have not been thoroughly compared under the same experimental framework. For example, some algorithms are tested only on specific datasets. This makes it rather difficult for practitioners to decide which algorithms should be used for various scenarios. To address this problem, in this paper we provide a comprehensive survey on a wide spectrum of existing string similarity join algorithms, classify them into different categories based on their main techniques, and compare them through extensive experiments on a variety of real-world datasets with different characteristics. We also report comprehensive findings obtained from the experiments and provide new insights about the strengths and weaknesses of existing similarity join algorithms which can guide practitioners to select appropriate algorithms for various scenarios.

1. INTRODUCTION

Given two collections of strings, e.g., products and movie names, the string similarity join problem is to find all similar string pairs from the two collections. The similarity between two strings can be quantified by similarity metrics (see Section 2). String similarity join can play an important role in many real-world applications, e.g., data cleansing and integration, and duplicate detection. The brute-force algorithm that enumerates every string pair and checks whether the two strings in the pair are similar is rather expensive. To alleviate this problem, many algorithms have been proposed in the recent two decades [1–5,8,13,14,17–20,22–25,27,28].

One widely-adopted technique employs a filter-verification framework, which includes two steps: (1) Filter step: devising effective filtering algorithms to prune large numbers of dissimilar pairs and generating a set of candidate pairs; and (2) Verification step: verifying each candidate pair by computing the real similarity and outputting the final results. Filtering algorithms in the first step play an important role in the framework. Most of existing filtering algorithms employ a signature-based technique, which generates signatures for each string such that if two strings are similar, their signatures must have overlaps. Thus the signature-based technique can prune string pairs that have no common signature.

Recently many filtering techniques have been proposed, e.g., count filtering [8,13,18], length filtering [8,14], position filtering [25,27], prefix filtering [4] and content filtering [25]. As prefix filtering is the most effective filtering technique, many algorithms have been proposed to optimize prefix filtering for different similarity metrics, e.g., `AllPair` [2], `PPJoin` [27], `EDJoin` [25], `QChunk` [17], `VChunk` [24], `AdaptJoin` [23]. There are also many other signature schemes, e.g., `PartEnum` [1], `PassJoin` [14], `FastSS` [20]. In addition, there are some algorithms which directly compute join results using tree-based index structures, e.g., `TrieJoin` [22].

However these algorithms have not been thoroughly compared under the same experimental framework. For example, some algorithms [20,22] are tested only on specific datasets, e.g., datasets with short strings. Some algorithms [14, 17] are effective for some specific similarity metrics while expensive for other similarity metrics. Some fast algorithms, e.g., `PassJoin`, `FastSS`, and `QChunk`, are never compared. This makes it rather difficult for practitioners to decide which algorithms should be used for various scenarios.

To address this problem, in this paper we thoroughly compare existing similarity join algorithms on the same experimental framework. We make the following contributions. (1) We provide a comprehensive survey on a wide spectrum of existing string similarity join algorithms and classify them into different categories based on their main techniques. Figure 1 summarizes these algorithms. (2) We compare existing algorithms through extensive experiments on a variety of real-world datasets with different characteristics. (3) We report comprehensive findings obtained from the experiments and provide new insights about the strengths and weaknesses of existing algorithms which can guide practitioners to select appropriate algorithms for various scenarios.

2. PRELIMINARIES

Given two collections of strings, the *string similarity join* problem is to find all the *similar* pairs from the two collections. The criterion used to judge whether two strings are similar is called a *similarity metric*. We can broadly classify existing similarity metrics into two categories: *character-based similarity metrics* and *token-based similarity metrics*.

Proceedings of the VLDB Endowment, Vol. 7, No. 8

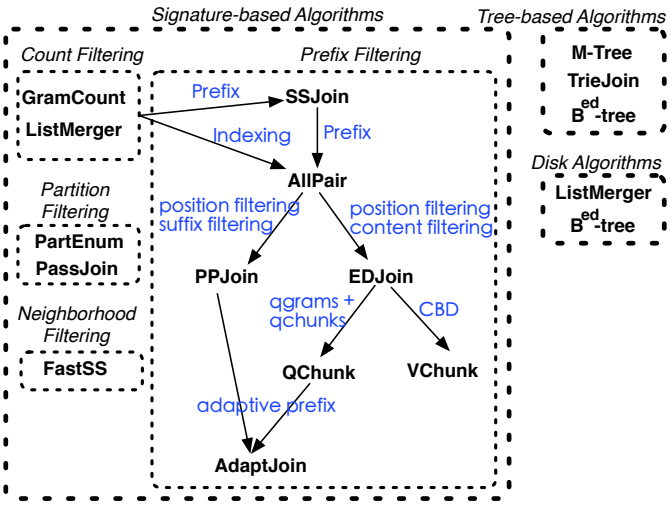


Figure 1: Overview of Similarity Join Algorithms.

Character-based Metrics. These metrics quantify the similarity between two strings based on character transformations. They are fit for capturing typographical errors. One representative character-based metric is edit distance. The edit distance between two strings is the minimum number of edit operations required to transform one string to the other, where the allowed edit operations include insertion, deletion, and substitution. For example, consider two strings "vldb" and "pvldb". Their edit distance ED("vldb", "pvldb") = 1, since the first one can be transformed to the second one by inserting a character "p". Two strings are similar w.r.t. the edit distance metric if their edit distance is not larger than a given threshold τ.

Token-based Metrics. These metrics first transform strings into sets of tokens and then use the set-based similarity metrics to quantify their similarity. The token-based metrics are suitable for long strings, e.g., documents. Two strategies are widely used to transform strings into sets: (1) tokenization and (2) q-grams. The former one tokenizes strings based on special characters, e.g., white-space characters. The latter one uses a string's substrings with length q to generate the set, where the substring with length q is called a q-gram. For example, the 2-gram set of "pvldb" is {"pv", "vl", "ld", "db"}. For simplicity, each element in the set (token or gram) is called a token and we also use string s to denote its corresponding token set, if there is no ambiguity. The well-known token-based metrics include OVERLAP, JACCARD, COSINE, and DICE, defined as below.

$$\text{OVERLAP:} \quad \text{OLP}(r,s) = |r \cap s|;$$

$$\text{JACCARD:} \quad \text{JAC}(r,s) = \frac{|r \cap s|}{|r \cup s|};$$

$$\text{COSINE:} \quad \text{COS}(r,s) = \frac{|r \cap s|}{\sqrt{|r| * |s|}};$$

$$\text{DICE:} \quad \text{DICE}(r,s) = \frac{2|r \cap s|}{|r| + |s|}.$$

where r and s are two string sets.

Two strings are similar w.r.t. the token-based similarity metrics if their similarity is not smaller than a threshold δ.

Problem Formulation. Based on these similarity metrics, we formulate the string similarity join problem.

DEFINITION 1 (STRING SIMILARITY JOIN). *Given two sets of strings R, S and a similarity metric, the string similarity join problem is to find the set $\{(r,s) \mid r \in R, s \in S, r$ and s are similar w.r.t. the similarity metric$\}$.*

For example, consider the strings in Table 1. Suppose we use JACCARD and the threshold is 0.4. We tokenize the string to generate a set using white spaces. The JACCARD similarity of s_3 and s_4 is 0.5, thus (s_3, s_4) is a similar pair. Here the similarity-join result is $\{(s_1, s_5), (s_3, s_4)\}$.

Table 1: An Example String Set.

ID	String
s_1	database concepts system
s_2	database large-scale techniques
s_3	programming concepts oracle
s_4	programming techniques oracle
s_5	database system illustration

Table 2: Algorithms and Supported Metrics (TOKEN={OLP, JAC, COS, DICE}, ALL={ED}∪TOKEN).

Techniques		Algorithms	Parameters	Metrics
Signature	Count	GramCount	q (for ED)	ALL
		ListMerger	q (for ED)	ALL
	Prefix	AllPair	q (for ED)	ALL
		PPJoin	N/A	TOKEN
		EDJoin	q	ED
		QChunk	q	ED
		AdaptJoin	q (for ED)	ALL
		VChunk	rules	ED
	Partition	PartEnum	N1, N2	ALL
		PassJoin	N/A	ALL
	Neighbor	FastSS	N/A	ED
Tree	Tree	M-Tree	N/A	ED
	Trie	TrieJoin	N/A	ED
	B-Tree	B^ed-tree	q	ED

Figure 1 classifies existing similarity join algorithms into different categories. Algorithms in the same rectangle belong to the same category. Each edge $A \xrightarrow{C} B$ denotes that algorithm B improves algorithm A by introducing new technique C. More details of these algorithms will be discussed later. Table 2 illustrates their supported similarity metrics.

3. FILTER-VERIFICATION FRAMEWORK

In this section, we introduce the filter-verification framework. We first discuss the filter step (Section 3.1) and then introduce the verification step (Section 3.2).

3.1 Filter Step

The filter step aims to devise effective filtering algorithms to prune dissimilar pairs. Since the filtering algorithm itself has overhead, it is a tradeoff between filtering power and filtering cost. In other words, the filtering algorithm should be "light-weight" while pruning large numbers of dissimilar pairs. In addition, for exact similarity joins, the filtering algorithms should not involve any false negative. To meet these requirements, one widely-used technique in the filter step is the signature-based technique, which generates a set of signatures for each string such that if two strings are similar they must share at least one common signature (and prunes the dissimilar pairs that have no common signature). We will introduce various signature schemes in Section 4.

To facilitate utilizing signatures to do pruning, existing algorithms usually build an inverted index, where entries are signatures and each signature is associated with an inverted list of strings that contain the signature. Then for each string, to find its candidates that share common signatures with the string, these algorithms first generate the signatures of this string and then the strings in the inverted lists of these signatures will be the candidates of this string.

Self-Join. We first discuss the algorithm for the self-join case, i.e., $R = S$. Algorithm 1 illustrates the pseudo code of the filter-verification algorithm. It iteratively builds the inverted index. First it initializes an empty index (line 2).

Algorithm 1: Signature-based Framework (Self-Join)

```
Input: String set S
Output: The similarity join result set A
1  A ← Φ;
2  Initialize an empty index I; // a map from a
       signature to a list of string IDs
3  for each s ∈ S do
4      Generate the signature set S_s of string s;
5      for each sig ∈ S_s do
6          for each c ∈ I[sig] do
7              verify(c, s);
8              if c is similar to s then A = A ∪ {(c, s)};
9          append s to I[sig];
10 return A;
```

Then for each string s, it generates s's signature set $\mathcal{S}_s$ (line 4). For each signature $sig \in \mathcal{S}_s$, it retrieves the inverted list $\mathcal{I}[sig]$. Each string c in the list is a candidate of s and it verifies candidate pair (c, s) (line 7). If c and s are similar, it adds the pair into the result set (line 8). Notice that it requires to append s into the list $\mathcal{I}[sig]$ (line 9).

R-S Join. The above techniques can be easily extended to support the case of $R \neq S$. It first builds the inverted index for strings in one set, e.g., R, and then scans strings in the other set S and uses the index on R to compute the results.

For example, consider the strings in Table 1. Suppose we use JACCARD and the threshold δ is 0.4. According to the definition of JACCARD, if the similarity between two strings is larger than 0, they should share at least one common token. Thus we can take the token set as the signature set. For example, the signature set of s_1 is {"database", "concepts", "system"}. Using this signature scheme, it generates the following candidate pairs: $(s_2, s_1), (s_4, s_2), (s_4, s_3), (s_5, s_1), (s_5, s_2)$. Obviously, the number of candidate pairs is reduced from 10 to 5. In the verification step, it verifies each candidate pair by computing their real similarity. For example, we will eliminate pair (s_2, s_1) since their similarity (0.2) is smaller than the threshold (0.4). Similarly, we can prune (s_4, s_2) and (s_5, s_2). Here we get two results: (s_4, s_3) and (s_5, s_1).

3.2 Verification Step

The verification step aims to verify each candidate pair by computing the real similarity.

Token-based Metrics. The token-based similarity metrics rely on the overlap of two string sets and string set sizes to compute the similarity. It is easy to get the string set sizes, and we discuss how to compute the overlap. It first sorts the tokens in the set (e.g., by alphabetical order) and then uses a merge-based algorithm to compute the overlap.

Character-based Metrics. The dynamic programming algorithm is used to compute the edit distance. Given two strings r and s, a matrix M with $|r|+1$ rows and $|s|+1$ columns is used to compute their edit distance, where $|r|(|s|)$ is the length of $r(s)$. $M[i][j]$ is the edit distance between the prefix of r with length i and the prefix of s with length j. Initially $M[i][0] = i$ and $M[0][j] = j$ for $0 \leq i \leq |r|$ and $0 \leq j \leq |s|$. Then each value of the matrix can be computed using the following equation,

$$M[i][j] = \min(M[i-1][j]+1, M[i][j-1]+1, M[i-1][j-1]+\gamma),$$

where $\gamma = 0$ if the i-th character of r is equal to the j-th character of s; $\gamma = 1$ otherwise. Obviously, the complexity of this algorithm is $\mathcal{O}(|r||s|)$.

To avoid unnecessary computations, given an edit-distance threshold τ, only values $M[i][j]$ for $|i-j| \leq \tau$ are required to compute since $M[i][j] > \tau$ for $|i-j| > \tau$. Thus in each row (or column), only $2\tau+1$ values are required to compute and the complexity can be improved to $\mathcal{O}((2\tau+1)\cdot\min(|r|,|s|)) = \mathcal{O}(\tau \cdot \min(|r|,|s|))$. Moreover, the early termination techniques [14] can be used to further improve the performance.

4. SIGNATURE-BASED ALGORITHMS

In this section, we discuss signature-based filtering algorithms. We first introduce the count filtering (Section 4.1) and length filtering (Section 4.2). Then we discuss the prefix filtering (Section 4.3). Finally we introduce other signature-based algorithms (Section 4.4).

4.1 Count Filtering

Count filtering is proposed in `GramCount` [8]. The basic idea is that if two strings are similar, their signatures must share at least T common signatures. In other words, if the number of shared signatures between two strings is smaller than T, the string pair can be pruned. `GramCount` utilizes this property to support edit distance. `ListMerger` extends `GramCount` to support token-based metrics in [13,18].

Computing T. `GramCount` generates q-grams for each string and takes q-grams as signatures. Suppose edit-distance threshold is τ. As one edit operation destroys at most q grams and string s has $|s|+1-q$ grams, if strings r and s are similar, their signature sets share at least $T = \max(|r|,|s|)+1-q-q\tau$ signatures. `ListMerger` takes each token as a signature. Suppose the overlap threshold is δ. Two strings are similar w.r.t. the overlap similarity, they must share at least δ common signatures, and $T = \delta$. Other token-based metrics can be converted to overlap. We take JACCARD as an example. If $\frac{|r\cap s|}{|r\cup s|} = \frac{|r\cap s|}{|r|+|s|-|r\cap s|} \geq \delta$, it is easy to deduce that $|r \cap s| \geq \lceil \delta|s| \rceil$ and $T = \lceil \delta|s| \rceil$. Similar techniques can be applied to COSINE and DICE. Thus if string r is similar to s, r shares at least T signatures with s, where

$$T = \begin{cases} |s|+1-q-q\tau & \text{ED} \\ \delta & \text{OLP} \\ \lceil \delta|s| \rceil & \text{JAC} \\ \lceil \delta^2|s| \rceil & \text{COS} \\ \lceil \frac{\delta}{2-\delta}|s| \rceil & \text{DICE} \end{cases} \quad (1)$$

Pruning with T. `ListMerger` builds inverted indexes for signatures. For each string s, `ListMerger` generates its signatures and retrieves inverted lists of its signatures. For each string r on these lists, `ListMerger` computes its count number, which is the number of lists that contain r. Obviously if the count number of string r is small than T, it can prune the pair (r, s). This is called the T-occurrence problem. Five efficient algorithms have been proposed to solve this problem by merging inverted lists [13,18] and the two heap-based algorithms MergeOpt [18] and DivideSkip [13] achieved better performance.

4.2 Length Filtering

Length filtering is proposed in `GramCount` [8]. The basic idea is that if two strings are similar, their length difference

cannot be large. Formally, if a string r is similar to s, then

$$\begin{aligned} \text{ED}: &\quad |s| - \tau \le |r| \le |s| + \tau \\ \text{Jac}: &\quad \delta|s| \le |r| \le \frac{|s|}{\delta} \\ \text{Cos}: &\quad \delta^2|s| \le |r| \le \frac{|s|}{\delta^2} \\ \text{Dice}: &\quad \frac{\delta}{2-\delta}|s| \le |r| \le \frac{2-\delta}{\delta}|s| \end{aligned} \tag{2}$$

To utilize the length filtering, one can partition the strings into different groups $G_{l_1}, G_{l_2}, \cdots, G_{l_n}$ and strings in each group have the same length. The string pairs between two groups G_{l_i} and G_{l_j} can be pruned, if the string lengths of the two groups do not satisfy Equation 2.

4.3 Prefix Filtering

4.3.1 Prefix Filtering

The prefix filtering is proposed in `SSJoin` [4] which focuses on enabling similarity joins inside DBMS. `AllPair` [2] extends the idea and proposes a prefix filtering based framework. We first use the Overlap similarity to illustrate how prefix filtering works and then apply it to other metrics.

Overlap. Assume we have a global ordering on all tokens in the string sets, e.g., the alphabetical order or inverse document frequency (idf) order. For each string, the prefix filtering sorts its tokens based on the given token ordering. Given the overlap threshold δ, for each string s, the prefix filtering selects the first $|s| - \delta + 1$ tokens as the signatures, denoted by s_p. It is easy to prove that if two strings r and s are similar, then $s_p \cap r_p \neq \phi$. For example, consider the first two strings "database concepts system" and "database large-scale techniques" in Table 1. Suppose the overlap threshold is 2. The prefix size is 3-2+1=2. Assume the tokens are sorted by idf in the descending order. The prefix filtering selects "concepts system" and "large-scale techniques" as their prefixes. As their prefixes have no overlap, the pair can be pruned. Existing methods use idf in descending order and take tokens with large idf (infrequent tokens) as signatures.

Edit Distance. The gram-based method [8] is used to support the edit distance. It generates the q-gram set for each string, sorts the grams based on a global ordering, and selects the first $q\tau + 1$ grams as its signatures. It is easy to prove that if r and s are similar, their prefixes must have common signatures. This is because each edit operation can only destroy at most q q-grams, and if more than $q\tau + 1$ grams are destroyed, it requires at least $\tau+1$ edit operations. For example, consider strings "`sigmod`" and "`sigir`" with edit-distance threshold $\tau = 1$. It sorts the q-grams by idf in descending order and generates their prefixes (underlined grams): {gm, mo, od, ig, si} and {gi, ir, ig, si}. Since they do not share any common signature, this pair will be pruned.

Based on Equation 1, for string s, the prefix filtering sorts its tokens and selects its first p tokens as signatures, where

$$p = \begin{cases} q\tau + 1 & \text{ED} \\ |s| - \delta + 1 & \text{Olp} \\ \lfloor (1-\delta)|s| \rfloor + 1 & \text{Jac} \\ \lfloor (1-\delta^2)|s| \rfloor + 1 & \text{Cos} \\ \lfloor (1 - \frac{\delta}{2-\delta})|s| \rfloor + 1 & \text{Dice} \end{cases} \tag{3}$$

4.3.2 Optimizations on Edit Distance

`EDJoin` [25]. The `EDJoin` algorithm proposes two optimization techniques to improve prefix filtering for edit distance.

(1) Position Filtering. `EDJoin` removes unnecessary signatures from the prefix to further reduce the prefix length. For example, consider string "`sigmod`" with $q = 2$ and $\tau = 1$. Suppose the signatures are "`si`", "`gm`" and "`od`" in the prefix filtering. `EDJoin` proves that it can remove the last signature "`od`" safely since it requires at least $\tau + 1$ edit operations to destroy both "`si`" and "`gm`". `EDJoin` devises effective algorithms to detect and remove unnecessary signatures.

(2) Content Filtering. The position filtering prunes dissimilar pairs using grams' positions. It is effective for mismatch grams scattered in different positions in the string but ineffective for consecutive grams. To alleviate this problem, `EDJoin` proposes the content filtering to do further pruning. For example, consider the following two strings: `sigmod` and `sigkdd`. Consider the underlined substrings. Suppose the edit-distance threshold is $\tau = 1$. As the two substrings is rather different, `EDJoin` can prune this pair. To achieve this goal, `EDJoin` uses the L_1 distance as a bound of edit distance to do pruning, where L_1 distance between two strings is the total number of different characters. `EDJoin` proves that the edit distance cannot be smaller than half of the L_1 distance. For example, the L_1 distance of the two substrings is 4 so the edit distance between the two strings is at least 2. Therefore, this pair can be pruned safely.

`QChunk` [17]. The `QChunk` algorithm contains two types of signatures: q-grams and q-chunks. The q-chunks are q-grams with starting positions at $i * q + 1$ for $0 \le i \le \frac{l-1}{q}$ where l is the string length. To guarantee the last q-chunk has exactly q characters, it appends several special characters, e.g., \$. For example, the q-chunks of "`vldbj`" are {"`vl`", "`db`", "`j$`"}. Let g_q denote the q-gram set and c_q denote the q-chunk set. Given two strings r and s, `QChunk` proves if they are similar,

(1) $|g_q(r) \cap c_q(s)| \ge \lceil \frac{|s|}{q} \rceil - \tau$, and

(2) $|c_q(r) \cap g_q(s)| \ge \lceil \frac{|r|}{q} \rceil - \tau$.

To use the two properties, `QChunk` devises two strategies to prune dissimilar pairs. The first one is to index q-grams of r and use q-chunks of s to generate candidates. The second one is to index q-chunks of r and use q-grams of s to generate candidates. Both of the two strategies can be applied into the prefix filtering framework. Moreover, `QChunk` devises effective techniques to decrease the number of signatures to the lower bound $\tau + 1$ which are similar to those in `EDJoin`.

4.3.3 Optimizations on Jaccard, Cosine, Dice

`PPJoin` [27]. The `PPJoin` algorithm proposes two optimization techniques for Jaccard, Cosine, and Dice.

(1) Position Filtering. It is similar to the first optimization of `EDJoin`, by using positions of signatures to prune dissimilar pairs. For example, consider the following two sets sorted by the alphabetical order: $\{\underline{B, C}, D, E, F\}$, $\{\underline{A, B}, C, D, F\}$. Suppose the Jaccard threshold is 0.8. The prefix filtering selects the first 5-0.8*5+1=2 tokens as the signatures (underlined). As the two prefixes have overlaps, the prefix filtering cannot prune the pair. However, from two prefixes, one can easily estimate the lower bound of the union size (3+3=6) and the upper bound of the intersection size (1+3=4). Thus the upper bound of the Jaccard similarity can be computed ($\frac{4}{6}$). Since the upper bound is smaller than the threshold, the position filtering can prune this pair. Similar ideas can be applied to Cosine and Dice.

(2) Suffix Filtering. It probes tokens in the suffix to estimate a tighter upper bound. For example, consider the following

two sets ("?" stands for unknown signatures):

$$\{\underline{A, B, D, E}, ?, ?, ?, ?, ?, ?, ?, Q, ?, ?, ?, ?, ?, ?, ?\}$$
$$\{\underline{A, C, D, E}, ?, ?, ?, ?, ?, Q, ?, ?, ?, ?, ?, ?, ?, ?, ?\}$$

This pair cannot be pruned by the first optimization technique since the upper bound is $\frac{17}{19} > 0.8$. It chooses the middle token in the suffix of the first set and uses it to do a binary search in the second set. By using the searching result position, it can estimate a tighter upper bound of the JACCARD similarity. In the example, the upper bound of the intersection size is $3+4+1+7 = 15$, the lower bound of the union size is $5+6+1+9 = 21$, and the upper bound of the JACCARD similarity is $\frac{15}{21} < 0.8$. Then this pair can be pruned by the suffix filtering. `PPJoin` iteratively estimates much tighter upper bounds [27].

4.3.4 The `AdaptJoin` Algorithm

The `AdaptJoin` algorithm aims to improve the prefix filtering fundamentally for all similarity metrics [23]. All the aforementioned algorithms select fixed-length prefix for strings with same length (different content), and they take the string pairs which have common signatures as candidate pairs. However, it is worth noting that if l more tokens in the set are selected into the prefix, called l-prefix scheme, two similar strings should share at least $l+1$ signatures. Thus the filtering power is enhanced while it involves more filtering cost. Thus there is a tradeoff and the `AdaptJoin` algorithm proposes an adaptive framework to choose the best prefix scheme to achieve high performance.

4.4 Other Signatures

4.4.1 The `VChunk` Algorithm

The `VChunk` algorithm [24] is designed for the edit distance. The signatures of `VChunk` are variable-length chunks, called "vchunk". The key part of the signature generation is the *chunk boundary dictionary* (CBD) which is a set of rules used to split the strings into chunks. For example, "`vl`" can be a rule. For each string, if it contains a substring "`vl`", `VChunk` cuts the string just after the substring. For example, the string "`pvldb`" will be split into two chunks: "`pvl`" and "`db`" by this rule. `VChunk` proposes the *tail-restricted CBD* which is a subset of all the CBDs and proves that if using a tail-restricted CBD to split strings, any edit operation will destroy at most 2 chunks. Thus the gram-based technique can be applied to the chunks split by CBDs. Next we define the tail-restricted CBD. It divides the character alphabet Σ in the strings into two disjoint subsets: the prefix character set P and the suffix character set Q (i.e., $P \cup Q = \Sigma$ and $P \cap Q = \phi$). All the rules in the CBD can be described by regular expression $[P]^*[Q]$ which denotes strings with arbitrary numbers of characters from P and a single character from Q. For example, {"`vl`", "`v`"} is an invalid tail-restricted CBD and {"`vl`", "`d`"} is a valid tail-restricted CBD. If this CBD is used to split string "`pvldb`", it will get three chunks {"`pvl`", "`d`", "`b`"}. `VChunk` adopts similar techniques of `EDJoin` to do the similarity join using the CBD. The most important part of this algorithm is to select a proper method to generate the CBD. More details are referred to [24].

4.4.2 The `PassJoin` Algorithm

The `PassJoin` algorithm [14] employs a partition-based framework. We will first introduce how to support the edit distance and then extend the idea to other metrics. The basic idea is based on the pigeon-hole principle. Given an edit-distance threshold τ, for each string s, `PassJoin` splits the string into $\tau+1$ segments and if another string r is similar to the string s, r must contain a substring which is equal to one of the segments of string s. For example, consider the following two strings with threshold $\tau = 2$: $\{si|gm|od,\ pvldb\}$. As the second string does not contain any substring that matches a segment of the first one, their edit distance must be larger than 2.

Intuitively, one should enumerate all substrings to check the condition, and this process is obviously rather expensive. To alleviate this problem, `PassJoin` proposes the *multi-match-aware selection* technique to select the minimum number of substrings. The number of selected substrings can be reduced to $\lfloor \frac{\tau^2-\Delta^2}{2} \rfloor + \tau + 1$, where $\Delta = ||r| - |s||$ is the length difference between two strings. For example, suppose $\tau = 3$ and $\Delta = 1$. `PassJoin` selects only 8 substrings.

For JACCARD, COSINE and DICE, `PassJoin` can convert them to EDIT DISTANCE. Take JACCARD as an example. First, `PassJoin` sorts all the tokens in the set based on a global ordering. Next it converts the JACCARD threshold to OVERLAP and then to EDIT DISTANCE. Notice that how to convert JACCARD to OVERLAP has been discussed in Section 4.1. Here we discuss how to convert OVERLAP to EDIT DISTANCE. Suppose the OVERLAP threshold is τ. Then the corresponding edit-distance threshold is at most $l_1 + l_2 - 2\tau$ where l_1 and l_2 are the numbers of tokens in the two sets respectively. This is because one string can be transformed to another string by $l_1 - \tau$ deletions and $l_2 - \tau$ insertions. Since l_2 is unknown when building the index, one can use the maximum possible length $\lfloor \frac{l_1}{\delta} \rfloor$ (see Equation 2) to replace l_2. Thus given a string with length l, the JACCARD threshold δ can be converted to the edit-distance threshold $\tau = \lfloor \frac{1-\delta}{\delta} l \rfloor$. For COSINE and DICE, the thresholds are respectively $\tau = \lfloor \frac{1-\delta^2}{\delta^2} l \rfloor$ and $\tau = \lfloor \frac{2(1-\delta)}{\delta} l \rfloor$.

4.4.3 The `PartEnum` Algorithm

The `PartEnum` algorithm [1] is designed for hamming distance which is the minimum number of substitutions required to change one string to the other. The signature of the `PartEnum` algorithm is a combination of "partition" and "enumeration". There are two parameters in the `PartEnum` algorithm: N1 and N2. Given a hamming distance threshold τ, in the first level, it partitions the string into N1 parts. According to the pigeon-hole principle, there should exist at least one partition and the corresponding partitions of the two strings have hamming distance no larger than $\tau_2 = \lfloor \frac{\tau}{\mathsf{N1}} \rfloor$. In the second level, it partitions each part into N2 partitions and generates the enumeration signatures: all possible combinations of selecting $\mathsf{N2} - \tau_2$ parts from N2 parts (i.e., $\binom{\mathsf{N2}}{\mathsf{N2}-\tau_2}$ combinations). Obviously if two strings are similar, they must share a common signature.

Next we introduce how to convert the JACCARD threshold δ to the hamming distance threshold. Suppose all strings are with the same length l. If the JACCARD similarity of two strings is not smaller than δ, then their OVERLAP should be not smaller than $\frac{2\delta l}{1+\delta}$. Thus the hamming distance between their vectors should be no larger than $2l - 2 * \frac{2\delta l}{1+\delta} = \frac{2(1-\delta)l}{1+\delta}$. Similarly, both thresholds of COSINE and DICE can be converted to $\frac{2l}{1-\delta}$. To deal with strings with different lengths, it splits all the strings into groups based on string lengths,

applies the techniques to strings in the same group, and uses the length filtering to do pruning between different groups. Finally, we discuss how to convert the ED threshold τ to the hamming distance threshold. It also uses the q-gram based method. Since an edit operation can at most destroy q q-grams, the ED threshold τ can be converted to the hamming distance threshold $2q\tau$.

4.4.4 *The* `FastSS` *Algorithm*

The `FastSS` algorithm [20] is designed only for the edit distance, which uses a neighborhood-based method. The basic idea is that if two strings are similar, their neighbors must have overlap. Thus it takes neighbors as signatures. Formally, suppose the edit-distance threshold is τ. Given a string s, let $D_i(s)$ denote the set of s' substrings by deleting i characters. Obviously, $|D_i(s)| = \binom{|s|}{i}$. For example, D_2("`pvldb`") = {`ldb, vdb, vlb, vld, pdb, plb, pld, pub, pvd, pvl`}. Let $\bar{D}_\tau(s) = \cup_{i=0}^{\tau} D_i(s)$. It can be proved that if two strings r and s are similar within threshold τ, then $\bar{D}_\tau(r) \cap \bar{D}_\tau(s) \neq \phi$. Based on this property, for each string s, `FastSS` takes the substrings (called neighborhoods) in $\bar{D}_\tau(s)$ as the signatures of s. Then it can use the filter-verification framework to compute join results. `FastSS` also devises effective verification algorithms to verify the candidate pairs based on the deleted characters. Obviously `FastSS` is ineffective for long strings since it involves larger numbers of signatures.

5. OTHER ALGORITHMS

5.1 The `TrieJoin` Algorithm

The `TrieJoin` algorithm [22] uses a trie structure to calculate the similarity join result directly. Each trie node is associated with a character and the path from the root to a leaf node corresponds to a string. Two strings with a common prefix will share a common ancestor.

`TrieJoin` relies on an important concept "*active nodes*". A node is called an active node for string s if the edit distance between s and the string w.r.t. the node is not larger than a given threshold τ. For self join, `TrieJoin` first builds a single trie structure and computes all the active nodes of the leaf nodes. Then given a leaf node, the strings corresponding to active nodes must be similar to the string corresponding to the leaf node. Thus `TrieJoin` can easily obtain the join result based on active nodes of leaf nodes. `TrieJoin` devises three efficient algorithms to compute the active nodes. The main idea is to traverse the trie index in pre-order and use the active nodes of the parent node to compute those of the child nodes. In other words, `TrieJoin` can share the computations among sibling nodes when computing their active nodes. `TrieJoin` also proposes a partition-based method to improve the algorithm. It partitions strings into two parts, and thus can decrease the threshold to $\frac{\tau}{2}$. Then it builds two tries for the two parts and utilizes the two tries to find join results. More technical details can be found in [22].

5.2 The `M-Tree` Algorithm

The `M-Tree` algorithm [5] is devised to support metric space similarity functions, which satisfy: (1) $d(x, y) \geq 0$; (2) $d(x, y) = 0 \Leftrightarrow x = y$; (3) $d(x, y) = d(y, x)$; and (4) $d(x, z) \leq d(x, y) + d(y, z)$, where d is a similarity function. We can see that `M-Tree` can be used for Edit Distance since it is a metric. The basic idea of `M-Tree` is to use the triangle inequality to prune dissimilar pairs. To this end, it builds a tree structure, called `M-Tree`, where each leaf node contains a group of similar strings and each internal node is used for pruning. An internal node N in an `M-Tree` stores a landmark string N_m and a radius N_r, which denotes that all the strings under the node have distance to N_m no larger than N_r and vice versa. Based on this property, for each string s, it traverses the `M-Tree` from the root node. Consider a node N. If $d(N_m, s) > N_r + \tau$, the node can be pruned; otherwise, it accesses each of its children. If the node is a leaf, it visits each string in the node and computes its similarity to the string. It is important to construct an effective `M-Tree` with high pruning power and more details are referred to [5].

5.3 Other Related Works

Similarity Joins inside DBMS. Gravano et. al. [8] studied how to enable similarity joins inside DBMS using database functionalities. `SSJoin` [4] proposed prefix filtering and incorporated the techniques into DBMS.

Parallel Similarity Joins. Vernica et. al. [21] studied parallel similarity joins using MapReduce by extending prefix filtering based techniques. Metwally et. al. [16] studied the parallel set similarity join problem using count filtering.

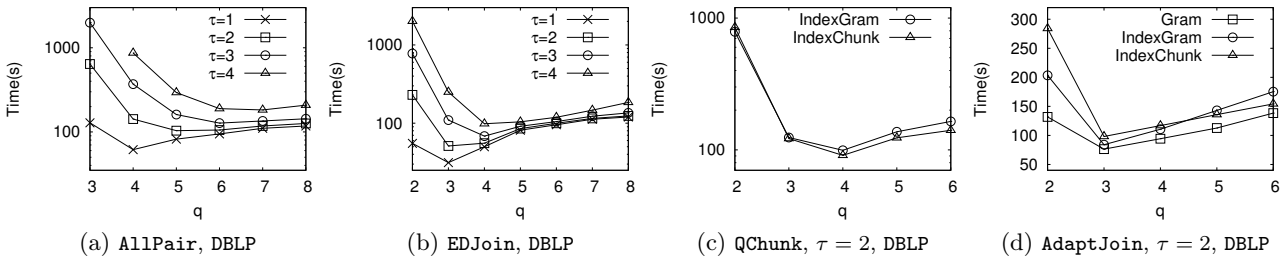
Top-k Similarity Joins. Given two sets of strings, the top-k similarity join problem is to find top-k similar pairs with the largest similarity. Xiao et. al. [26] extended prefix filleting to support top-k similarity join. Zhang et. al. [28] extended B-tree to support top-k similarity search problem. Lee et. al. [11,12] studied the similarity join size estimation problem which can also be utilized to compute top-k similar pairs. Kim et. al. [10] studied parallel top-k string similarity join problem using MapReduce.

Jestes et. al. [9] and Lian et. al. [15] studied the problem of similarity joins on probabilistic data. Satuluri et. al. [19] studied the problem of approximate similarity joins (which may miss results) and proposed a Bayesian algorithm by extending traditional locality-sensitive hashing.

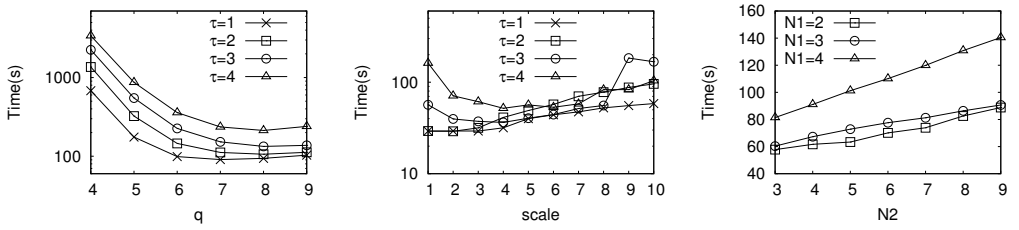
Similarity Search. Similarity search has been extensively studied [6,7,13,18], which finds the similar strings of the query string from a set of data strings. Similarity joins based techniques can be extended to support similarity search [17, 23]. Disk-based similarity search was studied in [3,28]. Different from similarity joins, the indexing time can be excluded in similarity search, and thus in similarity joins we need to consider the indexing cost which cannot be expensive but in similarity search we can build sophisticated indexes.

Recently EDBT/ICDT 2013 organized a competition on string similarity search and join[1]. The competition used two datasets: the city-name dataset with short strings and the `Genome` dataset with long strings. It allowed participants to use parallel algorithms. For similarity search, `PassJoin` won the champion on the city-name dataset and the partition-based method with sophisticated graph indexes won the champion on the `Genome` dataset. For similarity join, `PassJoin` won the champion on both of the two datasets. Our work had the following differences from the competition. First, the competition only considered the edit-distance functions and our work discussed both edit-distance functions and token-based functions. Second, in the competition, different authors implemented different algorithms while we implemented all the algorithms by ourselves. Third, we evaluated on more datasets, varied the dataset distributions, and compared disk-based algorithms.

[1] www2.informatik.hu-berlin.de/~wandelt/searchjoincompetition2013/



(a) AllPair, DBLP (b) EDJoin, DBLP (c) QChunk, $\tau = 2$, DBLP (d) AdaptJoin, $\tau = 2$, DBLP



(e) ListMerger, DBLP (f) VChunk, DBLP (g) PartEnum, $\tau = 2$, DBLP-50K

Figure 2: Evaluation on Parameter Selection (ED).

6. EXPERIMENTS

We conducted extensive experiments to compare existing string similarity join algorithms. Our experimental goal was to evaluate running time, candidate size, memory usage and scalability of different algorithms. The running time was the total time to compute the join results, including sorting tokens, generating signatures, indexing signatures, filtering and verification (if necessary). The candidate size was the number of candidates generated in the filtering step. The memory usage was the peak memory used during the algorithm execution. We compared the following algorithms.

Character-based Metrics. ListMerger [13,18], AllPair [2], EDJoin [25], QChunk [17], AdaptJoin [23], VChunk [24], PassJoin [14], PartEnum [1], FastSS [20], TrieJoin [22], BiTrieJoin [22], M-Tree [5]. BiTrieJoin was the improved version of TrieJoin by using two tries as discussed in Section 5.1. We implemented the best algorithm DivideSkip [13] for ListMerger.

Token-based Metrics. ListMerger [13,18], AllPair [2], PPJoin [27], PPJoin+ [27], AdaptJoin [23], PassJoin [14], PartEnum [1]. As discussed in Section 4.3.3, we used PPJoin to denote the algorithm which only contained the first optimization (position filtering) and PPJoin+ to denote the algorithm which used two optimizations (position & suffix filtering).

All the algorithms were implemented by ourselves using C++, compiled by GCC 4.8.2 with -O3 flag. All the experiments were conducted on a Ubuntu server with two Intel Xeon E5420 CPUs (8 cores, 2.5GHz) and 32GB memory.

Datasets. We used a variety of datasets with different characteristics which were also widely used in previous works. Tables 3-4 showed the details of the datasets. For Edit Distance we used the following datasets.

Short Strings: Word [22] was a set of real English words and its average string length was 8.7. Querylog [14,22,23] was a set of query log strings[2] and the average length was 18.9. These datasets can be used in data cleansing applications.

Middle-Length Strings: DBLP [1,2,14,17,23–25,27] included publication titles[3] and the average string length was 70.3. It can be used in duplicate detection applications.

Long Strings: Genome was a dataset of human gene sequences[4] and the average length was 100. It can be used in data integration applications.

Table 3: Datasets for ED.

Dataset	Cardinality	Avg Len	Max Len	Size	Type
Word	122,823	8.7	29	1.2M	Short
Querylog	500,000	18.9	500	9.7M	Short
DBLP	1,000,000	70.3	766	69M	Middle
Genome	250,000	100	100	25M	Long

Table 4: Datasets for Jac, Cos and Dice.

Dataset	Cardinality	Avg Tok	Max Tok	Size	Type
Trec	347,949	75	273	103M	Small
Enron	245,567	135	3162	129M	Middle
Wiki	4,000,000	213	36907	3.2G	Large

For Jaccard, Cosine and Dice, we used the following datasets. We used white spaces to tokenize each string.

Small Token Number: Trec [2,17,24,25,27] was a set of documents from the well-known benchmark in information retrieval[5] and the average token number was 75. It can be used in data cleansing applications.

Middle Token Number: Enron [2,23,27] included a set of emails with titles and bodies[6] and the average token number was 135. It can be used in data integration applications.

Large Token Number: Wiki [19] included a set of English wikipedia webpages[7] and the average token number was 213. It can be used in document clustering applications.

Global Ordering. Algorithms for token-based metrics required the tokens to be sorted in a global ordering. Some algorithms for Edit Distance needed to sort grams using a global ordering. We used the best order for all algorithms. For PartEnum and PassJoin, the best order is random order. For other algorithms, the best order is idf order.

6.1 Experiments on Edit Distance

Parameter Selection. Many existing algorithms required to tune parameters to achieve the best performance. The q-gram based algorithms, e.g., ListMerger, AllPair, EDJoin, QChunk and AdaptJoin, required to tune parameter q. VChunk required to tune parameter scale. PartEnum required to tune parameters N1 and N2. Figure 2 shows the results. As PartEnum was slow, we ran it on a small dataset DBLP-50K (by randomly selecting 50K string from DBLP) and other algorithms were run on the DBLP dataset.

[2] http://www.gregsadetsky.com/aol-data/
[3] http://dblp.uni-trier.de/xml/
[4] http://www.1000genomes.org/
[5] http://trec.nist.gov/data/t9_filtering.html
[6] http://www.cs.cmu.edu/~enron/
[7] http://dumps.wikimedia.org/

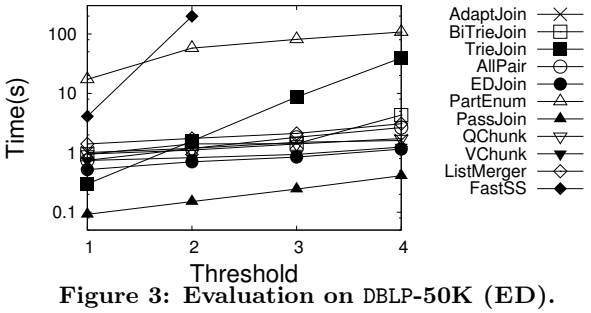


Figure 3: Evaluation on DBLP-50K (ED).

Table 5: Best parameter values for Figure 3.

Method	ED = 1	ED = 2	ED = 3	ED = 4
AdaptJoin	q =2	q =2	q =3	q =3
AllPair	q =3	q =4	q =5	q =5
EDJoin	q =2	q =3	q =3	q =3
PartEnum	N1/N2=3/6	N1/N2=3/3	N1/N2=2/4	N1/N2=2/5
QChunk	q =2	q =3	q =3	q =4
VChunk	scale =1	scale =2	scale =3	scale =3
ListMerger	q =6	q =7	q =8	q =7

We have the following observations. First, different parameter choices indeed had effect on the performance, thus it is very important to select an appropriate parameter. For example, consider EDJoin in Figure 2(b). The performance on $q = 2$ was much worse than that on $q = 3$. Second, for the gram-based methods, q should be neither too large nor too small, because a large q involved large numbers of signatures (the number of signatures is related to q, e.g., $q\tau + 1$) and a small q leaded to low pruning power (short grams had large probabilities to match). Thus we should carefully choose an appropriate parameter q. Third, QChunk had two strategies to build indexes: (1) indexing q-grams; and (2) indexing q-chunks, and AdaptJoin had three strategies, (1) traditional q-gram based method; (2) indexing q-grams; and (3) indexing q-chunks. We can see that it was rather hard to select the best strategy and the two methods required to tune different strategies to achieve the best performance, and this was coincided with the findings in the original papers.

In the following experiments, for each threshold τ, we tuned the parameters for every algorithm and reported the best performance each algorithm can achieve.

Small Dataset. As PartEnum and M-Tree were rather inefficient, we first compared all algorithms on the small dataset DBLP-50K. Figure 3 shows the results and Table 5 shows the best parameters. (Since there are many experiment figures, we cannot show the best parameter values for the following experiments due to the space constraints.) We also evaluated the M-Tree algorithm but it failed to report results within 2 hours. In the rest of the experiments, we do not show the result in the figure if the corresponding algorithm cannot return results within 2 hours.

We have the following observations. First, PartEnum were much worse than other algorithms because PartEnum was designed for hamming distance and had low pruning power for other similarity metrics. Thus we excluded it in the remainder of the experiments. Second, FastSS can only compute the results for small thresholds, e.g., 1 and 2. This is because it was designed for short strings, and for long strings it involved exponential numbers of signatures. We will compare it deeply on datasets with extremely short strings later. Third, TrieJoin was efficient for small thresholds but inefficient for large thresholds. This is because, for large thresholds, it involved large numbers of active nodes which was rather expensive to calculate. BiTrieJoin could alleviate such problems as it can decrease the threshold by half. Fourth, PassJoin achieved the best performance since it had large pruning power using the partition-based strategy which was effective for both long strings and short strings. Fifth, some q-gram based algorithms, e.g., AllPair, EDJoin, QChunk, achieved similar results. As these algorithms relied on q, their signatures were usually short and leaded to low pruning power.

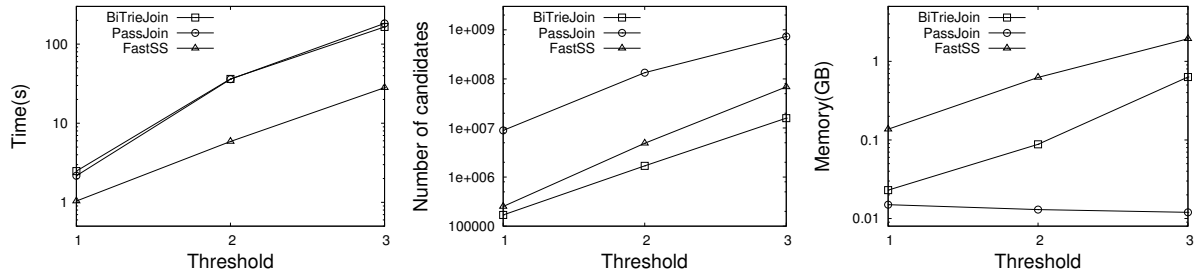
Datasets with Short Strings. Since FastSS only achieved high performance on datasets with short strings, we conducted an experiment on the Word dataset with average string length of 8.7. We compared with other two algorithms PassJoin and BiTrieJoin which were also efficient for short strings. Figure 4 shows the results. FastSS was faster than both PassJoin and BiTrieJoin. This is because the signatures of FastSS have much stronger pruning power. However FastSS had rather large space overhead to maintain the signatures as it generated exponential numbers of signatures. In terms of candidate sizes, as BiTrieJoin did not involve false positives, its candidate number was the minimum. FastSS had smaller numbers of candidates than PassJoin which implied it had larger pruning power.

Large Datasets. We evaluated the algorithms on large datasets. As many algorithms cannot return results in 2 hours on the Genome dataset, we used Genome-100K (by randomly selecting 100K strings from Genome). Figure 5 shows the results. We have the following observations. First, PassJoin was nearly the best among all the algorithms. On the Genome dataset with long strings, PassJoin was much faster than other algorithms, even by an order of magnitude. Similar results can be achieved on the DBLP dataset with middle length strings. On the datasets with short strings, PassJoin and BiTrieJoin achieved similar results and outperformed other algorithms. Second, similarity joins were hard for datasets with short strings using large thresholds, since there were many similar pairs and it was rather hard to select appropriate signatures to prune dissimilar pairs.

Scalability. We evaluated the scalability of different algorithms, including running time, candidate numbers, and memory usage. Figure 6 shows the result. We can see that BiTrieJoin and PassJoin outperformed other algorithms. This is consistent with results on the large datasets. Moreover, with the increase of dataset sizes, PassJoin nearly achieved the linear results, and this was also attributed to its effective pruning techniques. PassJoin involved less memory than other algorithms, because the number of signatures in the q-gram-based methods depended on the string length, while that of PassJoin relied on threshold τ. BiTrieJoin and TrieJoin had the smallest number of candidates as they directly computed the answers.

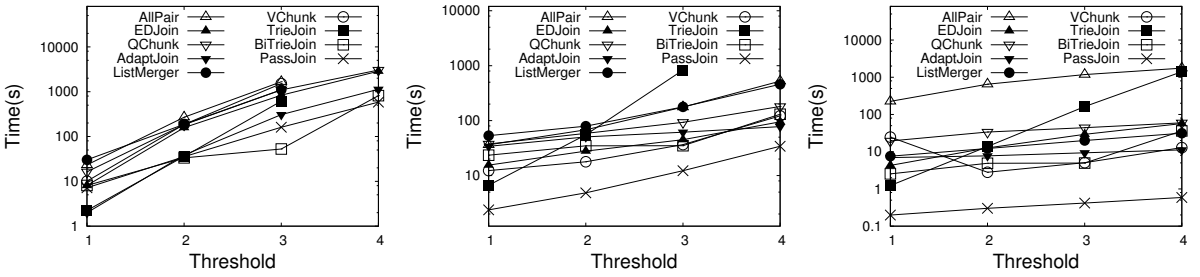
R-S Join. We evaluated the R-S join on the Word dataset. We randomly divided the original dataset into two sets with cardinalities ratio of 1:4. We compared FastSS, PassJoin, and BiTrieJoin. For each algorithm, we tested two strategies: indexing the smaller set or indexing the larger set. Figure 7 shows the results. We can see that these algorithms can efficiently support R-S join and the running time of the different choices of which part to index was similar.

Algorithm Selection. We report our findings from the results and discuss how to select appropriate algorithms (Table 6). On the datasets with short strings, FastSS $>$ PassJoin $\sim$ BiTrieJoin $>$ AdaptJoin $>$ TrieJoin $>$ QChunk $\sim$



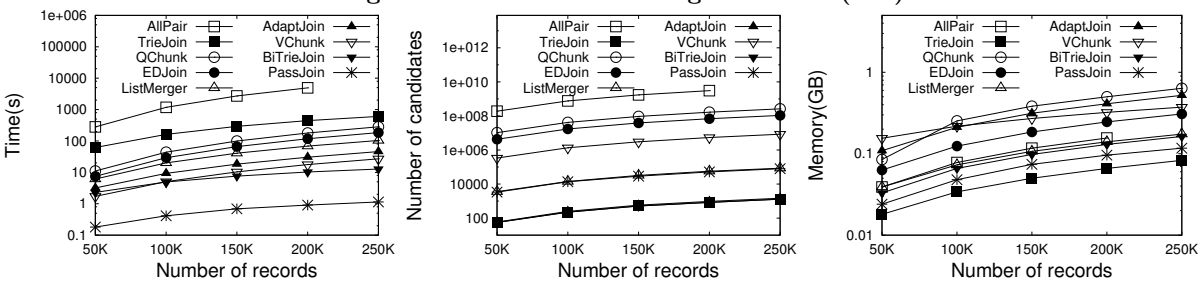
(a) Running Time (b) Candidate Number (c) Memory Usage

Figure 4: Evaluation on the `Word` Dataset with Very Short Strings (ED).



(a) `Querylog` (Short) (b) `DBLP` (Middle) (c) `Genome`-100K (Long)

Figure 5: Evaluation on Large Datasets (ED).



(a) Runing Time (b) Candidate Number (c) Memory Usage

Figure 6: Evaluation on Scalability on the `Genome` Dataset (ED, $\tau = 3$).

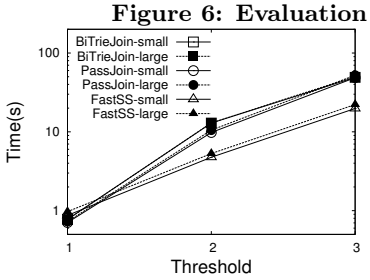


Figure 7: Evalution on R-S Join (`Word`, ED).

Table 6: Algorithm Selection for ED.

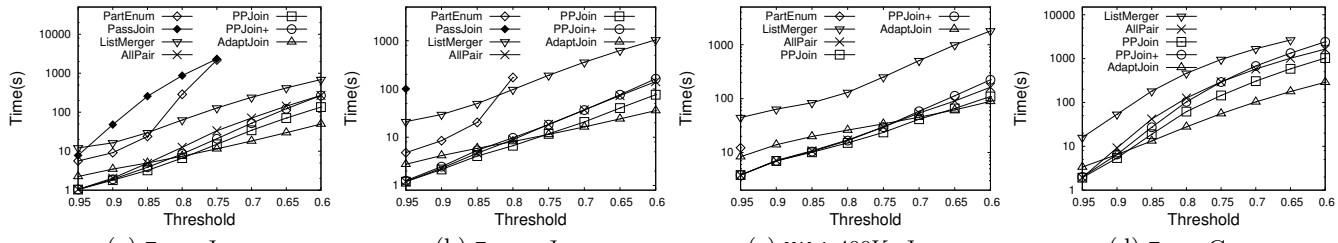
Datasets	Suggested Algorithms
Short Strings	`FastSS` & `PassJoin`
Middle-length Strings	`PassJoin`
Long Strings	`PassJoin`

`EDJoin` > `ListMerger` > `VChunk` > `AllPair` ≫ `PartEnum` ≫ `M-Tree`, where ∼ denotes similar performance, > denotes better, and ≫ denotes significantly better. On the datasets with long strings, `PassJoin` > `BiTrieJoin` > `EDJoin` > `AdaptJoin` > `QChunk` > `VChunk` > `ListMerger` ∼ `AllPair` > `TrieJoin` ≫ `PartEnum` ∼ `FastSS` ≫ `M-Tree`. On the datasets with middle-length strings, `PassJoin` > `VChunk` > `BiTrieJoin` > `TrieJoin` > `AdaptJoin` > `QChunk` > `EDJoin` > `AllPair` > `ListMerger` ≫ `PartEnum` ∼ `FastSS` ≫ `M-Tree`.

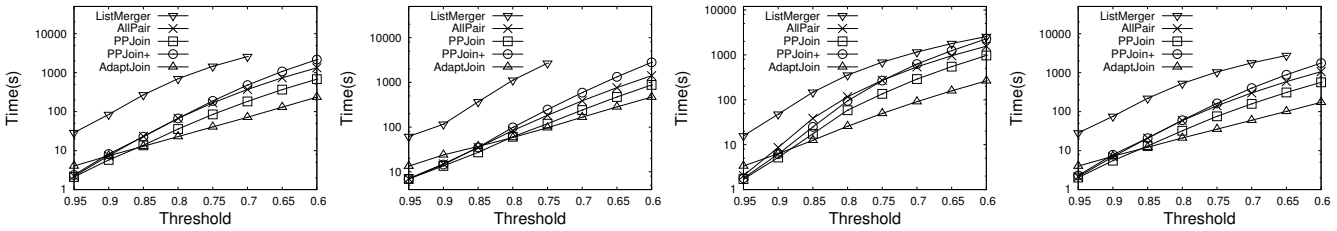
6.2 Experiments on Jaccard, Cosine and Dice

Among algorithms for Jaccard, Cosine and Dice, only `PartEnum` required parameters and we tuned its parameters as discussed in Section 6.1. Due to space constraint, we omitted the details to tune its parameters.

Large Datasets. We compared the algorithms on large datasets. As many algorithms cannot return results in 2 hours on the `Wiki` dataset, we used `Wiki`-400K (by randomly selecting 400K strings from `Wiki`). Figure 8 shows the results. Since `PassJoin` and `PartEnum` are very slow for Cosine and Dice, we only showed their results on Jaccard. We make the following observations. First, different token-based similarity metrics had no large effect on the performance of different algorithms. In other words, these algorithms nearly achieved the same results on different metrics. Second, `AllPair`, `PPJoin`, `PPJoin+` and `AdaptJoin` achieved similar results and outperformed `PartEnum` and `PassJoin`. Third, `PPJoin` and `PPJoin+` were better than `AllPair` because the two filtering techniques (position and suffix filtering) can prune more dissimilar pairs. For large thresholds (≥ 0.8), `PPJoin+` was better than `PPJoin` because the suffix was long and the suffix filtering can significantly prune dissimilar pairs. For small similarity thresholds (< 0.8), `PPJoin` was better than `PPJoin+`, because the suffix was short and the suffix filtering was more expensive than calculating overlap directly. Fourth, on large similarity thresholds (≥ 0.8), `PPJoin+` was the best. On small thresholds (< 0.8), `AdaptJoin` outperformed `AllPair`, `PPJoin`, `PPJoin+`. This is because for small thresholds, similarity join algorithms took longer time and `AdaptJoin` had opportunity to select adap-

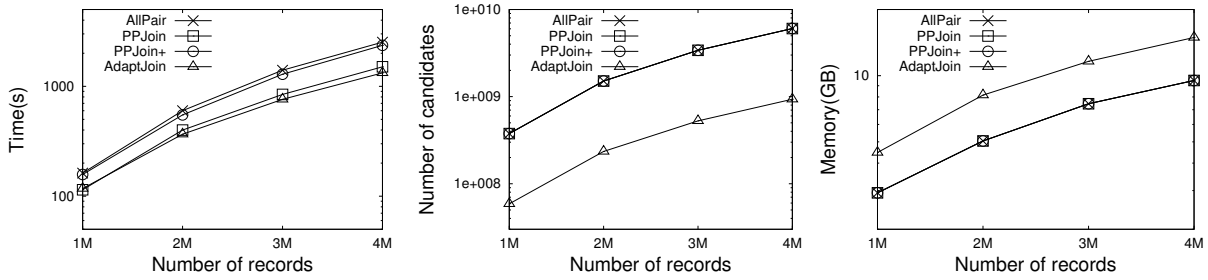


(a) Trec, Jac (b) Enron, Jac (c) Wiki-400K, Jac (d) Trec, Cos



(e) Enron, Cos (f) Wiki-400K, Cos (g) Trec, Dice (h) Enron, Dice

Figure 8: Evaluation on Large Datasets (Token-based Metrics).



(a) Runing Time (b) Candidate Number (c) Memory Usage

Figure 9: Evaluation on Scalability on the Wiki Dataset (Jac, $\delta = 0.75$).

tive prefix lengths to improve performance. Fifth, different from Edit Distance, different string lengths had no much effect on performance for token-based similarity metrics.

Scalability. We evaluated the scalability of different algorithms. Figure 9 shows the results on the large dataset Wiki. We only showed the results of Jaccard since the results of Cosine and Dice are similar. We can see that when the number of strings was large, AdaptJoin was always the fastest one. This is because for large datasets, similarity join algorithms took long time to find similar pairs and AdaptJoin had large potential to select variable-length prefixes to improve the performance. AdaptJoin involved slightly larger indexes than PPJoin+ as AdaptJoin required to build additional index structures.

R-S Join. We evaluated the R-S join on the Wiki dataset. We randomly selected 2 million strings as one set and other 2 million strings as another set. We compared AllPair, PPJoin, PPJoin+ and AdaptJoin. For each algorithm, we tested two strategies, indexing the small set or indexing the large set. Figure 10 shows the results. We can see that AdaptJoin still achieved the best performance. Moreover, AdaptJoin-small (indexing the small set) was slightly better than AdaptJoin-large (indexing the large set) because indexing small/large datasets will not affect the pruning power but indexing the large set will take longer time.

Algorithm Selection. We report our findings and discuss how to select an algorithm (Table 7). On large datasets, for large thresholds, AdaptJoin > PPJoin+ > PPJoin > AllPair ≫ ListMerger ≫ PassJoin > PartEnum; for small thresholds, AdaptJoin > PPJoin > PPJoin+ > AllPair ≫ ListMerger ≫ PassJoin > PartEnum. On small datasets, for large thresholds, PPJoin+ > PPJoin > AdaptJoin > AllPair ≫ ListMerger ≫ PassJoin > PartEnum; for small thresholds, AdaptJoin > PPJoin > PPJoin+ > AdaptJoin > AllPair ≫ ListMerger ≫ PassJoin > PartEnum.

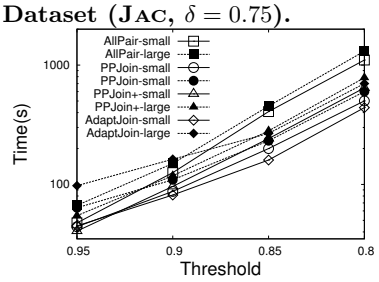


Figure 10: Evalution on R-S Join (Wiki, Jac).

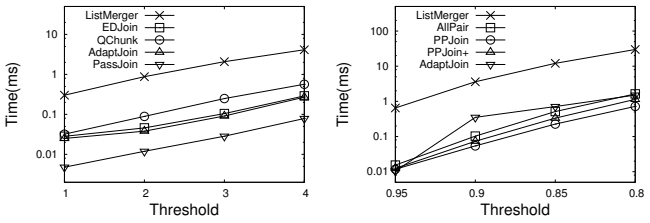
Table 7: Algorithm Selection for Jac, Cos, Dice.

Datasets	Thresholds	Suggested Algorithms
Large Token No.	Large/Small	AdaptJoin
Small Token No.	Large	PPJoin+ & PPJoin
Small Token No.	Small	AdaptJoin

6.3 Other Experiments

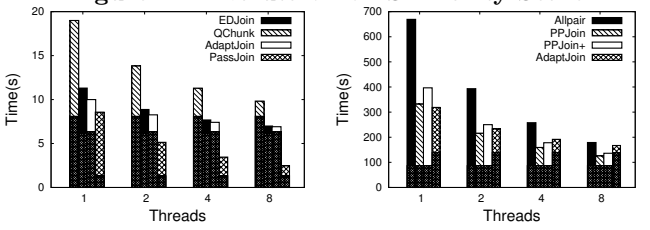
6.3.1 Similarity Search

We extended the similarity join algorithms to support similarity search and compared them with the similarity search algorithm ListMerger. We randomly selected 10K queries and reported the average search time. Figure 11 shows the results. Note that TrieJoin and BiTrieJoin relied on dual subtrie pruning to achieve high performance and they cannot utilize this feature for similarity search and thus they were inefficient for similarity search. In addition, AllPair was always slower than EDJoin. Thus we did



(a) ED, DBLP (b) JAC, Wiki

Figure 11: Evalutaion on Similarity Search.



(a) ED, DBLP (b) JAC, Wiki

Figure 12: Evalutaion on Parallel Algorithms

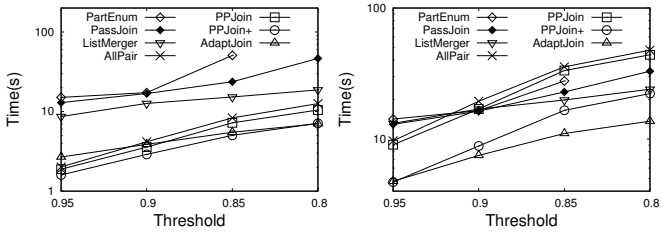
not show TrieJoin, BiTrieJoin and AllPair for EDIT DISTANCE. We can see that the extended algorithms were better than ListMerger as they shorten the signatures using effective signature-based techniques (e.g., prefix filtering) and accessed smaller numbers of signatures. On EDIT DISTANCE, the average search time was under 1 ms and PassJoin outperformed other methods; and on JACCARD, the average search time was also under 1 ms and PPJoin and PPJoin+ outperformed other algorithms. AdaptJoin was slower than PPJoin and PPJoin+ as AdaptJoin had additional overhead to select high-quality signatures.

6.3.2 Evaluation on Parallel Algorithms

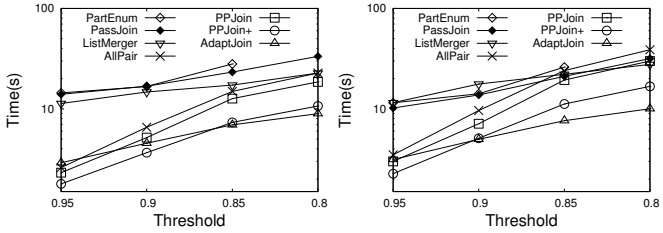
We parallelized existing similarity join algorithms and evaluated the parallel algorithms. We used R-S joins as an example. We evenly split each dataset into two parts. We first indexed strings in one dataset and then performed search on strings in another dataset in parallel. Figure 12 shows the results. We showed the running time of the index step and search step separately where the lower bar denoted the indexing time and the upper bar denoted the search time. We did not show ListMerger since it was too slow. We can see that the parallel algorithms can improve the performance. For example, PassJoin can improve the join time to 2 seconds on 8 nodes from 9 seconds on 1 node. We find that the speedup cannot reach the optimal value as we can parallelize the search step but cannot parallelize the indexing step. It is still challenging to devise efficient parallel algorithms.

6.3.3 Evalutaion on Dataset Distribution

We evaluated different dataset distributions. We generated four datasets with different distributions as follows. We first generated $N = 400K$ distinct tokens and then constructed each string by randomly selecting tokens. The probability to select the i-th token t_i is $p(t_i) = \frac{score(t_i)}{\sum_{1 \le j \le N} score(t_j)}$, where $score(t_i) = \frac{1}{i}$ for Zipfian distribution, $score(t_i) = \frac{1}{N}$ for uniform distribution, $score(t_i) = \int_{i-1}^{i} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ for normal distribution. We generated two datasets using normal distribution with $\mathcal{N}(0, (\frac{N}{3})^2)$ $(\mu = 0, \sigma = \frac{N}{3})$ and $\mathcal{N}(\frac{N}{2}, (\frac{N}{10})^2)(\mu = \frac{N}{2}, \sigma = \frac{N}{10})$. Each dataset contained 500K strings and each string contained 50 tokens. Figure 13 shows the results. We can see that in the traditional Zipfian distribution, PPJoin, PPJoin+, and AdaptJoin achieved the best performance. However, for some special distributions such as uniform distribution, the performance for PPJoin and



(a) Zipfian (b) Uniform



(c) Normal, $\mathcal{N}(0, (\frac{N}{3})^2)$ (d) Normal, $\mathcal{N}(\frac{N}{2}, (\frac{N}{10})^2)$

Figure 13: Evalutaion on Distributions (JAC).

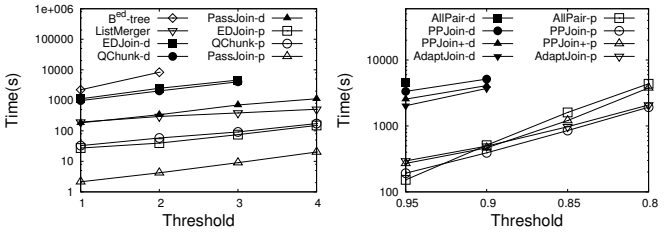
PPJoin+ were slightly worse. We got similar results for uniform distribution especially $\mathcal{N}(\frac{N}{2}, (\frac{N}{10})^2)$. The main reason is that the high pruning power of PPJoin and PPJoin+ relied on the low frequency of signatures in the prefix. In Zipfian distribution, the signatures in prefixes were tokens with largest idf and their frequencies were small. Thus the matching probability between different strings was low. However in other distributions, the matching probability can be larger than that in the Zipfian distribution. Instead, AdaptJoin can adaptively selected the high-quality prefixes and thus it still achieved higher performance in different distributions.

6.3.4 Evaluation on Disk-based Algorithms

We extended similarity join algorithms to support really large datasets and compared them with ListMerger [3] and B^{ed}-tree [28], which were specially designed for disk-based settings in order to support similarity search using disk-based indexes. We implemented two types of methods to support large datasets that cannot be loaded into the memory. The first built the disk-based indexes on strings in one dataset and performed search on strings in another dataset by utilizing the disk-based indexes (denoted by EDJoin-d, PPJoin-d). The second partitioned each dataset into several small groups and performed join algorithms on the small groups which can be loaded into the memory (denoted by EDJoin-p, PPJoin-p). For EDIT DISTANCE, we set the memory buffer as 64MB and for JACCARD, we set the buffer as 512MB, which were about 5% of memory used by in-memory algorithms EDJoin and PPJoin respectively. Figure 14 shows the results. Our extended algorithms using the first strategy outperformed B^{ed}-tree but were worse than ListMerger as B^{ed}-tree involved large numbers of random accesses and ListMerger avoided unnecessary disk accesses using a novel physical layout for inverted indexes. Our extended algorithms using the second strategy outperformed ListMerger, as we loaded the data into memory, used in-memory methods to compute results, and avoided randomly accessing disk-based indexes. Thus for similarity search the disk-based methods were better but for similarity joins the partition-based methods achieved higher performance.

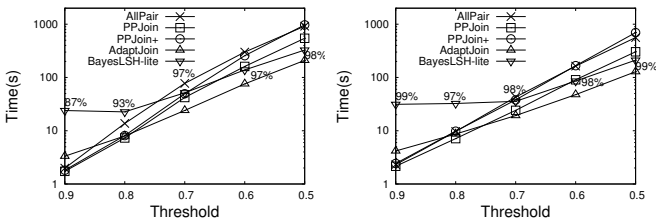
6.3.5 Comparision with Approxiamte Algorithms

We compared with approximate similarity join algorithms, BayesLSH-lite [19], which approximately computed the results and may miss results. We set the expected recall of



(a) ED, DBLP (b) Jac, Wiki

Figure 14: Evalutaion on Disk-based Algorithms.



(a) Jac, Trec (b) Jac, Enron

Figure 15: Comparision with Approxiamte Methods

`BayesLSH-lite` to 97%. Figure 15 shows the results. In the figure, we illustrated the actual recall of `BayesLSH-lite`. We can see that `BayesLSH-lite` indeed missed some results. For larger τ, `BayesLSH-lite` achieved worse performance than exact similarity join algorithms. There are two main reasons. First, exact similarity join algorithms can generate high-quality signatures for larger thresholds. Second, `BayesLSH-lite` was expensive to implement the LSH functions and took very long time to build the index. For very small thresholds (e.g., $\leq$0.4), `BayesLSH-lite` was better, because there were large numbers of candidates and `BayesLSH-lite` can effectively prune these candidates. Thus for large thresholds, we recommended exact similarity join algorithms; and for small thresholds (e.g., $\leq$0.4), we suggested approximate similarity join algorithms.

7. CONCLUSION

This paper provides a comprehensive survey on a wide spectrum of existing string similarity join algorithms, including, `ListMerger`, `PartEnum`, `AllPair`, `PPJoin`, `EDJoin`, `QChunk`, `VChunk`, `AdaptJoin`, `PassJoin`, `TrieJoin`, `FastSS`, and `M-Tree`, and compares them through extensive experiments on seven real-world datasets with different characteristics. We provide the following experimental findings.

(1) For Edit Distance, `PassJoin` is the best choice and it outperforms other algorithms in terms of both running time and memory usage. On datasets with very short strings, `FastSS` is an alternative choice as it achieves better performance than `PassJoin` but with higher memory overhead.

(2) For Jaccard, Cosine, Dice, `AdaptJoin` and `PPJoin+` are the best choices. On large datasets, `AdaptJoin` outperforms other algorithms. On small datasets, for large similarity thresholds, `PPJoin+` outperforms other algorithms; for small thresholds, `AdaptJoin` beats other algorithms.

(3) For Edit Distance, the similarity join problem is hard for datasets with short strings as there are large numbers of results and it is hard to devise effective filtering techniques.

(4) For the q-gram based methods, parameter q has effect on the performance and it is not easy to select an appropriate parameter q to achieve high performance. Generally q should be neither too small nor very large.

(5) Most of existing algorithms cannot achieve high performance for really large datasets and it calls for new disk-based algorithms and parallel algorithms for really large datasets.

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

8. REFERENCES

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

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

[3] A. Behm, C. Li, and M. J. Carey. Answering approximate string queries on large data sets using external memory. In *ICDE*, pages 888–899, 2011.

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

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[6] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD*, 2014.

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

[8] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[9] J. Jestes, F. Li, Z. Yan, and K. Yi. Probabilistic string similarity joins. In *SIGMOD Conference*, pages 327–338, 2010.

[10] Y. Kim and K. Shim. Parallel top-k similarity join algorithms using mapreduce. In *ICDE*, pages 510–521, 2012.

[11] H. Lee, R. T. Ng, and K. Shim. Power-law based estimation of set similarity join size. *PVLDB*, 2(1):658–669, 2009.

[12] H. Lee, R. T. Ng, and K. Shim. Similarity join size estimation using locality sensitive hashing. *PVLDB*, 4(6):338–349, 2011.

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

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

[15] X. Lian and L. Chen. Set similarity join on probabilistic data. *PVLDB*, 3(1):650–659, 2010.

[16] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

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

[18] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[19] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.

[20] B. S. T. Bocek, E. Hunt. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007. http://fastss.csg.uzh.ch/.

[21] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD Conference*, pages 495–506, 2010.

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

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

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

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

[26] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

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

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