Efficient Parallel Partition-based Algorithms for Similarity Search and Join with Edit Distance Constraints

Yu Jiang† Dong Deng† Jiannan Wang† Guoliang Li‡ Jianhua Feng‡
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China.
†{y-jiang12, dd11, wjn08}@mails.tsinghua.edu.cn, ‡{liguoliang, fengjh}@tsinghua.edu.cn

ABSTRACT

The quantity of data in real-world applications is growing significantly while the data quality is still a big problem. Similarity search and similarity join are two important operations to address the poor data quality problem. Although many similarity search and join algorithms have been proposed, they did not utilize the abilities of modern hardware with multi-core processors. It calls for new parallel algorithms to enable multi-core processors to meet the high performance requirement of similarity search and join on big data. To this end, in this paper we propose parallel algorithms to support efficient similarity search and join with edit-distance constraints. We adopt the partition-based framework and extend it to support parallel similarity search and join on multi-core processors. We also develop two novel pruning techniques. We have implemented our algorithms and the experimental results on two real datasets show that our parallel algorithms achieve high performance and obtain good speedup.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems-*Textual Databases*; H.3.3 [[**Information Storage and Retrieval**]: Information Search and Retrieval-*Search Process*

General Terms

Algorithms, Experimentation, Performance

Keywords

Similarity Search, Similarity Join, Parallel Algorithms, Content Filter

1. INTRODUCTION

String similarity search and join are two essential operations in data cleaning and integration. Informally, given a collection of strings and a query string, *string similarity search* is to find all similar strings to the given query string; given two collections of strings, *string similarity join* is to find all pairs of similar strings from the two collections. Efficient approaches for similarity search and join are highly required by a variety of applications, such as data integration and document clustering.

Typically, a similarity function and a threshold are specified to determine whether two strings are similar or not. If the similarity of two strings computed by the similarity function is no smaller than the threshold, the two strings are taken as a similar string pair. There are many similarity functions, such as Jaccard, Cosine, and Edit Distance. In this paper we focus on using edit distance to quantify string similarity. The edit distance between two strings is defined as the minimum number of operations (deletion, insertion, replacement) to transform one string to the other. For example, consider two strings "`edby`" and "`edbt`". Their edit distance is ED("`edby`", "`edbt`") = 1 as we can transform "`edby`" to "`edbt`" by one replacement operation (i.e. replace "`y`" with "`t`"), and this is the minimum number of required operations. We say two strings are similar if their edit distance is no larger than the user specified edit-distance threshold. For this example, if the specified threshold is 1, then ("`edby`", "`edbt`") is returned as a similar string pair.

There are many studies on string similarity search and join. We can broadly classify them into three categories. (1) Gram-based method [14, 33, 31, 29]. They transform strings into grams and develop a filtering condition: if two strings are similar then they must share a certain number of common grams. To enhance the performance, they also proposed prefix-filtering based technique: they generated a prefix for each string by selecting some of its representative grams and deduced that if two strings are similar, their prefixes must share a common gram. They developed effective pruning techniques to generate high-quality prefixes. (2) Trie-based method [13, 18, 19, 27, 9]. They build a trie structure on top of strings and utilized the trie structure to do effective pruning: if two strings are similar, their prefixes must be *similar enough*[27]. Different from gram-based method, these methods can directly find all answers and do not employ a filter-and-verification framework. (3) Partition-based method [30, 28, 16]. They partitioned strings into segments and proved that if two strings are similar then they must share some common segments. They utilized this property to support similarity join and search.

In order to obtain a clear picture of the performance of state-of-the-art approaches, we extensively compare these methods on different datasets and make the following ob-

EDBT Workshop March 22, 2013, Genoa, Italy

servations. The trie-based method got high performance on the datasets with short strings and the gram-based method obtained high performance on the datasets with long strings. The partition-base method always achieved high performance for both short strings and long strings. Thus, in this paper, we adopt the partition-based method as our framework. We enable it to support parallel similarity search and join on multi-core processors. To further enhance the performance, we propose two pruning techniques, content filter and effective indexing strategy.

The rest of this paper is organized as follows. Section 2 formulates string similarity join and search problems. Section 3 introduces our framework. We propose two novel pruning techniques in Section 4, and a parallel algorithm in Section 5. Section 6 extends our join algorithm to support similarity search. We report experimental results in Section 7. Section 8 reviews related works and we conclude the paper in Section 9.

2. PROBLEM DEFINITION

In this section, we formally define string similarity join and search problems with edit-distance constraints.

DEFINITION 1 (STRING SIMILARITY JOINS). *Given two collections of strings $\mathcal{R}$ and $\mathcal{S}$, and an edit-distance threshold τ, a similarity join finds all string pairs $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$ such that* $\text{ED}(r, s) \leq \tau$.

Table 1: A collection of strings

(a) Strings

Strings
avataresha
caushik chakrabar
kaushic chaduri
kaushik chakrab
kaushuk chadhui
vankatesh

(b) Sorted strings

ID	Strings	Length
s_1	vankatesh	9
s_2	avataresha	10
s_3	kaushic chaduri	15
s_4	kaushik chakrab	15
s_5	kaushuk chadhui	15
s_6	caushik chakrabar	17

Without loss of generality, we focus on the self-join problem (i.e. $\mathcal{R} = \mathcal{S}$) in this paper. Note that our method can be easily extended to support similarity joins on two different collections. For example, consider the strings in Table 1(a). If the specified edit-distance threshold is $\tau = 3$, then ⟨"`kaushik chakrab`", "`caushik chakrabar`"⟩ will be returned as a similar pair since their edit distance is no larger than τ.

DEFINITION 2 (STRING SIMILARITY SEARCH). *Given a collection of strings $\mathcal{S}$, a query string q and an edit-distance threshold τ, a similarity search finds all strings $s \in \mathcal{S}$ such that* $\text{ED}(s, q) \leq \tau$.

For example, consider the strings in Table 1(a). Given the query string "`avatars`" and threshold $\tau = 3$, "`avataresha`" will be returned as its edit distance to the query is ED("`avatars`", "`avataresha`") ≤ 3.

In the following sections, we mainly focus on how to devise efficient parallel algorithms to support similarity join and search based on our partition-based framework [16].

Algorithm 1: SIMILARITYJOIN($\mathcal{S}, \tau$)

Input: $\mathcal{S}$: A collection of strings
τ: A given edit-distance threshold
Output: $\mathcal{A} = \{(s \in \mathcal{S}, r \in \mathcal{S}) \mid \text{ED}\ (s, r) \leq \tau\}$

```
1 begin
2     Sort S by string length in ascending order ;
3     Partition each s ∈ S and add its segments into
      L^i_{|s|}(1 ≤ i ≤ τ + 1);
4     foreach s ∈ S do
5         foreach L^i_l (|s| − τ ≤ l ≤ |s|, 1 ≤ i ≤ τ + 1) do
6             foreach w ∈SubstringSelection(s, L^i_l) do
7                 foreach r ∈ L^i_l[w] & |r| ≤ |s| do
8                     d = Verification(s, r, τ);
9                     if d ≤ τ then A ← ⟨r, s⟩;
```

Figure 1: Partion-based Algorithm

3. PARTITION-BASED FRAMEWORK FOR SIMILARITY JOINS

This section provides some background knowledge about our partition-based framework [16]. We first illustrate the basic idea. Consider two strings r and s, and a given edit-distance threshold τ. We partition r into $\tau + 1$ disjoint segments. If s has no substring that matches a segment of r, s cannot be similar to r [16], thus we can prune the pair $\langle r, s\rangle$ without computing their real edit distance. If s has a substring that matches a segment of r, we verity the pair $\langle r, s\rangle$. Notice that we do not use the dynamic programming algorithm to compute their edit distance [16]. Instead, we utilize the matching part to verify the pair.

For example, consider the strings in Table 1. Suppose $\tau = 3$. We partition s_1 = "`vankatesh`" into four segments {"`va`", "`nk`", "`at`", "`esh`"}. As strings s_3, s_4, s_5, s_6 have no substring which matches segments of s_1, they are not similar to s_1. Consider an another example. Consider s_5 = "`kausic_chakduri`" with four segments {"`kau`","`sic_`","`chak`", "`duri`"} and s_6 = "`caushik_chakrabar`". s_6 has a substring that matches the third segment "`chak`" of s_5. Then we utilize the matching part "`chak`" to verify the pair.

Based on this idea, we introduce the partition-based algorithm. The pseudo-code is shown in Figure 1. It first sorts strings by length in ascending order (line 2). Then, it generates segments for each string and builds an inverted index for each segment (line 3). Next it visits each string in sorted order (line 4). For each inverted index $\mathcal{L}^i_l(|s| - \tau \leq l \leq |s|, 1 \leq i \leq \tau + 1)$, it selects the substrings of s (line 5 to line 6). For any string r with length no larger than $|s|$ in inverted list $\mathcal{L}^i_l[w]$, the string pair $\langle r, s\rangle$ is a candidate pair and it verifies the pair (line 7 to line 8). If the pair pass the verification, it adds the pair into the result set (line 9).

In the algorithm, there are three challenges. The first one is how to partition the strings. The second one is how to select substrings (i.e., function SUBSTRINGSELECTION). The third challenge is how to verity the pair based on the matching part (i.e., function VERIFICATION). We will discuss the details in Sections 3.1, 3.2, and 3.3 respectively.

3.1 Partition Scheme

Given a string, there could be many strategies to partition

the string into $\tau+1$ segments. In the paper we use the *even-partition* scheme as an example [16]. Consider a string s with length $|s|$. In even partition scheme, each segment has a length of $\lfloor \frac{|s|}{\tau+1} \rfloor$ or $\lceil \frac{|s|}{\tau+1} \rceil$. Let $k = |s| - \lfloor \frac{|s|}{\tau+1} \rfloor * (\tau+1)$. In even partition, the last k segments have length $\lceil \frac{|s|}{\tau+1} \rceil$, and the first $\tau+1-k$ ones have length $\lfloor \frac{|s|}{\tau+1} \rfloor$. For example, consider s_1="vankatesh" and $\tau = 3$. Then length of s_1 ($|s_1|$) is 9. $k = 1$. s_1 has four segments {"va","nk","at", "esh"}.

3.2 Substring Selection

After partitioning a string into segments, we need to select substrings from another string and check if there is any selected substring matches any of the segments. Intuitively, we can select all the substrings. However this method is rather expensive. To address this issue, we have proposed several effective substring selection strategies and proved that the multi-match-aware substring selection method selected the minimum number of substrings in [16] . In the paper we use the multi-match-aware substring selection method.

Multi-match-aware Substring Selection. Given two strings r and s. Suppose we partition r into $\tau+1$ segments. For each segment of r, e.g., the i-th segment, we select some substrings of s and check whether they match the i-the segment of r. Suppose the start position of the i-th segment of r is p_i. The multi-match-aware substring selection method only selects substrings with start position in $[\perp_i, \top_i] = [p_i - (i-1), p_i + (i-1)] \cap [p_i + \triangle - (\tau+1-i), p_i + \triangle + (\tau+1-i)]$ where $\triangle$ is the length difference of the two strings.

For example, consider string r = "vankatesh" with four segments, {va, nk, at, esh}. Consider string s = "avataresha". For the first segment, we have $\perp_i = 1-0 = 1$ and $\top_i = 1+0 = 1$. We select "av" for the first segment. For the second segment, we have $\perp_i = 3-1 = 2$ and $\top_i = 3+1 = 4$. We select substrings "va", "at", and "ta" for the second segment. For the third segment, we have $\perp_i = 5+1-(3+1-3) = 5$ and $\top_i = 5+1+(3+1-3) = 7$. We select substrings "ar", "re", and "es" for the third segment. For the fourth segment, we have $\perp_i = 7+1-(3+1-4) = 8$ and $\top_i = 7+1+(3+1-4) = 8$. Thus we select the substring "sha" for the fourth segment.

3.3 Verification

If string s has a substring that matches a segment of r, we need to verify this string pair. We have proposed two verification methods, length-aware verification method and extension-based verification method, in [16]. In the dynamic programming algorithm to compute edit distance, the length-aware verification method utilizes the length difference to estimate the minimum number of edit operations which can reduce the time cost. The extension-based verification method partitions r and s into three parts, the matching parts ($r_m = s_m$), the left parts r_l, s_l (on the left side of the matching part), and the right parts r_r, s_r. If the left parts or the right parts are dissimilar within deduced threshold, it can prune the pair. Formally, if r and s are similar, then $\text{ED}(r_l, s_l) \leq i-1$ and $\text{ED}(r_r, s_r) \leq \tau+1-i$. If any condition is not true, we can prune the pair.

For example, suppose $\tau = 3$ and we want to verify s_5 = "kausic chakduri" and s_6 = "caushik chakrabar". s_5 and s_6 share a segment "chak". We have the left parts s_{5_l} = "kausic_" and s_{6_l} = "caushik_", and the right parts s_{5_r} = "duri" and s_{6_r} = "rabar". As $\text{ED}(s_{5_r}, s_{6_r}) = 3$ we can eliminate the pair of matching substring and segment as r and s cannot be similar by aligning the third segment of r to the substring of s [16].

4. PRUNING TECHNIQUES

In this section, we propose two **new** pruning techniques, *content filter* and *effective indexing strategy*, to further enhance our partition-based framework.

4.1 Content Filter

After we obtain a set of string pairs that share common segments, we aim to verify each string pair efficiently in order to avoid expensive edit-distance computation. Hence, we propose content filter to achieve this goal. The idea is a little similar to content-based mismatch filtering [31] but with several major improvements: 1) we derive a stronger filter condition by considering string length difference; (2) we group symbols in order to improve the content-filter running time; (3) we integrate content filter with our extension-based verification method in order to enhance the content-filter pruning power for large edit-distance thresholds.

Stronger Filter Condition

Let Σ be a set of distinct symbols in the collection of strings. For each string s, let H_s denote its frequency histogram, where H_s is a $|\Sigma|$-dimensional vector, and its i-th dimension $H_s[i]$ represents the number of occurrences that the symbol $\delta_i \in \Sigma$ appears in s. Given two strings r and s, the $\mathcal{L}_1$ distance between their frequency histograms is defined as $\mathsf{SUM}_{1 \leq i \leq |\Sigma|} \left| H_r[i] - H_s[i] \right|$. If they are similar within edit distance τ, since an edit distance at most changes the distance by two, the $\mathcal{L}_1$ distance of their frequency histograms should be no larger than 2τ.

Our next goal is to derive a stronger filter condition. Without loss of generality, suppose $|r| \geq |s|$, and let $\triangle = |r| - |s|$ denote the length difference between r and s. Consider the edit operations that are used to transform r to s. As a deletion or insertion operation changes the number of characters in r by one while a substitution operation cannot change the number of characters in r, the transformation with the minimum number of edit operations from r to s contains at least $\triangle$ deletion operations and at most $\tau - \triangle$ substitution operations. In addition, a deletion or insertion operation changes the $\mathcal{L}_1$ distance between the frequency histograms by one and a substitution operation changes it by two. Thus, the $\mathcal{L}_1$ distance between the frequency histograms of r and s is at most $2(\tau - \triangle) + \triangle = 2\tau - \triangle$.

We can further optimize the filter condition by taking into account the frequency difference w.r.t each symbol. For the string s, a deletion operation, an insertion operation, or a substitution operation can at most change a symbol's frequency by one. Thus for each symbol, the frequency difference between r and s should be no larger than τ.

In summary, our new filter condition, namely content filter, can prune a string pair if the two strings have any symbol with frequency difference larger than τ or the $\mathcal{L}_1$ distance of their frequency histograms is larger than $2\tau - \triangle$.

For example, consider two strings "abcdef" and "axxcdexf", and suppose the edit-distance threshold is $\tau = 2$. We partition "abcdef" into three segments using even partition scheme and get "ab", "cd", and "ef". The multi-match-aware substring selection method returns this pair of strings as a candidate pair since the selected substring "cd" from "axxcdexf" matches the segment "cd" in "abcdef". Using the content

filter, we can prune this pair of strings since either the frequency difference of symbol "x" is 3 which is more than $\tau = 2$, or the $\mathcal{L}_1$ distance between the two strings is 4 which is larger than $2\tau - |\triangle| = 2$. Note that the content-based mismatch filtering [31] cannot filter this pair of strings.

Grouping Symbols

Sometimes the symbol size $|\Sigma|$ is relatively large compared to string lengths. For example, the city name dataset used in our experiment has a table with about 200 symbols while the average string length is no more than 12. Thus the cost to scan the entire symbol table might be very high.

To solve this problem, we divide the symbols in Σ into $k(< |\Sigma|)$ disjoint symbol groups, and set the group frequency of a symbol group as the sum of frequencies of all the symbols in this group. It is easy to prove that the above content-filter condition still holds after grouping the symbols. That is, if two strings are similar within edit-distance threshold τ, for any symbol group, its group frequency difference between the two strings should be no more than τ, and the $\mathcal{L}_1$ distance between the group frequency histograms of the two strings should be no more than $2\tau - \triangle$. By grouping symbols, we can significantly reduce the scan cost to the number of symbol groups (i.e., k), thus improve the content-filter running time.

For example, consider the two strings "abcdef" and "axxcdexf" and suppose the edit distance threshold $\tau = 2$. If "a", "b", "c" and "d" belong to the same group, and "e", "f" and "x" belong to another group of symbols, we can prune the pair of strings since the $\mathcal{L}_1$ distance between their group frequency histograms is $1 + 3 = 4$ which is larger than $2\tau - |\triangle| = 2$.

Integrating the two techniques with our extension-based verification method

In order to enhance the content-filter pruning power for large thresholds, we integrate content filter with our extension-based verification method. Given two strings r and s, if the multi-match-aware substring selection method finds a substring s_m of s matching the i-th segment r_m of r, the extension-based verification method needs to verify whether $\text{ED}(r_l, s_l) \leq i-1$ and $\text{ED}(r_r, s_r) \leq \tau+1-i$ using length-aware verification method. By applying content filter to left strings and right strings respectively, we need to (1) check the left strings whether their $\mathcal{L}_1$ distance is within $2(i-1)-\big||r_l|-|s_l|\big|$ and the group frequency differences of each symbol group are all within $i-1$; (2) check the right strings whether their $\mathcal{L}_1$ distance is within $2(\tau+1-i)-\big||r_r|-|s_r|\big|$ and the group frequency differences of each symbol group are all within $\tau+1-i$. Note that the thresholds that we need to check are much smaller than the original threshold, thus potentially leading to better pruning power.

For example, suppose $|r| = |s| = 102$, edit-distance threshold $\tau = 16$ and a substring of s with start position 39 matches the 8-th segment of r with start position 43. Instead of checking whether the $\mathcal{L}_1$ distance is larger than 32 we only need to check wether the $\mathcal{L}_1$ distance of r_l and s_l is within $2(i-1)-\big||r_l|-|s_l|\big| = 2*(8-1)-|43-39| = 10$ and the $\mathcal{L}_1$ distance of r_r and s_r is within $2(\tau+1-i)-\big||r_r|-|s_r|\big| = 2*(16+1-8)-4 = 14$.

4.2 Effective Indexing Strategy

Consider two strings "abcdef" and "adcxxef" and suppose edit-distance threshold is $\tau = 2$. If we partition "abcdef" into three segments using even partition scheme, "ab", "cd" and "ef", our multi-match-aware substring selection method considers this pair as a candidate pair as they share a common substring/segment "ef" and content filter cannot prune it either as their $\mathcal{L}_1$ distance is 3 which is within $2\tau - \triangle$ and the frequency differences of all symbols are also within $\tau = 2$. But if we partition "adcxxef" into three segments using even partition scheme, 'ad", 'cx" and 'xef", multi-match-aware substring selection method cannot select a substring from "abcdef" which matches with any of the three segments, thus this pair can be pruned.

Intuitively, the longer the length of a segment, the lower the possibility it matches a substring selected by the multi-match-aware substring selection method. Thus we sort the given collection of strings in the decreasing order of string length, and index the longer strings which have a longer average length of segments. Then we select substrings from shorter strings and use them to find candidate pairs based on the indexes of longer strings.

Algorithm 2: ParallelSimilarityJoin($\mathcal{S}, \tau$)

Input: $\mathcal{S}$: A collection of strings
τ: A given edit-distance threshold
Output: $\mathcal{A} = \{(s \in \mathcal{S}, r \in \mathcal{S}) \mid \text{ED}\ (s,r) \leq \tau\}$

```
1  begin
       // parallel sorting
2      Paralleled sort S by string length in descending order ;
       // parallel indexing longer strings
3      Paralleled build inverted indexes
       L_l^i (l_min ≤ l ≤ l_max, 1 ≤ i ≤ τ + 1) ;
       // spliting datasets
4      Split S into several small datasets S' and each
       thread processes one dataset S' ;
       // signle thread algorithm
5      foreach s ∈ S' do
6          foreach L_l^i (|s| ≤ l ≤ |s| + τ, 1 ≤ i ≤ τ + 1) do
7              foreach w ∈SubstringSelection(s, L_l^i) do
8                  foreach r ∈ L_l^i[w] & |r| ≥ |s| do
                       // content filters
9                      ContentFilter(s_l, r_l, i) ;
10                     ContentFilter(s_r, r_r, τ + 1 − i) ;
11                     d = Verification(s, r, τ);
12                     if d ≤ τ then A ← ⟨r, s⟩;
```

Figure 2: Parallel Similaity Join

5. PARALLEL SIMILARITY JOIN

In this section, we extend our algorithm and devise a parallel algorithm. The algorithm includes three main components, the sort phrase, the indexing phrase and the similarity joining phrase. The pesudo-code is illustrated in Figure 2.

Parallel Sorting: We first sort the dataset by string length in descending order to index the longer strings. We use existing parallel sorting algorithm (line 2) to sort the strings. The experiment results show the speedup can reach up to 6x with 8 threads.

Parallel Indexing: In the partition-based framework we

need to build inverted indexes $\mathcal{L}_l^i$ for each $l_{max} \leq l \leq l_{min}$ and $1 \leq i \leq \tau+1$. As all inverted indexes are disjoint to each other, we can easily build the indexes in parallel (line 3). The experiment results show the speedup of the indexing phrase is about 3x with 8 threads. This step has poor parallelism. The main reason may be from the cache architecture. The (L1/L2) cache can be well-utilized in a single thread and our parallel algorithm may involve more cache miss.

Parallel Join: To execute the similarity join operation in parallel, we first split the dataset $\mathcal{S}$ into several small datasets $\mathcal{S}'$ and use each thread to process a small dataset (line 4). Note that, before utilizing the extension-based verification method to verify two strings r and s, we apply the content filter on the left side an right side of r and s (line 9 to line 10). The experiment results show the speedup can reach 6x with 8 threads.

6. PARALLEL SIMILARITY SEARCH

The similarity search problem is similar to the similarity join problem except that the edit distance threshold is not given in advance. To solve this problem, we first build inverted indexes for each possible edit-distance thresholds. Notice that we can only build index for some thresholds. Then for each query string, we utilize the corresponding indexes to find its similar strings. The pseudo-code is shown in Figure 3.

Notice that we build inverted indexes in a pre-processing step (line 2). For each query with edit-distance threshold τ, we select inverted indexes $\mathcal{L}_l^i(\tau')$ with the smallest edit-distance threshold τ' such that $\tau' \geq \tau$ (line 3). Then, we select substrings for this query string using multi-match-aware substring selection method and verify all candidate pairs using extension-based algorithm with edit distance τ'(line 4 to line 9). The extension-based algorithm returns a value *ed*. If $ed \leq \tau$ we add this pair into result set as we can find a transformation from r to q with the number of edit operations not larger than τ. If $ed > \tau'$ we drop this pair. Otherwise, we need to call the length-aware verification method to compute their real edit distance(line 9 to line 12).

To parallel the similarity search algorithm, we only need to split all the query strings into several parts and use each thread to process one part of the query set. Thus our algorithm can achieve good speedup.

7. EXPERIMENT

In this section we evaluate the parallel similarity search and join algorithms. Our goal is to evaluate (1) the effectiveness of new pruning techniques, (2) the parallelism of our algorithms, (3) the scalability of our algorithms.

Setup: All the algorithms were implemented in C++. The programs were complied by GCC 4.7.2 with -O3 and -pthread flags. We ran our programs on a Fedora machine with 16 Intel Xeon E5-2650 2GHz processors and 64GB memory.

Dataset: We used the two real datasets Reads and GeoNames provided by the organizer of the workshop for string similarity search/join competition*. Reads was a human genome read dataset which consisted of 750,000 reads from different human genomes. Its symbol size was 5. GeoNames consisted of 400,000 city names from all over the world

*http://www2.informatik.hu-berlin.de/~wandelt/searchjoincompetition2013/

Algorithm 3: SIMILARITYSEARCH($\mathcal{S}$, q, τ)

Input: $\mathcal{S}$: A collection of strings
q: query string; τ: query edit distance threshold;
Output: $\mathcal{A} = \{s \in \mathcal{S} \mid \text{ED}(s, q) \leq \tau\}$

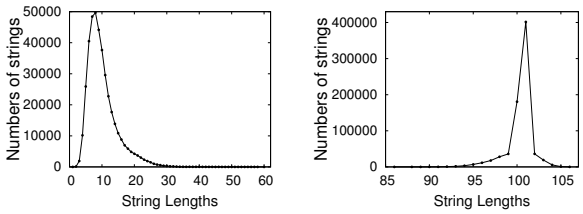
```
1  begin
       // build inverted indexes off-line
2      Paralleled build all the inverted indexes L_l^i(t) for
       each 0 ≤ t ≤ τ_m(l_min ≤ l ≤ l_max, 1 ≤ i ≤ t + 1) ;
       // online processing queries
3      Select inverted indexes L_l^i(τ') with smallest edit
       distance threshold τ' such that τ' ≥ τ ;
4      foreach L_l^i(τ')(|q|−τ'≤l≤|q|+τ', 1≤i≤τ'+1) do
5          foreach w ∈SubstringSelection(q, L_l^i(τ')) do
6              foreach r ∈ L_l^i(τ')[w] do
                   // content filters
7                  ContentFilter(q_l, r_l, i) ;
8                  ContentFilter(q_r, r_r, τ' + 1 − i) ;
9                  ed =Verification(q, r, τ') ;
10                 if ed ≤ τ then add r into A ;
11                 else if ed > τ' then continue ;
12                 else verify(q, r, τ) ;
```

Figure 3: Parallel Similarity Search

and its symbol size was about 200. We also synthetically generated two query datasets for similarity search and each dataset consisted of 100,000 queries. In the following experiments, we assigned the same edit-distance thresholds for each of the queries. We give some statistic of the four datasets in Table 2. The length distributions are shown in Figure 4.

Table 2: Datasets

Datasets	cardinality	avg len	min len	max len
GeoNames	400,000	11.1	1	60
GeoName Query	100,000	10.7	2	43
Reads	750,000	101.4	86	106
Read Query	100,000	101.2	88	116



(a) GeoNames (b) Reads

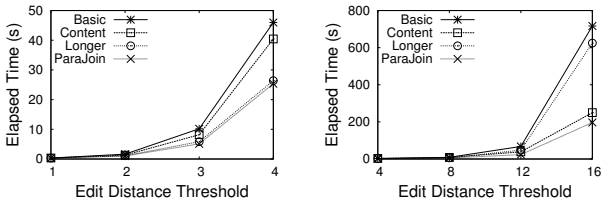
Figure 4: Length distribution.

7.1 Evaluating Pruning Techniques

In this section we evaluate the effectiveness of our pruning techniques for similarity search and join.

Similarity Join. We have implemented four methods based on the two pruning techniques proposed in Section 4 for sim-

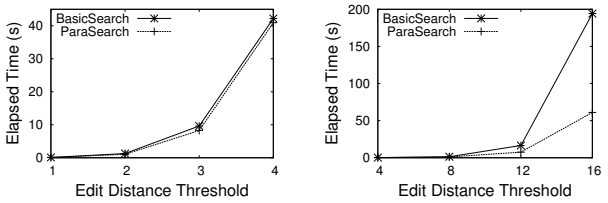
ilarity join. The `Basic` algorithm does not apply any pruning techniques. The `Content` algorithm applies the content-filter technique and indexes the shorter strings. The `Longer` algorithm indexes the longer strings but does not use the content-filter technique. The `ParaJoin` algorithm contains the content-filter technique and indexes the longer strings. We report the running time of each of the four methods on Reads and GeoNames dataset as shown in Figure 5. We observe that the `ParaJoin` algorithm achieved the highest performance and the `Longer` and `Content` algorithms outperformed the `Basic` algorithm. For example, on Reads dataset with edit-distance threshold $\tau = 16$, the elapsed time for `Basic` algorithm was about 800 seconds while the `ParaJoin` algorithm only used 200 seconds, leading to 3x faster. This shows that our content filter technique and indexing longer strings technique can improve the performance. Hereafter, we only show the performance of the `ParaJoin` algorithm for similarity join.



(a) GeoNames (b) Reads

Figure 5: Evaluating effectiveness of the pruning techniques for similarity join(8 threads).

Similarity Search. Since similarity search indexes all the strings, the index longer strings filter is not applicable in this problem. Thus we only implemented two methods `BasicSearch` and `ParaSearch` for similarity search based on whether using the content filter or not. The experimental results on two datasets are shown in Figure 6. We can see that the `ParaSearch` method was better than the `BasicSearch` method. For example, in Reads dataset, the `ParaSearch` algorithm used 60 seconds while the `BasicSearch` algorithm took 200 seconds. This also shows the effectiveness of the content filter on similarity search and hereafter we only show the performance of the ParaSearch algorithm for similarity search.

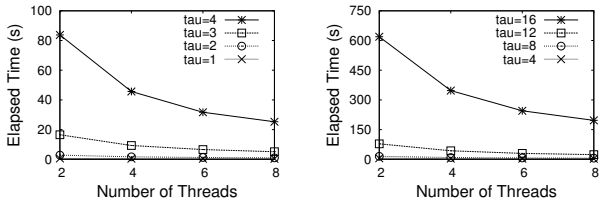


(a) GeoNames (b) Reads

Figure 6: Evaluating effectiveness of the pruning techniques for similarity search(8 threads).

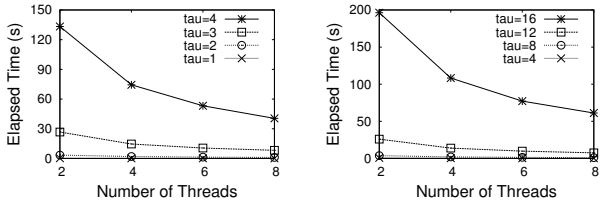
7.2 Evaluating Parallelism

In this section, we evaluate the running time of our similarity search and join algorithms by varying the number of threads from 2 to 8. The experimental results are shown in Figure 7 and Figure 8. We can see that the running time is decreased with the increase of the number of threads used. For example, the elapsed time of the similarity join algorithm with edit-distance threshold $\tau = 16$ on `Reads` dataset was about 600 seconds when using 2 threads and the time decreases to 200 seconds when using 8 threads. For the similarity search algorithm on `GeoNames` dataset with $\tau = 4$, the elapsed time was 130 seconds when number of threads used is 2 while the time was 40 seconds when using 8 threads. The main reason is as follows. For similarity join, we can parallel sorting, indexing and joining steps. Although the indexing step cannot achieve high parallelism, it as dominated by the joining step, thus the algorithm still achieved high overall parallelism. For similarity search, we can use different threads to answer different queries thus our algorithm achieved high parallelism.



(a) GeoNames (b) Reads

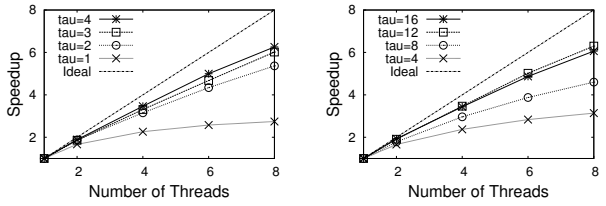
Figure 7: Evaluating running time of similarity join by varying number of threads.



(a) GeoNames (b) Reads

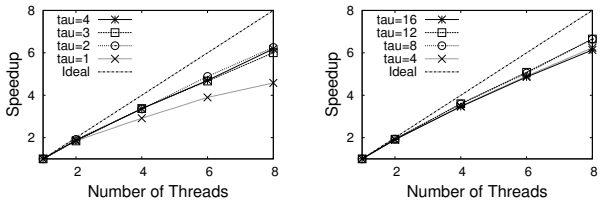
Figure 8: Evaluating running time of similarity search by varying number of threads.

We also evaluated the speedups of our similarity search and join algorithms as shown in Figure 9 and Figure 10. The speedup is the ratio of time consuming of the parallel program with that of the single thread program. Except the speedup of our algorithms with different edit-distance thresholds, we also illustrated the ideal speedup curve. We can see that our algorithms have good speedups especially when τ is large. For example, the 8-threads similarity join algorithm have a speedup of 6 when $\tau = 4$ on the `GeoNames` dataset and the 8-threads similarity search algorithm achieved a speedup of 7 when $\tau = 8$ on the `Reads` dataset.



(a) GeoNames (b) Reads

Figure 9: Evaluating speedup of similarity join.

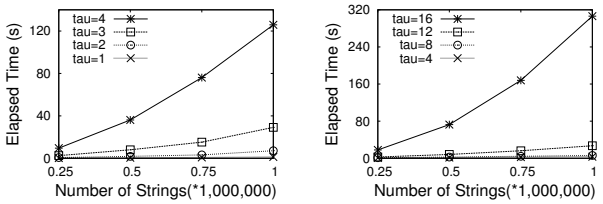


(a) GeoNames (b) Reads

Figure 10: Evaluating speedup of similarity search.

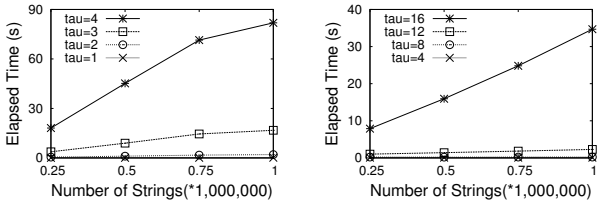
7.3 Evaluating Scalability

In this section we evaluated the scalability of our parallel algorithms. Figure 11 and Figure 12 show the running time of our algorithms on datasets with different sizes. We increased the size of original datasets by following the same rules of frequencies of symbols and distributions of string lengths. We can see our algorithms achieved very good scalability. For example, for the similarity search algorithm on `GeoNames` dataset with edit distance threshold $\tau = 4$, the running time for dataset sizes of 0.25, 0.5, 0.75 and 1 millions are 8, 16, 25, and 34 seconds respectively.



(a) GeoNames (b) Reads

Figure 11: Evaluating the scalability of the similarity join algorithm(8 threads).



(a) GeoNames (b) Reads

Figure 12: Evaluating the scalability of the similarity search algorithm(8 threads).

8. RELATED WORKS

There are many related works on string similarity join [10, 24, 6, 1, 2, 25, 31, 32, 26, 27, 23, 16, 28, 22, 29] and string similarity search [5, 15, 14, 11, 34, 12, 3, 35, 4, 17, 21, 20, 8].

String similarity join. Most existing works used signature-based framework to solve string-similarity-join problem. They first generated a set of signatures for each string and ensured that every similar string pair must share at least one common signature, and then utilized some index structures (e.g. inverted index) to filter string pairs whose signature sets have no overlap, and finally only verified the remaining string pairs by comparing their real edit distance. A variety of signature schemes are proposed to achieve this goal, such as q-gram [10], deletion-based neighborhood [25], gram-chunk [23], and partition-based segments [16]. In addition to signature schemes, many works also try to optimize the filtering step by reducing redundant signatures [6, 2, 31], or improve the verification performance by avoiding expensive edit-distance computation [16]. Trie-Join is another direction **??** which used trie structure to achieve high performance. It can directly find similar pairs and avoid the verification step.

String similarity search. Similarity-search approaches will first build indexes for the string collection. Then, for a given query string, they utilized the index to filter a large number of dissimilar strings to the query string, and only verified the survived strings to find similar strings to the query string. In terms of indexing structures, most approaches [14, 11, 3, 12, 4] employed an inverted index for storing mappings from signatures to strings in the data. Another different approach adopted B^{ed}-tree [35], which is a B^+-tree based index structure with the benefit of supporting a diverse set of query types, such as top-k query and range query. In terms of searching, various filtering algorithms (e.g.,DivideSkip [14]) were proposed to efficiently find similar strings to the query string based on the constructed index structure. Dong et al.[7] proposed progressive algorithms to find top-k similar strings.

9. CONCLUSION

In this paper we study the problem of similarity search and joins with edit distance constraints. We proposed efficient parallel algorithms to accelerate similarity search and similarity joins. We utilize the partition-based framework [16] and integrate it with two novel pruning techniques. We discuss how to parallel the partition-based framework and how to support similarity searches. Experiments show high efficiency and good speedup of our algorithms.

10. ACKNOWLEDGEMENT

This work was partly supported by the National Natural Science Foundation of China under Grant No. 61003004 and 61272090, National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, and a project of Tsinghua University under Grant No. 20111081073, and the "NExT Research Center" funded by MDA, Singapore, under Grant No. WBS:R-252-300-001-490.

11. 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, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, pages 604–615, 2009.
[4] 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.
[5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.

[6] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

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

[8] J. Feng and G. Li. Efficient fuzzy type-ahead search in xml data. *IEEE Trans. Knowl. Data Eng.*, 24(5):882–895, 2012.

[9] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.

[10] 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.

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

[12] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD Conference*, pages 429–440, 2009.

[13] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 433–439, 2009.

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

[15] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

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

[17] G. Li, J. Feng, and C. Li. Supporting search-as-you-type using sql in databases. *IEEE Trans. Knowl. Data Eng.*, 25(2):461–475, 2013.

[18] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD Conference*, pages 695–706, 2009.

[19] G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20(4):617–640, 2011.

[20] G. Li, S. Ji, C. Li, J. Wang, and J. Feng. Efficient fuzzy type-ahead search in tastier. In *ICDE*, pages 1105–1108, 2010.

[21] G. Li, J. Wang, C. Li, and J. Feng. Supporting efficient top-k queries in type-ahead search. In *SIGIR*, pages 355–364, 2012.

[22] 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.

[23] 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.

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

[25] 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/.

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

[27] 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.

[28] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[29] 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.

[30] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD Conference*, pages 759–770, 2009.

[31] 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.

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

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

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

[35] 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*, pages 915–926, 2010.