A MapReduce-based Quick Search Approach on Large Files

Ye-feng Li[1], Jia-jin Le[2], and Mei Wang[2]
[1]College of Computer Science and Technology, Beijing University of Technology, China
[2]College of Computer Science and Technology, Donghua University, China

Abstract: *String search is an important branch of pattern matching for information retrieval in various fields. In the past four decades, the research importance has been attached on skipping more unnecessary characters to improve the search performance, and never taken into consideration on large scale of data. In this paper, two major achievements are contributed. At first, we propose a Quick Search algorithm for data Stream (QSS) on a single machine to support string search in a large text file, as opposed to previous researches that limits to a bound memory. For the next, we implement the search algorithm on MapReduce framework to improve the velocity of retrieving the search results. The experiments demonstrate that our approach is fast and effective for large files.*

Keywords: *String search, mapreduce, data stream and large file*.

Received May 21, 2015; accepted September 24, 2017

1. Introduction

String search technology is widely applied in various fields, such as finding keywords in documents, locating variables or functions in source codes, and intrusion detection in security management systems [8], etc., It can also be used on bio-chemical researches like string matching in Domain Name System (DNS) sequence [7]. Currently it is still a hot topic not only in computer science but also in many other subjects.

The goal of a string search problem can be expressed as follows: Given the input text $T = T[0,...,n-1]$ of length n and the specified pattern $P = P[0,...,m-1]$ of length m over an alphabet Σ, find out all the positions of the matched strings inside T. Algorithms derived and extended from this descriptive model are studied in [6, 21].

In the 1970s, two of the most notable pattern matching algorithms were born: the Kunth-Morris-Pratt (KMP) [17] and the Boyer-Moore (BM) [5]. Both algorithms match the pattern and the text by skipping characters that are not likely to result in exact matching with the pattern. They require $O(|\Sigma|+m)$ time for pre-processing the pattern and O(mn) as the worst time or $O(\frac{n}{m})$ as the best time for searching. Since then, many variants of the two algorithms have been proposed to optimize the worst and best searching time [11, 21]. For instance, the Franek-Jennings-Symth (FJS) algorithm [12] reaches O(n) as the worst time for searching.

By investigating and making comparison on KMP, BM and all their variants, we can find that the performance of those algorithms are greatly affected by the length of the input text T, i.e., n. For the first, the algorithms run at a linear time to n, whatever on a worst searching time of O(n) or a best searching time $O(\frac{n}{m})$. In the second place, the algorithms assume that the search operation on the text T is done in the memory, so that one or more pointers are defined to move forwards and backwards on the characters of T without any restrict. However, if n grows up to Giga-Bytes (GB) or even Tera-Bytes (TB) level, T might have to be retrieved from a large text file through data stream. As only one direction is permit, the key steps in search operation of the algorithms would be different.

We introduce MapReduce framework [10] to reduce the string searching time on large text files. MapReduce is programming model for processing huge datasets using a number of machines usually in a cluster. A MapReduce job starts with partitioning the input file into several smaller “splits” and distributing them on corresponding machines. Then the operations are performed in parallel on their own machine. Finally, the results from each machine are collected, sorted and organized for output.

This research includes two major contributions:

1. We propose the Quick Search for data Stream (QSS) algorithm to support quick search on data stream as an extension of Sunday’s Quick Search (QS) algorithm [22]. The search operation is pushed as the data stream flows until the end of file is reached, without knowing the text length n beforehand. The complexity of QSS does not come down and remains to be the same as that of QS.

2. On such basis, we apply the QSS algorithm on MapReduce framework, using a complementary strategy to avoid the situation that the matched string resides in two splits. The experiments demonstrate the algorithm runs mush faster compared to that on a single machine.

The rest of paper contains five more sections. Section 2 describes the related work of string search algorithms. Section 3 illustrates the QSS algorithm with an example of search operation on data stream. Section 4 elaborates the QSS implementation on MapReduce framework. Section 5 proves the excellent performance of the research using comparative experiments. Section 6 gives the conclusion and future work.

2. Related Works

In this section, we describe the states of art of pattern matching algorithms on a single machine. Such research has never been done for data stream or even in a distributed environment so far.

There are many algorithms proposed for pattern matching since KMP and BM came out to the world. Most of them are usually classified into three categories [11]: forward orientation, backward orientation and no specific direction. In forward orientation, the text is compared to the pattern from left to right, where KMP algorithm is the candidate. Apostolico and Crochemore's research are also in this category [3, 9]. In backward orientation, the text is compared to the pattern from right to left, with the BM algorithm as its candidate. In the third category, algorithms use both directions for comparison at the same time, such as the Horspool algorithm [15], Sunday's quick search algorithm [22], the FJS algorithm [12], Lin's Faster Quick Search (FQS) algorithm [20] and Al-Ssulami's hybrid string matching algorithm [2]. See Charras and Lecroq's book [6] for more similar algorithms.

Besides, there are pattern matching algorithms not based on character comparison. Karp and Rabin proposed a randomized algorithm using a hashing function to represent longer strings to shorter "fingerprints", which are manipulated to achieve the algorithm efficiency [16]. Baeza-Yates and Gonnet introduced a different pattern matching operation using bitwise operations such as "shift" and "and", where pattern is represented by a binary mask [4]. On this basis, Fredrikkson and Grabowski improved the approach to bitwise parallelism with the purpose of exact pattern matching [13]. Adjeroh etc. provided a survey of techniques for pattern matching in compressed text and images [1], listing algorithms suitable for pattern matching of various compression methods. Gagie etc. made use of the classical LZ77 algorithm, and designed a self-index for the string to support both random access and pattern matching queries [14]. Moreover, parallel techniques are used to make the matching process faster, such as multithreading approach by Kofahi and Abusalama [18] and and Graphics Processing Unit (GPU) usage by Kouzinopoulos *et al*. [19].

The above algorithms are required to be worked in a bounded resource, which results in a restriction on length of the string to be matched.

3. Quick Search for Data Stream

In this section, we illustrate the QSS algorithm, which is an extension of Sunday's QS algorithm for data stream. At first, we take a brief introduction on the QS algorithm. Then we use an example to show the process of QSS. In the end, we give out the pseudo code of QSS as implementation.

3.1. The Quick Search Algorithm

The QS algorithm was proposed by Sunday in 1990 [22]. It is a simplified BM algorithm using a bad-character shift array of length |Σ|. The array stores the position of rightmost occurrence of each letter c in *P*, defined as follows.

$$\Delta(\sigma)=\begin{cases}\min\left\{k : k\in(0,m]\middle| P[m-k]=\sigma\right\} & \text{if } \sigma\in\Sigma\\ m+1 & \text{if } \sigma\notin\Sigma\end{cases} \quad (1)$$

The QS algorithm contains two stages. The pre-processing stage constructs the Δ array, while the searching stage makes comparison of each substring in *T* and shifts the current pointer to a new position according to Δ. For instance, given *P* = "*abcace*", we will have *m* = 6 and Σ = {*a*, *b*, *c*, *e*} with Δ[*a*, *b*, *c*, *e*] = [1, 2, 3, 5].

Same to the BM algorithm, QS requires $O(|\Sigma|+m)$ time for pre-processing the pattern and O(*mn*) as the worst time or $O(\frac{n}{m})$ as the best time for searching. However, it is very simple and easy to implement, see Christian and Thierry's webpage (http://www-igm.univ-mlv.fr/~lecroq/string/index.html) for C code.

3.2. Search on Data Stream

In the new era of "big data", the conception of "Volume", "Velocity" and "Variety" is proposed as new requirements for large scales of data. However, traditional string matching algorithms could no longer come up with the new demands, and the reasons are listed as follows:

- *Volume*: The size of data grows rapidly so that the input text is usually stored in a large file rather than fit in a bound memory. Also the backward comparison is not suggested as it might produce additional disk I/O.
- *Velocity*: This attribute is related to Volume. As the complexity is linear to the text length, more

execution time is required as the volume of data grows.

- *Variety*: The character range should be extended from ASCII Standard to Unicode Standard to support string matching on multiple languages. In that case, |Σ| would possibly reach a maximum value of 65535 in case that all the characters are used. As one character might take 1-2 bytes, the length of input text is totally unknown.

In this section, we propose a new QSS for the purpose of "Volume" and "Variety", where the "Velocity" attribute will be handled in section 4. The QSS algorithm is an extension from QS. It also contains two stages and the pre-processing step is exactly the same as that of QS.

We use the following example to illustrate the search operation on data stream, given pattern P= "abcace". We use two char buffers, cbuf and tmp. The former is used to buffer the streaming data with the maximum of m characters, and the latter to accumulate the current position of offset to the head of file with the maximum size of m+1 characters.

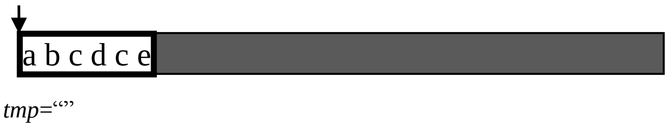

Figure 1. The initial state of the char buffer.

At the very beginning, the first *m* characters are read to the *cbuf*, shown as Figure 1. Obviously, we have a mismatch and nothing is output.

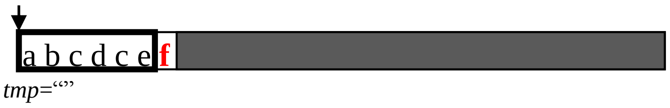

Figure 2. Read "f" into the char buffer.

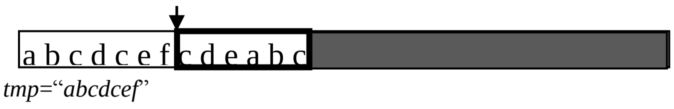

Figure 3. Read m more characters.

After that, one more character is read from the stream, as shown in Figure 2. Since "*f*" does not belong to Σ, we copy *cbuf* to *tmp*, adding the single character at the end of *tmp*, and then read *m* more characters to overwrite *cbuf*, as shown in Figure 3. Again, it is a mismatch from *P*. We count on the actual bytes in *tmp*, and move the file pointer forward.

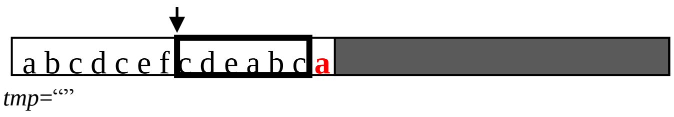

Figure 4. Read "a" into the char buffer.

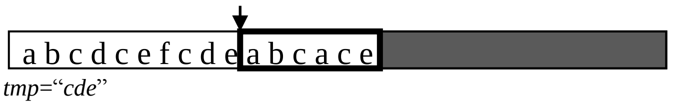

Figure 5. A match is found.

For the next, the similar steps are performed. We get a character "*a*" from the stream, and clear *tmp*, as shown in Figure 4. As Δ[*a*]=3, we copy the first 3 characters from *cbuf* to *tmp*, and shift the rest *m*-3 characters left-sided in *cbuf*. Then we append the character "*a*" to *cbuf* and read 3-1=2 more characters from the stream. Now we get a match from *P*, shown as Figure 5. The file pointer is accumulated by adding up the bytes in *tmp*, and output to the console.

The string search process continues repeatedly as illustrated above until reaching the End Of File (EOF), shown in Algorithm 1. The algorithm takes filename and start_pos of file as part of the input parameters.

Algorithm 1: QSS search stage

```
Input: filename, start_pos, P, Σ, Δ
 1.  is = new InputStream(filename);
 2.  pos = start_pos; is.seek(pos);
 3.  m=|P|;
 4.  char cbuf[0…m-1];
 5.  is.read(cbuf, 0, m);
 6.  While true
 7.     matched = compare(cbuf, P);
 8.     If matched == true  output(pos);  End If
 9.     char c = is.read();
10.     If c == EOF  Break;  End If
11      If c∈Σ
12.       delta =Δ[c];
13.       char tmp[0…delta-1];
14.       tmp[0…delta-1] ← cbuf[0…delta-1];
15.       cbuf[0…m-delta-1] ← cbuf[delta…m-1];
16.       cbuf[delta] ← c;
17.       is.read(cbuf, m-delta+1, delta-1);
18.       If EOF Break;  End If
19        acc = tmp.getBytes();
20.     Else
21.       char tmp[0…m];
22.       tmp[0…m-1] ← cbuf[0…m-1];
23.      tmp[m] ← c;
24.      is.read(cbuf, 0, m);
25.      If EOF  Break;  End If
26.      acc=tmp.getBytes();
27.    End If
28.    pos += acc;
29.  End While
```

Note that some programming languages support reading Unicode characters by calling specific functions, e.g., BufferedReader in Java (Line 9, 17, and 24). Different from QS, this algorithm outputs the offset position to the header of file (Line 8), rather than number of characters read. Moreover, QSS is flexible that another boundary such as *end_pos* of the input file might be defined to terminate the search operation in advance, regardless of the EOF symbol.

4. Implementation on MapReduce

As the *volume* of data grows, the *velocity* for achieving computational results needs to be improved. The MapReduce framework is designed to perform parallel computing on large data sets. It is capable of speeding up certain algorithms that each element is not globalized, such as QSS. A MapReduce job usually includes five phases.

- *Input*: The input data is divided into *M* smaller splits in a user-specific approach, and distributed to certain machines in a cluster.
- *Map*: A map() function is invoked to retrieve data from each split, and writes the computational results in the form of <*key*, *value*> pairs to a temporary storage.
- *Shuffle*: The intermediate results are sorted and partitioned on key through a hash function (e.g., *key* mod *R*).
- *Reduce*: A reduce() function is invoked to retrieve data from the temporary storage with values grouped by key. The computational results are emit in the form of another <*key*, *value*> format for final output.
- *Output*: *R* files are output in a user-specified format.

Most MapReduce jobs perform the computation or comparison in *Map* stage or *Reduce* stage. However, in our approach, the string search operation is done in the Input stage, while the Map stage is only used to pass the result. Moreover, we skip *Shuffle* and *Reduce* stage to *Output* directly to save the cost from sorting and network transfer.

4.1. Splitting the Input Data

The Input phase of a MapReduce job is controlled by the file input format class and RecordReader class or their user-defined extensions. The get Splits() function in extended File Input Format class creates input splits in a certain way. Users can specify the size of each split to control the number of map tasks as *M*. The extended RecordReader class is used to read the content from each split, usually through data stream, and pass the information under current file pointer to *Map* phase in the form of <*key*, *value*> pairs.

About to partition the input data into splits, we face with the problem that a string that matches the pattern might be placed in two splits. Figure 6 shows an example of such case. Given *P*= “*abcace*”, the characters “*abca*” appears at the end of Split 1, and “*ce*” appears at the beginning of Split 2.

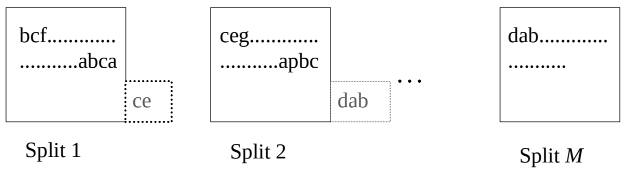

Figure 6. A string in two splits and the complementary strategy.

Although the probability of such cases is extremely tiny especially for a long pattern, we have to take care of it to retrieve exact and accurate results. Thus we propose the complementary strategy to deal with this problem. We read a few (no more than *m*) more characters in all the splits but the last to make the string complete.

As the MapReduce framework supports reading a large file through the same data stream, no matter how many splits produced and where they reside in, we just make extensions on the RecordReader class by applying the QSS algorithm on certain functions. The getSplit() function remains unchanged unless the size of each split should be modified. The main procedure of the RecordReader class is shown in Algorithm 2.

```
Algorithm 2: class QSRecordReader

Input: split, start_pos, split_len, P, Σ
Function initialization()
{
 1. Δ ← pre-processing(P, Σ);
 2. end = start_pos + split_len;
 3.  filename = split.getFilePath();
 4. Line 1-5 from Algorithm 1
}

Function nextKeyValue()
{
 1. While pos < end
 2.   key.set(pos); value.set(null);
 3.   matched = compare(cbuf, P);
 4.   Line 9-28 from Algorithm 1
 5.    If matched == true
          /* Emit <key, value> pair to Map phase */
 6.      Return true;
 7.    End If
 8.   End While
      /* Finish reading the split */
 9.  Return false;
   }
```

We need to explain some points about Algorithm 2:

1. The pre-processing stage from QSS is now placed in the initialization function. It might also be in the getSplits() function before creating splits, so that the piece of code would be only executed once. However, the produced Δ array has to be distributed to each split for further process, resulting in additional network transfer. Thus we choose to do it locally in each split.
2. The nextKeyValue() function is used to read the contents of the current split. If it returns true, the current <*key*, *value*> will be emit to the *Map* phase, where *key* is an integer variable representing the current offset to the head of input file and value is null. Otherwise, it stops reading and the job is ready for the *Map* phase.
3. Although the size of each split is explicitly defined, we can read bytes across the end as mentioned in the previous paragraph. The additional bytes read in each split would be *pos* - *end* in case of *pos* > *end*. Also, only the reading in the last split can meet with the EOF.

4.2. Collecting the Results

As the Input phase takes the charge of string search operation, the others are invoked to collect the results for the output files.

It is not surprisingly that the map() function in the Map phase is so simple that it has only one line of code to pass the keys directly to the Output phase. As the keys, i.e., positions of the file pointer, are received in an ascending order, it is not necessary to enter the Shuffle phase for sort, neither the Reduce phase to group the same keys as all the keys are different.

We just use the default text output format class in the Output phase. Then the MapReduce job produces *M* output files, each of which contains positions in ascending order. Finally, we merge all the files into a single one by the appending operation file to file, which takes only a few seconds.

5. Experiments

In this section, we conduct some experiments with two public datasets. The cluster consists of 1 NameNode and 6 DataNodes. Each machine has Intel QuadCore CPU, 8GB RAM and 450GB SCSI disk, with CentOS 6.4 system and Hadoop 2.6.0 [23] as the basis of MapReduce framework.

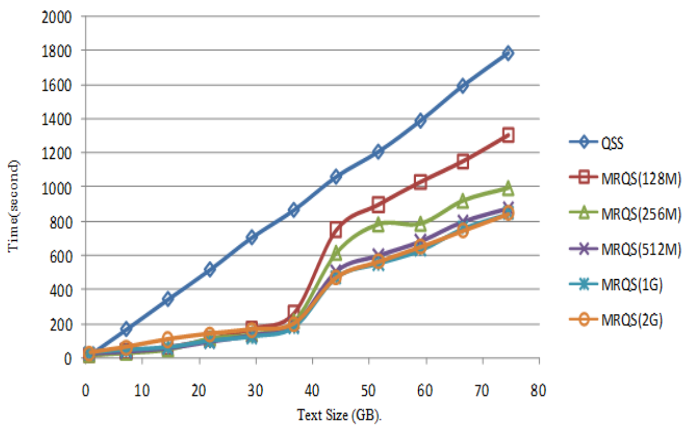

Figure 7. Time comparison on QSS and different split size of MRQS.

Table 1. Statistics of pattern search result using MRQS.

Pattern	**Length**	**Occurrence**	**Execution time of different split size (seconds)**				
			512M	**1G**	**2G**	**4G**	**8G**
阿拉伯	3	3273817	2175.698	2111.416	2157.683	2289.663	2370.372
模式匹配	4	3877	1993.313	1964.504	1943.47	2179.774	2348.804
中华人民共和国	7	10304775	1838.381	1797.676	1850.357	1924.036	2136.473
MapReduce	9	4854	1789.886	1808.724	1837.99	1876.242	2120.342
1945 年 8 月 15 日	10	28657	1788.845	1734.017	1838.595	1856.964	1941.259
好人人人人人人人人人人人人人人人人人人人	20	0	1666.224	1687.979	1732.772	1794.141	1942.266

5.1. TPC-H Benchmark

TPC-H is a decision support benchmark consisting of a suite of business oriented ad-hoc queries and concurrent data modifications. It includes two fact tables and six dimension tables Most tables have a Scale Factor (SF) so that users can alternate the size the performance of different split size by modifying the getSplits () function in the InputFormat class. The pattern we used is “TRUCK”, which is a frequently appeared value in the SHIPMODE column of Lineitem table. Figure 7 shows the comparison results of QSS on a single machine and its MapReduce implementation (MRQS) on the cluster with different split size.

The QSS algorithm performs quite well on small datasets. When SF=1, the text size is around 707 MB and the search process costs only 18.343 seconds. However, as the volume of data grows, it takes great time to find out all the places that match the pattern, about 30 minutes when SF reaches 100.

The MapReduce approach achieves shorter time for large datasets, but the split size is another factor that affects the performance. Given a fixed data size, bigger split size results in fewer number of map tasks in the job. According to the scheduling mechanism of the framework, a map task requires a few seconds to be launched before starting, and another few seconds to be terminated. On the other hand, if the split size is bigger than the configured Hadoop Distributed File System (HDFS) block size, one split might contain blocks residing on different machines, resulting in additional network transfer.

Obviously the default configured split size of 128MB is the worst as the data volume grows. The split size of 2GB does not perform well enough when the input text is less than 30GB, but becomes much better when the volume increasing to 60GB. The split size of 1GB appears to be the best in most situations of the range.

5.2. Wikipedia Database Dump

Wikipedia provides revision histories of articles in the form of database dump files. In this experiment, we pick up a document containing Chinese articles in all the pages to verify supporting of Unicode characters, available at http://dumps.wikimedia.org/zhwiki/. The document file is in xml format with the size of 456GB after decompression. We pick up some patterns of different length, calculating their occurrences in the document and making comparison on the execution time in different split size. The statistics listed in Table 1 reveals the facts as follows.

1. It proves the fact that MRQS becomes slower when the split size is set to 4G or 8G. We can assume that the values around 1G can achieve the optimal performance as they might leverage the cost balance between the number of map tasks and network transfer from other machines to the best extent.
2. Both the length and occurrence of pattern would affect the search speed. For instance, the 3-length pattern “阿拉伯” (Arab) with occurrence over three

million is slower than the 4-length pattern "模式匹配" (Pattern Matching) with only 3877 occurrence, but the 7-length pattern "中华人民共和国" (PR China) with more than 10 million occurrence is even faster. As a result, longer pattern length and fewer occurrences would take less time.

3. The last example shows an extreme case style of pattern as "ab^{m-1}" and $|\Sigma|$ =2. With $\Delta[b]$ =1, only one character is shifted in each time meeting with a character "*b*". However, the performance is still satisfactory and cost the least time compared with the previous examples. It can also be inferred that the alphabet size $|\Sigma|$ does not much affect the search time.

Even though the performance is actually limited by the scale of cluster, the experiments have proved the validity and effectiveness on large text files. Given more machines will absolutely speed up the velocity of search process. Also the support of Unicode characters is enabled to perform string search on documents written in other languages, such as Arabian text.

6. Conclusions

As the data volume grows, previous string search approaches could not work in a bound memory any more. Also the execution speed of such algorithms becomes a great challenge. This paper is the first study of string search on data stream, combining with the MapReduce framework to reduce the process time on large files. The experiments demonstrate that the proposed QSS algorithm is effectively to perform string search in a text file and the MapReduce implementation is able to speed up the velocity to a certain extent. Restricted by the experimental environment, although the improvement of search time is limited, it can be seen a better scope in a large cluster.

There is some more progress to be made in the future. Since the QSS algorithm is not optimal, we will try to reduce the complexity by combining with other proper string search approaches. Besides, we will evaluate the performance on distributed computational frameworks other than MapReduce to seek a more suitable one for string search algorithms.

Acknowledgements

This paper is partly a result of technology development project supported by the Technology Bureau Project of Zhengzhou City under the grant No.153PKJGG115.

References

[1] Adjeroh D., Bell T., and Mukherjee A., *Pattern Matching in Compressed Texts and Images*, Now Publishers Incorporated, 2013.

[2] Al-Ssulami A., "Hybrid String Matching Algorithm with A Pivot," *Journal of Information Science*, vol. 41, no. 1, pp. 82-88, 2015.

[3] Apostolico A. and Crochemore M., "Optimal Canonization of all Substrings of a String," *Information and Computation*, vol. 95, no. 1, pp. 76-95, 1991.

[4] Baeza-Yates R. and Gonnet G., "A New Approach to Text Searching," *Communications of the ACM*, vol. 35, no. 10, pp. 74-82, 1992.

[5] Boyer R. and Moore J., "A Fast String Searching Algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762-772, 1977.

[6] Charras C. and Lecroq T., *Handbook of Exact String Matching Algorithms*, Colleague Publications, 2004.

[7] Chen L., Cheung D., and Yiu S., "Approximate String Matching in DNA Sequences," *in Proceedings of 8th International Conference on Database Systems for Advanced Applications*, Kyoto, pp. 303-310, 2003.

[8] Coit C., Staniford S., and McAlerney J., "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *in Proceedings of DARPA Information Survivability Conference and Exposition*, Anaheim, pp. 367-373, 2001.

[9] Crochemore M. and Rytter W., *Text Algorithms*, Oxford University Press, 1994.

[10] Dean J. and Ghemawat S., "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, 2008.

[11] Faro S. and Lecroq T., "The Exact Online String Matching Problem: A Review of the Most Recent Results," *ACM Computing Surveys (CSUR)*, vol. 45, no. 2, pp. 1-42, 2013.

[12] Franek F., Jennings C., and Smyth W., "A simple fast Hybrid Pattern-Matching Algorithm," *in Proceedings of Annual Symposium on Combinatorial Pattern Matching*, Jeju Island, pp. 288-297, 2005.

[13] Fredriksson K. and Grabowski S., "Average-Optimal String Matching," *Journal of Discrete Algorithms*, vol. 7, no. 4, pp. 579-594, 2009.

[14] Gagie T., Gawrychowski P., Kärkkäinen J., Nekrich Y., and Puglisi S., "LZ77-based Self-Indexing with Faster Pattern Matching," *in Proceedings of Latin American Symposium on Theoretical Informatics*, Montevideo, pp. 731-742, 2014.

[15] Horspool R., "Practical Fast Search in Strings," *Software: Practice and Experience*, vol. 10, no. 6, pp. 501-506, 1980.

[16] Karp R. and Rabin M., "Efficient Randomized Pattern-Matching Algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249-260, 1987.

[17] Knuth D., Morris J., and Pratt V., "Fast Pattern

Matching in Strings," *SIAM Journal of Computing*, vol. 6, no. 2, pp. 323-350, 1977.

[18] Kofahi N. and Abusalama A., "A Framework for Distributed Pattern Matching Based on Multithreading," *The International Arab Journal of Information Technology*, vol. 9, no. 1, pp. 30-38, 2012.

[19] Kouzinopoulos C., Michailidis P., and Margaritis K., "Multiple String Matching on A GPU Using Cudas," *Scalable Computing: Practice and Experience*, vol. 16, no. 2, pp. 121-138, 2015.

[20] Lin J., Adjeroh D., and Jiang Y., "A Faster Quick Search Algorithm," *Algorithms*, vol. 7, no. 2, pp. 253-275, 2014.

[21] Smyth W., *Computing Patterns in Strings*, Addison-Wesley, 2003.

[22] Sunday D., "A Very Fast Substring Search Algorithm," *Communications of the ACM*, vol. 33, no. 8, pp. 132-142, 1990.

[23] White T., *Hadoop: the Definitive Guide*, O'Reilly Media Inc., 2009.

Ye-feng Li received his Ph.D degree from Donghua University in 2015. Currently he is a postdoctoral researcher in the college of computer science in Beijing University of Technology. His research major focuses on the big-data processing and information security technologies.

Jia-jin Le is professor and Ph.D supervisor in the college of Computer Science in Donghua University. He is also a senior member of China Computer Federation. His research interests include database and data warehouse, software engineering theory and practice.

Mei Wang is a professor and M.S. supervisor in the college of Computer Science in Donghua University. Her primary research interests include database, image semantic analysis, and information retrieval.