

Cache-Aware Parallel Approximate Matching and Join Algorithms Using BWT

Jiaying Wang
College of Information Science
and Engineering,
Northeastern University,
Liaoning 110819, China
wangjiaying@research.neu.
edu.cn

Xiaochun Yang
College of Information Science
and Engineering,
Northeastern University,
Liaoning 110819, China
yangxc@mail.neu.edu.cn

Bin Wang
College of Information Science
and Engineering,
Northeastern University,
Liaoning 110819, China
binwang@mail.neu.edu.cn

ABSTRACT

Nowadays, approximate string search and join, as essential operations in data integration and cleaning, has attracted significant attentions in academic. In this paper, we study string similarity search and join with edit distance constraints. Although multicore machines have become the mainstream computer architecture, most existing methods only work on a uniprocessor. To address this problem, we propose a novel parallel framework using BWT. We also devise efficient technique to utilize cache to further speed up the performance. Our method can solve similar search and join efficiently and generally. We conducted a comprehensive experimental study of our method to demonstrate the efficiency.

Keywords: Approximate string search, Similarity join, Edit distance, BWT

1. INTRODUCTION

There are so many applications where we would like to find approximate pattern matches, rather than exact occurrences. In practice, the approximate pattern matching algorithms are very useful. For example, searching a text database with keywords may have spelling errors. So providing approximate results may be more satisfying to users. And approximate pattern matching has already been used as powerful tools in the study of genomes, such as sequences alignment of a DNA sequence.

Approximate string matching typically contains two sub-problems: Finding approximate strings inside a string collection and approximate strings matches inside a given string. In this paper, we focus on solve the former problem, and leave the latter problem to future work.

There are many similarity functions to measure distance between two strings, such as edit distance, jaccard similarity, cosine metrics and so on. In this paper, we will focus on edit distance, which is the most common metric method.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT/ICDT'13 March 18 - 22, 2013, Genoa, Italy
Copyright 2013 ACM 978-1-4503-1599-9/13/03 ...\$15.00.

Approximate string search usually takes advantage of exact match to select candidates, and verifies these candidates to get the answer. Therefore we can improve the two processes to boost the overall performance. To improve the candidate selection, we take advantage of an improved BWT index to get exact match substrings. To improve the verification, we utilize a look ahead method to optimize the computation.

Nowadays, multicore machines have become the mainstream computer architecture. Combining with the multicore technique, we could accelerate the approximate string matching. However, most current methods still only work on single core, which enlightens us to devise a multicore framework. Another important component people often ignore is the cache. Cache is smaller but faster than memory. Try to re-use the data which are already in cache can help us to get extra performance.

Challenge: The challenge of this work is to develop an efficient cache-aware algorithm on a multicore machine to support approximate string search and join.

Contributions: We make the following contributions in the paper.

- We propose a cache-aware multicore framework to support approximate string search and join using BWT.
- We devise efficient pruning techniques to improve the performance.
- We develop a look ahead algorithm to support bounded edit distance and improve the verification of the candidate strings.
- We have implemented our method. And the experiment shows the efficiency of our method.

Related work: Existing methods to address the problem include metric space-based methods, signature-based methods and trie-based methods.

- The metric space-base methods take the advantage of triangle inequality property to prune dissimilar strings in metric space. These methods include BK-Tree [1], VP-Tree [2] and M-Tree [3].
- The signature-based methods usually use a filter-and-refine framework. In filter phase, these methods will

generate candidate pairs based on signature; In refine phase, these methods will verify the candidates to get the answer. Many strategies were proposed to increase the filter capability. These methods include Part-Enum [4], All-Pairs-Ed [5], DivideSkip [6], Vgram [7, 8], Ed-Join [9], Qchunkgram [10], and PassJoin [11].

- The trie-based methods use a trie structure to take the advantage of common prefixes of the strings to reduce repeated computations [12].

There have been many studies on backward search on BWT [13, 14, 15, 16, 17]. The method has been used in bioinformatics to solve sequence alignment problem [18, 19].

We organize the rest of the paper as follows: We introduce all associated vocabularies and structures and give the problem definitions in Section 2 and propose a query algorithm using a BWTPA structure in Section 3. Many improvements on the basic framework are proposed in Section 4. Approximate string join method is studied in Section 5. There are experimental results in Section 6. Finally, we summarize our conclusion in Section 7.

2. PRELIMINARIES

We represent the text collection \mathcal{C} as $T_1, T_2 \dots T_k$. And T_i contains $|T_i|$ characters over an ordered alphabet $\Sigma = \{c_1, c_2 \dots c_{|\Sigma|}\}$, where $|\Sigma|$ is the size of the alphabet. We represent the pattern to be searched for as P , which contains $|P|$ characters over the same alphabet Σ .

Edit Distance: Edit distance sometimes is called Levenshtein distance. Given two strings $S_1 = a_1 a_2 \dots a_m$ and $S_2 = b_1 b_2 \dots b_n$. The problem is to find the minimum number of edit operations to transform S_1 to S_2 . The operations are insertion, deletion and substitution.

Edit distance can be obtained by a dynamic programming algorithm. We define $d(i, j)$ to be the edit distance between the prefix string $S_1[1 \dots i]$ and $S_2[1 \dots j]$. The recurrence relation is

$$d(i, j) = \min \begin{cases} d(i-1, j) + 1 \\ d(i, j-1) + 1 \\ d(i-1, j-1) + c(i, j) \end{cases} \quad (1)$$

where $c(i, j) = 0$ if $a_i = b_j$; otherwise $c(i, j) = 1$.

We define $ed(S_1, S_2)$ to be the edit distance of the two strings, then $ed(S_1, S_2) = d(m, n)$.

Approximate String Search: Given a text collection \mathcal{C} , a pattern P , and threshold τ , the problem is to find all $T_i \in \mathcal{C}$ such that $ed(P, T_i) \leq \tau$.

For example, consider the set of strings shown in Figure 1(a). We want to search the patterns according to thresholds in Figure 1(b). Searching the first pattern *Majaura* and $\tau = 1$ will yield $1 : \{2, 5\}$ as the result.

Approximate String Join: Given a text collection \mathcal{C} , and threshold τ , the problem is to find all pairs $T_i \in \mathcal{C}$, $T_j \in \mathcal{C}$, such that the $ed(T_i, T_j) \leq \tau$ and $i \neq j$.¹

For example, suppose we have a set of strings as in Figure 1(a), and want to find all similar pairs with a given

¹We focus on self join in this paper.

id	Strings
1	Dehri
2	Majaura
3	Deghli
4	lodna
5	Madhura

(a) Strings.

id	Strings	τ
1	Madaura	1
2	muradgy	2
3	Madghuhra	2

(b) Patterns.

Figure 1: A set of strings, patterns and threshold.

threshold. If $\tau = 2$, we can get $\{1, 3\}$ and $\{2, 5\}$ as the result.

Suffix Array: Given a string S , its suffix array SA records the start positions of all the suffixes of one string. Since the suffixes are sorted lexicographically, $SA[i]$ is the start position of rank i suffix based on the lexicographical order.

BWT: Given a string S , BWT is permutation of S . We use array L as the result of the transform. The relationship of L and SA is:

$$L[i] = \begin{cases} T[SA[i] - 1] & \text{if } SA[i] \neq 0, \\ \$ & \text{if } SA[i] = 0. \end{cases} \quad (2)$$

3. AN APPROXIMATE STRING SEARCH ALGORITHM

In this section, we propose an algorithm on approximate pattern matching with the help of Burrows-Wheeler Transform, which can help to efficiently query substrings. Before giving the algorithm, we introduce our new index BWTPA.

3.1 Improving BWT Index

As we know, BWT is a transform which can support substring search. However, we want to search inside a text collection, not a string. Therefore we propose a new data structure called BWTPA to keep BWT and the id information of strings. BWTPA index contains two parts, a BWT array and a position array PA . $PA[i]$ records the string id of every $SA[i]$. Therefore we do not need the SA array any more, and replace the SA with PA . We use B to represent BWTPA, such that $B = L + PA$.

We concatenate the strings in collection \mathcal{C} with a special character “\$” and add another special character “#” to the end the last string².

For instance, suppose there are five words in the collection \mathcal{C} : Dehri, Majaura, Deghli, lodna, Madhura. We concatenate the words and get the following text:

Dehri\$Majaura\$Deghli\$lodna\$Madhura#

Transforming the text can yield the SA as follows:

34 13 26 5 20 14 0 27 6 33 12 25 28 7 9 29 23 15 1
16 17 2 30 4 19 8 18 21 24 22 32 11 3 31 10,

and the corresponding BWT is:

aaaii\$\$\$rrnMMjaoDdegdrlah\$dluuhha

We transform the SA and get the following PA:

²In practice, we use “\n” as “\$”, and “\0” as “#”, user can change it to any other characters $c \notin \Sigma$.

	L	SA	PA
1 #Dehri\$Majaura\$Deghli\$lodna\$Madhur a	34	5	
2 \$Deghli\$lodna\$Madhura#Dehri\$Majaur a	13	2	
3 \$Madhura#Dehri\$Majaura\$Deghli\$lodn a	26	4	
4 \$Majaura\$Deghli\$lodna\$Madhura#Dehri	5	1	
5 \$lodna\$Madhura#Dehri\$Majaura\$Deghli	20	3	
6 Deghli\$lodna\$Madhura#Dehri\$Majaura \$	14	3	
7 Dehri\$Majaura\$Deghli\$lodna\$Madhura #	0	1	
8 Madhura#Dehri\$Majaura\$Deghli\$lodna \$	27	5	
9 Majaura\$Deghli\$lodna\$Madhura#Dehri \$	6	2	
10 a#Dehri\$Majaura\$Deghli\$lodna\$Madhu r	33	5	
11 a\$Deghli\$lodna\$Madhura#Dehri\$Majaur	12	2	
12 a\$Madhura#Dehri\$Majaura\$Deghli\$lodn	25	4	
13 adhura#Dehri\$Majaura\$Deghli\$lodna\$ M	28	5	
14 ajaura\$Deghli\$lodna\$Madhura#Dehri\$ M	7	2	
15 aura\$Deghli\$lodna\$Madhura#Dehri\$Ma j	9	2	
16 dhura#Dehri\$Majaura\$Deghli\$lodna\$M a	29	5	
17 dna\$Madhura#Dehri\$Majaura\$Deghli\$ l	23	4	
18 eghli\$lodna\$Madhura#Dehri\$Majaura\$ D	15	3	
19 ehri\$Majaura\$Deghli\$lodna\$Madhura# D	1	1	
20 ghli\$lodna\$Madhura#Dehri\$Majaura\$D e	16	3	
21 hli\$lodna\$Madhura#Dehri\$Majaura\$D e	17	3	
22 hri\$Majaura\$Deghli\$lodna\$Madhura#D e	2	1	
23 hura#Dehri\$Majaura\$Deghli\$lodna\$Ma d	30	5	
24 i\$Majaura\$Deghli\$lodna\$Madhura#Deh r	4	1	
25 i\$lodna\$Madhura#Dehri\$Majaura\$Degh l	19	3	
26 jaura\$Deghli\$lodna\$Madhura#Dehri\$M a	8	2	
27 li\$lodna\$Madhura#Dehri\$Majaura\$Deg h	18	3	
28 lodna\$Madhura#Dehri\$Majaura\$Deghli \$	21	4	
29 na\$Madhura#Dehri\$Majaura\$Deghli\$ l	24	4	
30 odna\$Madhura#Dehri\$Majaura\$Deghli \$	22	4	
31 ra#Dehri\$Majaura\$Deghli\$lodna\$Madh u	32	5	
32 ra\$Deghli\$lodna\$Madhura#Dehri\$Maja u	11	2	
33 ri\$Majaura\$Deghli\$lodna\$Madhura#De h	3	1	
34 ura#Dehri\$Majaura\$Deghli\$lodna\$Mad h	31	5	l ₂
35 ura\$Deghli\$lodna\$Madhura#Dehri\$Maj a	10	2	h ₂

Figure 2: BWTPA data structure.

5 2 4 1 3 3 1 5 2 5 2 4 5 2 2 5 4 3 1 3 3 1 5 1 3
2 3 4 4 4 5 2 1 5 2.

We can see that $PA[0] = 5$, which means $SA[0] = 34$ belong to the 5th word. Fig. 2 shows a example of BWTPA data structure. Note that we only need L and PA .

3.2 A BWTPA Based Approximate String Search Framework

Our approximate string search algorithm is based on a partition approach. Suppose we have a pattern P and a threshold τ . We decompose P to $\tau + 1$ chunks \mathcal{S} . Then if P matches the text with at most τ errors, at least one of the parts will match a substring of the text exactly. We can use this feature to prune many dissimilar strings. As we know, short pattern usually appears with higher probability than long one, so we segment P to the same length L , where $L = \frac{|P|}{\tau+1}$. If there is a remainder $r = |P| \% (\tau + 1)$, we calculate the length of i th segment as Equation 3.

$$L_i = \begin{cases} L + 1 & \text{if } 1 \leq i \leq r, \\ L & \text{if } r + 1 \leq i \leq \tau + 1. \end{cases} \quad (3)$$

As we know, if pattern P is similar to a string T_i in \mathcal{C} , a segment of P should match a substring of T_i exactly. We take the advantage of BWT backward search algorithm to do the job. Then, we can access the PA array to get the candidate string id i , and verify it to get the answer.

We depict the basic process in Algorithm 1. We first segment the pattern P to $\tau + 1$ segments (line 1). Then we

Algorithm 1: Searching Pattern using BWTPA

Input: BWT L , Position array PA , Pattern P , Threshold τ

Output: $\mathcal{A} = \{T_i \in \mathcal{C} \mid ed(P, T_i) \leq \tau\}$

```

1  $\mathcal{P} = \text{segment}(P, \tau + 1)$ ;
2 foreach  $element\ s \in \mathcal{P}$  do
3    $pair\langle l, h \rangle = \text{backwardSearch}(L, s)$ ;
4   if  $l \leq h$  then
5     for  $j \leftarrow l$  to  $h$  do
6        $Candidate \leftarrow Candidate \cup T_{PA[j]}$ ;
7  $\mathcal{A} \leftarrow \text{verify}(Candidate, P, \tau)$ ;
8 return  $\mathcal{A}$ ;
```

Procedure $\text{segment}(P, n)$

Input: Pattern P , Segment number n

Output: $\mathcal{P} = \{S_1 S_2 \dots S_n\}$

```

1  $L = |P|/n$ ;
2  $r \leftarrow |P| \% n$ ;
3 for  $i \leftarrow 1$  to  $r$  do
4    $P \leftarrow L + 1$ 
5 for  $i \leftarrow r + 1$  to  $n$  do
6    $P \leftarrow L$ 
```

Procedure $\text{backwardSearch}(L, S)$

Input: BWT L , String S

Output: $pair\langle l, h \rangle$

```

1  $l \leftarrow 1, h \leftarrow L.length$ ;
2 for  $i \leftarrow s.length$  to 1 do
3    $l \leftarrow C[S[i]] + Occ(L, S[i], l - 1) + 1$ ;
4    $h \leftarrow C[S[i]] + Occ(L, S[i], h)$ ;
5   if  $l > h$  then
6     break;
7 return  $pair\langle l, h \rangle$ ;
```

Procedure $\text{verify}(Candidate, P, \tau)$

Input: Candidate $Candidate$, Pattern P , Threshold τ

Output: $\mathcal{A} = \{T_i \in \mathcal{C} \mid ed(P, T_i) \leq \tau\}$

```

1 foreach  $element\ T_i \in Candidate$  do
2   if  $ed(T_i, P) \leq \tau$  then
3      $\mathcal{A} \leftarrow \mathcal{A} \cup i$ ;
4 return  $\mathcal{A}$ ;
```

search each segment in our BWTPA data structure to get the range of occurrences as depicted in line 3. We insert each PA value in the range to get candidate set (line 5- 6). We verify the candidate set to get the answer (line 7).

For instance, consider string set in Figure 1(a). We search pattern *Madaura* with $\tau = 1$ as depicted in Figure 2. First, we segment the string *Madaura* to *Mada* and *ura*. We search *Mada* to get an empty range and search *ura* to get range $\langle 34, 35 \rangle$. Second, we can retrieve corresponding string id $\{2, 5\}$ as the candidate set. Third, we verify the candidate set to get the eventual result $\{2, 5\}$.

4. IMPROVING APPROXIMATE STRING SEARCH

In this section, we propose strategies to accelerate the basic approximate string search algorithm.

4.1 A Cache-aware Parallel Optimization Framework

Cache-aware Optimization: As we know, if the recent accessed data can be hit in cache can help to save a lot of time to access the memory. Figure 3 shows access time of different parts of hardware in the computer system. We can find that cache is smaller but faster than memory. In this section, we propose a cache-aware optimization framework to accelerate the approximate search process.

registers	0.3~0.5 ns
L1 Cache	
L2 Cache	1~10 ns
Memory	80~200 ns
Disk	10,000,000ns

Figure 3: Access time of different parts of hardware.

Our cache-aware optimization strategy is based on disjointed partition of the text collection \mathcal{C} . Consider cache size is M , we segment the \mathcal{C} to partitions $\{C_1, C_2 \dots C_n\}$, and build BWTPA index on each partition $\{B_1, B_2 \dots B_n\}$. We guarantee that $\text{sizeof}(B_i) \leq M$, for $1 \leq i \leq n$.

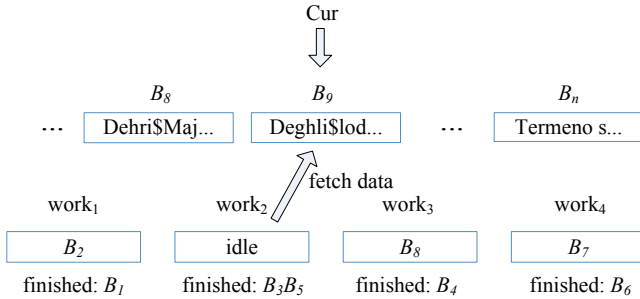


Figure 4: Assigning tasks to multicore.

Optimizing Approximate Processing Using a Multi-core Processor: To take advantage of the multicore resource, we utilize the multi threads technique. The straightforward assignment is to assign each work of a block to a core statically. However, as we can see that the work loads on different blocks could be different, assigning the tasks to different cores dynamically is more desired. We consider each thread as a worker in a factory. We adopt the pull model – a idle worker fetches the next block of index automatically. We maintain a counter to record the current block id to prevent the same task to be assigned to other workers. Since we segment the text collection \mathcal{C} without common strings,

each thread will not need to synchronize during the searching process. For long run search, every worker undertakes similar amount of work.

Figure 4 shows an example of assigning tasks to 4 workers. Worker₁ has finished the task of B_1 , it currently search on B_2 . worker₂ has finished two smaller tasks of B_3 and B_5 . Each worker has finished similar amount of work.

4.2 Pruning Techniques

Length Filtering: Consider two strings S_1, S_2 , if $\text{ed}(S_1, S_2) \leq \tau$, the length $|S_1|$ and $|S_2|$ must be within τ . With this property, We can prune many dissimilar strings. To make its utilities to further extent, we sort the strings in \mathcal{C} based on their lengths, and tag each block with a length range $[l_{min}, l_{max}]$. The possible length range of pattern P with τ is $[|P| - \tau, |P| + \tau]$. We search the block only if there is a common range between block tag $[l_{min}, l_{max}]$ and possible length range $[|P| - \tau, |P| + \tau]$. To accelerate the locating of the corresponding blocks, we use binary search algorithm.

For instance, if we search *Dehli* with $\tau = 1$ in Figure 1(a). The possible length range $[|P| - \tau, |P| + \tau] = [4, 6]$. The only possible block that can report answer is block 1. Therefore we can skip other blocks safely. Figure 5 shows the example of locating the possible blocks.

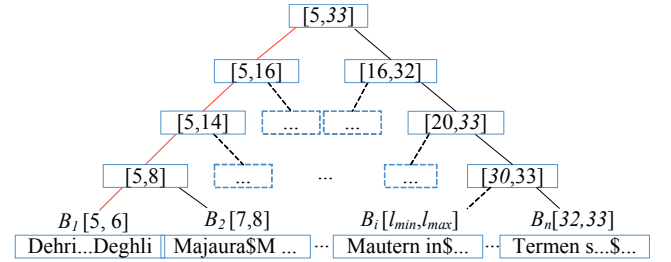


Figure 5: Sketch of locating a possible block.

Position Filtering: Position Filtering takes the advantage of position information to prune many dissimilar strings. Consider two strings S_1, S_2 , and $\text{ed}(S_1, S_2) \leq \tau$. If we partition the S_1 to $\tau + 1$ segments, there must be one of the segment s_i match a substring start at j position of S_2 , such that the start position of s_i and j must be within τ .

As we can see that the matched segment separate a string to three parts: prefix, matched segment, and suffix. The traditional position filter only considered of the prefix position, actually there is a symmetrical suffix position filter. In this section, we consider both the prefix and suffix position filter.

For instance, if we search *muradgy* with $\tau = 2$ in Figure 1(a). It will be partitioned to three segments *mur*, *ad* and *gy*. *Madhura* will be selected as a candidate, since the second segment *ad* can be found in *Madhura*, and the total length difference is 0. However we can also prune it as follows. If we match *ad* of the two strings, *Madhura* will be segmented to *M*, *ad*, and *hura*. We can see that $\text{ed}(M, \text{mur}) \geq 2$ and $\text{ed}(\text{gy}, \text{hura}) \geq 2$ based on length filtering, thus the total edit distance must be greater than or equal to 4. So *Madhura* can be safely pruned.

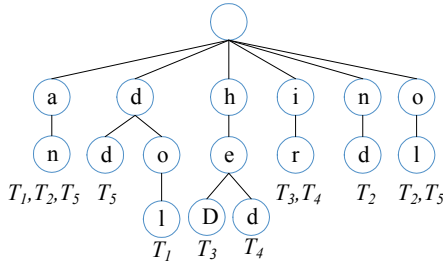
4.3 Optimizing Search Processing for Multiple Queries

In this section, we optimize the search processing for multiple queries. It is common that queries might share the same segments. Taking advantage of this property, we want to avoid the search processing for those duplicated segments of queries.

Consider that some patterns may share the same segment. We would like to search only once for these identical segments. Since we use the backward search algorithm of BWT, we need to reverse segments. By partitioning each string to $\tau + 1$ segments and inserting the reversed segment, we build a reversed segment trie to merge the identical segments.

ID	Strings	t
1	lodna	1
2	lodnna	2
3	Dehri	1
4	dehri	1
5	loddna	2

(a) Queries.



(b) Reverse segment trie.

Figure 6: Reverse segment trie on multi query.

For instance, we want to search the patterns in Figure 6(a). We start from the first string *lodna*. Since $\tau = 1$, we partition it to two segments *lod* and *na*. We insert the *dol* and *an* to the reverse segment trie. Then we process the other strings in the same way.

4.4 An Improved Look Ahead Verification Approach for Bounded ED

In this section, we focus on improving the verification step. Consider two strings S_1 and S_2 . A straightforward method to verify $\text{ed}(S_1, S_2) \leq \tau$ is to compute the actual edit distance of the two strings using dynamic programming. However, there is no need to compute the true edit distance of the two strings. There is a bounded edit distance algorithm only consider the cells within τ distance from the main diagonal[20]. The idea is that since each insert or delete a character will increase 1 to edit distance, so all the value of cells beyond τ distance from the main diagonal are large than τ . It can help us to improve the verification from $\mathcal{O}(|S_1| \times |S_2|)$ to $\mathcal{O}(\tau \times \min(|S_1|, |S_2|))^3$.

We take advantage of look ahead method to improve the bounded edit distance algorithm. When we come to current cell located at (i, j) on the edit distance matrix, we compute

³Bounded edit distance also improves the space from $\mathcal{O}(|S_1| \times |S_2|)$ to $\mathcal{O}(\tau)$.

the current value V_{ij} , and look ahead to get the maximum possible edit distance $V_{max} = V_{ij} + \max(|S_1|, |S_2|)$ and minimum possible edit distance $V_{min} = V_{ij} + ||S_1| - |S_2|| - i + j$. There are two cases to finish the computation early.

Case 1: If any $V_{max} \leq \tau$, we can stop the computation and report that $\text{ed}(S_1, S_2) \leq \tau$.

Case 2: If $V_{min} > \tau$ for all the columns, we can stop the computation and report that $\text{ed}(S_1, S_2) > \tau$.

Figure 7(a) shows the first case. there are two strings *similarly* and *similarly*. We want to check if $\tau \leq 3$, and we can see that there is a 0 in the shadow cell, and we look ahead the maximum possible error is 3. Therefore we can safely skip extra computation to be sure that $\text{ed}(\text{similarly}, \text{similarly}) \leq 3$. Figure 7(b) shows the second case. For two strings *reference* and *different*. We want to check if $\tau \leq 3$. We can find that only one cell of shadow column contains $V_{ij} \leq 3$. However, we look ahead the minimum possible error is 1. Sum the current error 3 and minimum look ahead error 1, we will get $\tau \geq 4$. Therefore we can safely skip extra computation to be sure that $\text{ed}(\text{reference}, \text{different}) > 3$.

		s i m i l a r l y									
		0	1	2	3	4	5	6	7	8	9
s		1	0	1	2	3	4	5	6	7	8
i		2	1	0	1	2	3	4	5	6	7
m		3	2	1	0	1	2	3	4	5	6
i		4	3	2	1	0	1	2	3	4	5
l		5	4	3	2	1	0	1	2	3	4
a		6	5	4	3	2	1	0	1	2	3
l		7	6	5	4	3	2	1	1	1	2
r		8	7	6	5	4	3	2	1	2	2
y		9	8	7	6	5	4	3	2	2	2

(a) Case 1.

		r e f e r e n c e									
		0	1	2	3	4	5	6	7	8	9
d		1	1	2	3	4	5	6	7	8	9
i		2	2	2	3	4	5	6	7	8	9
f		3	3	3	3	2	3	4	5	6	7
f		4	4	4	3	3	4	5	6	7	8
e		5	5	4	4	3	4	4	5	6	7
r		6	5	5	5	4	3	4	5	6	7
e		7	6	5	6	5	4	3	4	5	6
n		8	7	6	6	6	5	4	3	4	5
t		9	8	7	7	7	6	5	4	4	5

(b) Case 2.

Figure 7: Bounded look ahead verification.

5. APPROXIMATE STRING JOIN ALGORITHMS

In this section, we focus on the approximate string join problem. The main difference between the approximate string search and join is that the query pattern is the same with text collection.

5.1 An Incremental Approximate String Join Algorithm

Consider two strings S_1 and S_2 . If $\text{ed}(S_1, S_2) \leq \tau$, we know that $\text{ed}(S_2, S_1) \leq \tau$ for sure. We can improve the join process by removing the symmetrical case. Instead of searching the whole text collection \mathcal{C} for each string, we search the subset, which contains smaller string ids than current string id. However, the strings in BWTPA index are unordered. Note that we have partitioned the text collection \mathcal{C} to small portions $\{\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_n\}$. Although BWTPA index is unordered, these portions do have order. We tag each portion \mathcal{C}_i with the first string id T_i^f in \mathcal{C}_i .

Algorithm 2 gives the pseudo-code of the incremental approximate join. We first construct a BWTPA index for all the strings in \mathcal{C} (line1). For all the strings, we use binary

search to locate the corresponding block, which is the first block contains some strings with length less than or equal to the current string's length minus τ (line 3). Then we segment the current string to $\tau + 1$ segments (line 4). The search process can be early terminated if we reach a block to have all strings' lengths \geq current string length plus τ , or the first string id is greater than current id (line 6). We backward search these blocks just as the approximate search algorithm to get candidates (line 8 - 12). We verify these candidates to get the result (line 14).

Algorithm 2: Incremental Approximate Join Algorithm using BWTPA

Input: Text collection \mathcal{C} , Threshold τ
Output: $\mathcal{A} = \{\langle T_i, T_j \rangle \in \mathcal{C} \bowtie \mathcal{C} \mid ed(T_i, T_j) \leq \tau\}$

```

1 BWTPA  $\leftarrow$  buildBWTPA( $\mathcal{C}$ );
2 foreach  $T_i$  in  $\mathcal{C}$  do
3    $k = \text{binarySearch}(l_{max}, |T_i| - \tau)$ ;
4    $\mathcal{P} = \text{segment}(T_i, \tau + 1)$ ;
5   while  $k < n$  do
6     if  $l_k^{min} > |T_i| + \tau \mid i < T_k^f$  then
7       break;
8     foreach element  $s \in \mathcal{P}$  do
9        $\text{pair}(l, h) = \text{backwardSearch}(L_k, s)$ ;
10      if  $l \leq h$  then
11        for  $j \leftarrow l$  to  $h$  do
12           $\text{Candidate} \leftarrow \text{Candidate} \cup T_{PA[j]}^k$ ;
13       $k++$ ;
14    $\mathcal{A} \leftarrow \text{verify}(\text{Candidate}, T_i, \tau)$ ;
15 return  $\mathcal{A}$ ;
```

5.2 A Trie-based Approximate String Join Algorithm

Incremental approximate string join algorithm has to backward search every string in the text collocation \mathcal{C} , which is inefficient when the string collection is big. However we do not need to computer these strings one by one. Instead we build a trie for the reversed segments of these strings. Backward searching all these strings as a whole string collection can help us to avoid unnecessary repetitive computations. Figure 8 shows the reversed segment trie based on Figure 1(a). We segment each string in \mathcal{C} , and insert the reversed segments to trie. We add the id of these strings to the leaf nodes of the trie.

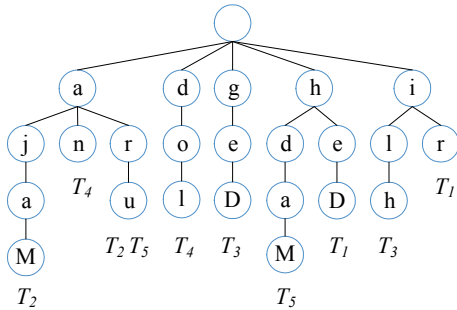


Figure 8: Reversed segment trie on text.

Algorithm 3: Trie-based Approximate Join Algorithm using BWTPA

Input: Text collection \mathcal{C} , Threshold τ
Output: $\mathcal{A} = \{\langle T_i, T_j \rangle \in \mathcal{C} \bowtie \mathcal{C} \mid ed(T_i, T_j) \leq \tau\}$

```

1 BWTPA  $\leftarrow$  buildBWTPA( $\mathcal{C}$ );
2  $\mathcal{T} \leftarrow$  new Trie;
3 foreach  $T_i$  in  $\mathcal{C}$  do
4    $P = \text{segment}(T_i, \tau + 1)$ ;
5   foreach element  $s \in P$  do
6      $\text{revs} = \text{reverse}(s)$ ;
7      $\text{insert}(\mathcal{T}, \text{revs})$ ;
8  $\mathcal{S} \leftarrow$  new Stack;
9  $l \leftarrow 1$ ;
10  $h \leftarrow L.\text{length}$ ;
11 Let  $r$  denote the root of Trie  $\mathcal{T}$ ;
12  $\mathcal{S}.\text{push}(r)$ ;
13  $p \leftarrow r.\text{firstchild}$ ;
14 while  $\mathcal{S}$  is not empty do
15   while  $p$  is not null do
16      $p \leftarrow \mathcal{S}.\text{top}$ ;
17      $l \leftarrow C[c] + \text{Occ}(L, c, l - 1) + 1$ ;
18      $h \leftarrow C[c] + \text{Occ}(L, c, h)$ ;
19     if  $p$  is a leaf node then
20       for  $j \leftarrow l$  to  $h$  do
21          $\text{Candidate} \leftarrow \text{Candidate} \cup \langle p.\text{id}, PA[j] \rangle$ ;
22      $\mathcal{S}.\text{push}(\langle p, l, h \rangle)$ ;
23      $c \leftarrow p.\text{firstchild}$ ;
24    $\langle p, l, h \rangle = \mathcal{S}.\text{pop}()$ ;
25    $p \leftarrow p.\text{nextsibling}$ ;
26  $\text{verify}'(\text{Candidate}, \tau)$ ;
27 return  $\mathcal{A}$ ;
```

Algorithm 3 gives the pseudo-code of the trie-based approximate join. Different from the incremental method, the trie-based method builds a trie initially, and inserts all the reversed segments of the strings in \mathcal{C} to trie (line 2 - 7). We use a stack to depth-first traverse of the trie. Initially, we set a range $\langle l, h \rangle$ to cover the whole text collection \mathcal{C} (line 9 - 10). When reach a node with character c , we use c to update the old range $\langle l, h \rangle$ (line 17 - 18). When reach a leaf node, we can get the candidate pair, which is the current node id and corresponding string id based on position array (line 21). Then we push the triple $\langle p, l, h \rangle$ to the stack, and verify all the candidate pairs to get the answer (line 26).

5.3 Pruning techniques

All the pruning methods given in Section 4.2 can be combined to the incremental method and the trie-based method. We increase a new count filter based on self join.

Count Filtering: Each string is the same to itself, so we do not need to verify the pair with identical string id. Consider a range $\langle l, h \rangle$, l is the position of the first occurrence of the substring based on lexicographical order, and h is the position of the last occurrence of the substring. When l equals to h , we can safely skip the current branch, since there is only one string in the range, which must be the same string.

6. EXPERIMENTS

We have implemented our method and conducted experimental studies on three real data sets:

- (1) Geographical name data, which contains 400,000 names of cities from all over the world⁴.
- (2) DBLP author data, which contains 612,781 author's name from the DBLP computer science bibliography⁵.
- (3) Human genome read data, which contains 750,000 genome reads of human.

Table 1 shows the detail information of the data sets. Figure 9 gives the string length distributions of the data sets.

Table 1: Data sets.

Data sets	count	Avg Len	Max Len	Min Len
Geonames	400,000	10.106	60	1
Authors	613,542	13.815	46	4
reads	750,000	100.388	106	86

We have implemented all the methods in this paper. All the algorithms were implemented in C++ and compiled with G++ 4.4.5 with -O3 flags. All the experiments were run on a machine with 2.93 GHz Intel Core CPU, 4 GB main memory, and Ubuntu operating system (Linux distribution).

6.1 Evaluate similarity search

In this section, we evaluate similarity search algorithms. We implemented the following five methods as follows.

- (1) **Basic**, which is the basic search algorithm using B-WTPA.
- (2) **Cache**, which is the Cache-aware parallel algorithm as discussed in Section 4.1.
- (3) **Prune**, which combines the pruning techniques in Section 4.2.
- (4) **Multi**, which takes the advantage of multi-query optimization, as given in section 4.3.
- (5) **Bound**, which improves the edit distance with bounded look ahead optimization, as depicted in section 4.4.

We randomly extract 20%, 40%, 60%, 80%, and 100% strings of the data sets as the text collections, and randomly select 5000 strings in the text collections as query. For geographical names and author names we randomly select $0, \dots, 4$ as threshold τ , for genome reads, we select $0, \dots, 16$ as threshold. Note that $\tau = 0$ is the same as searching exact pattern.

Figure 10 compares the performance of the five search algorithms on the three data sets. Figures 10(a) and 10(c) show the performance of search algorithms varied with different data size. We can see that basic method is the most time-consuming one. For 20% geographical names, it took 27.71 seconds, and for 100% geographical names it took 130.65 seconds. The cache method took only about half

of the basic method, which took 15.09 seconds, and 74.27 seconds for the two corresponding data sizes. The Prune method further reduced the time to 11.22 seconds and 64.56 seconds. The multi method reduced the cost to 10.72 seconds and 55.85 seconds. The bounded method is the fastest one, which took 8.33 seconds for 20% data size, and 40.3 seconds for 100% data size. We can find similar results for author names and genome reads.

Figures 10(d) and 10(f) show the performance of search algorithms on the whole data set varied with different thresholds. We set the threshold τ of each query to $0, \dots, 4$, respectively, for geographical names and author names. For genome reads, we set threshold τ to $0, 4, 8, 12$, and 16 . We see that the basic method had the worst performance for $\tau = 0$, it took 1.99 seconds, and for $\tau = 4$, it took 511.68 seconds. The cache method took 1.03 seconds and 270.77 seconds with the corresponding τ . The Prune method improved the time to 0.93 seconds and 230.55 seconds. The multi method further reduced the time to 0.75 seconds and 184.65 seconds. The bounded method is the fastest one. It took 0.561 seconds when $\tau = 0$, and 141.74 seconds when $\tau = 4$.

6.2 Evaluate similarity join

In this section, we evaluate similarity join algorithms. We implemented three methods as follows.

- (1) The incremental Approximate string join algorithm, which is denoted by Incremental.
- (2) The trie-based string join algorithm, which is denoted by Trie-based.
- (3) Prune method combines the prune techniques with the trie-based method.

First we compare the performance of join algorithms varied with different data sizes. We set $\tau = 2$. Figure 11 shows the results. We see that the trie-based method is better than the incremental method. For 20% geographical names, the incremental method took 13.32 seconds, and for 100% names, it took 74.88 seconds. The trie-based method only took 11.69 seconds and 59.71 seconds for corresponding data sizes. The reason is that trie-based method saves the unnecessary computation of the identical segments. The prune method further improved the performance to 7.93 seconds and 42.6 seconds.

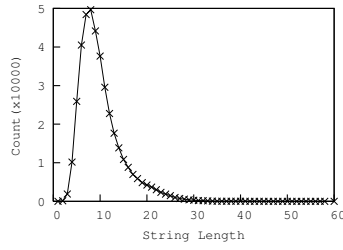
Figures 11(d) and 11(f) compare the performance of join algorithms using different thresholds. With the increase of the threshold, the time costs of all the methods increase. The prune method outperformed the trie-based method which in turns was better than the incremental method.

7. CONCLUSION

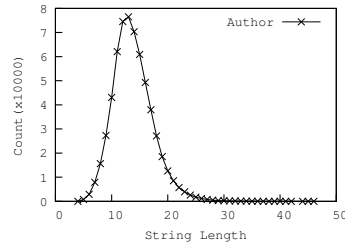
In this paper, we have studied the similarity search and similarity join problem with edit distance constraints. We proposed a cache-aware parallel method using BWT to find similar strings in text collection. We devised a new index called BWTPA. We used BWTPA to index strings, and we partitioned queries to segments. We developed an algorithm to search these segments to find candidates efficiently and verified the candidates to get the results. We also developed several optimization strategies to improve the performance.

⁴<http://www2.informatik.hu-berlin.de/~wandelt/searchjoincompetition2013/Dates.html>

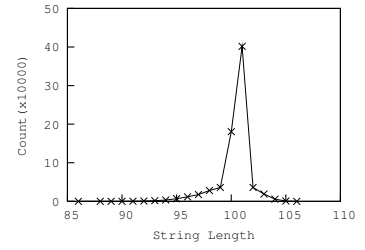
⁵<http://www.informatik.uni-trier.de/~ley/db/>



(a) Geographical names.

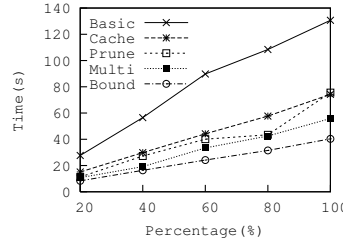


(b) DBLP Authors.

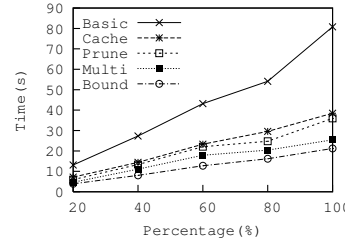


(c) Genome reads.

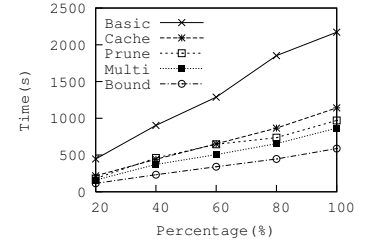
Figure 9: String length distributions.



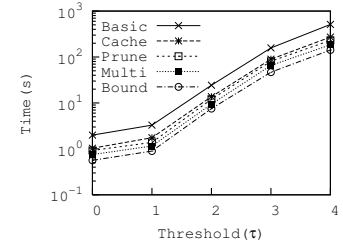
(a) Geographical names.



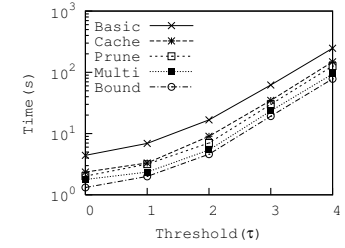
(b) DBLP Authors.



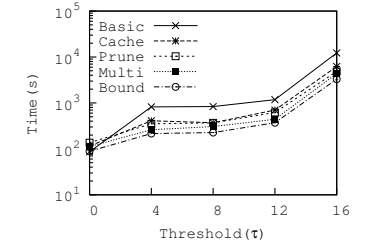
(c) Genome reads.



(d) Geographical names.

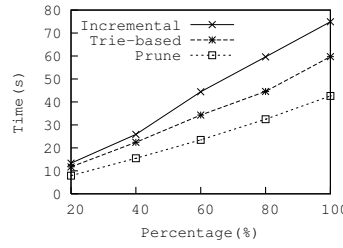


(e) DBLP Authors.

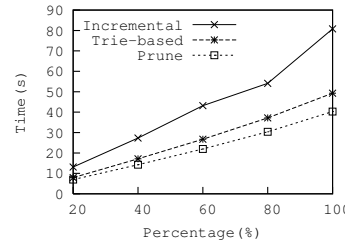


(f) Genome reads.

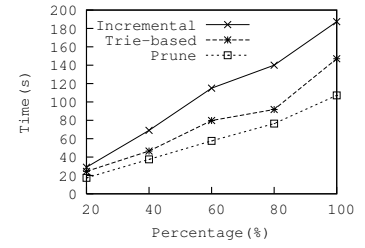
Figure 10: Performance of similarity search.



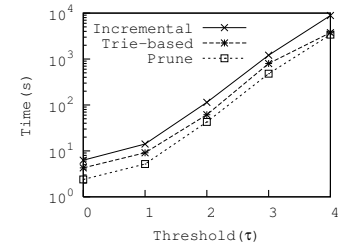
(a) Geographical names.



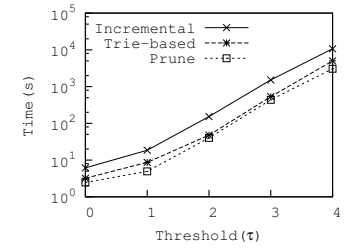
(b) DBLP Authors.



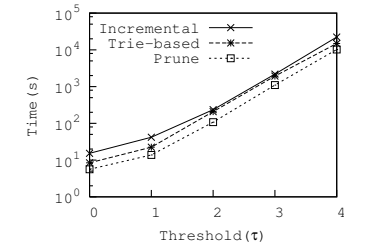
(c) Genome reads.



(d) Geographical names.



(e) DBLP Authors.



(f) Genome reads.

Figure 11: Performance of similarity join.

We proposed two methods to do similarity join on the index. We have implemented our algorithms, and experiments showed the efficiency of our method.

8. ACKNOWLEDGMENTS

The work is partially supported by the National Basic Research Program of China (973 Program) (No. 2012CB316201), the National Natural Science Foundation of China (Nos. 61129002, 61272178), the Doctoral Fund of Ministry of Education of China (No. 20110042110028) and the Fundamental Research Funds for the Central Universities (Nos. N110804002, N110404015).

9. REFERENCES

- [1] Walter A. Burkhard and Robert M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.
- [2] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311–321, 1993.
- [3] Paolo Ciaccia, A. Nanni, and Marco Patella. A query-sensitive cost model for similarity queries with m-tree. In *Australasian Database Conference*, pages 65–76, 1999.
- [4] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
- [5] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.
- [6] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
- [7] Chen Li, Bin Wang, and Xiaochun Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.
- [8] Xiaochun Yang, Bin Wang, and Chen Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD Conference*, pages 353–364, 2008.
- [9] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.
- [10] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD Conference*, pages 1033–1044, 2011.
- [11] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.
- [12] Jianhua Feng, Jiannan Wang, and Guoliang Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.
- [13] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398, 2000.
- [14] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
- [15] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.
- [16] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.
- [17] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In Amihood Amir, Andrew Turpin, and Alistair Moffat, editors, *SPIRE*, volume 5280 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2008.
- [18] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.
- [19] Ruiqiang Li, Chang Yu, Yingrui Li, Tak Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.
- [20] Esko Ukkonen. On approximate string matching. In *FCI*, pages 487–495, 1983.