Lucene Based Block Indexing Technology on Large Email Data

Chunyao Song, Yao Ge, Peng Nie and Xiaojie Yuan

College of Computer and
Control Engineering
Nankai University
Tianjin, China 300071
Email: {chunyao.song, geyao, niepeng, yuanxj}@nankai.edu.cn

Abstract—**As a warehouse for storing and managing data, a relational database supports the index mechanism, to meet users' needs of managing data resources. However, when the amount of data is too large or the users' queries are complicated, its simple index structure is not able to return an accurate query result within a short time. Thus, we need to establish a highly efficient index scheme for large amounts of data. Given that the users' primary requirement is searching keywords on a specified batch interval on large email data, where each email is associated with a batch attribute, this work builds an email retrieval system by using a full-text searching toolkit called Lucene. This work presents a scheme to build the index according to each email's batch attribute and achieves the coexistence of the block index and the integrated index. The evaluation shows that our scheme has significantly improved the searching efficiency of the email retrieval system compared to the basic system which does not allow a hybrid index structure.**

Keywords–Lucene; index; email data; big data.

I. Introduction

This is an information and Internet Plus era. The Internet is flooded with plenty of information. How to retrieve the most useful information from the massive data was a main challenge from the very beginning of the development of the Internet. The appearance of searching engine gives us a solution. Searching, as a mainstream method to get useful information, becomes part of people's daily life.

The index structure has determined the response time and query accuracy of a searching engine to a large extent. The index mechanism in a database system is a common scheme. However, the index structure of traditional relational database is very simple, and lacks the core functionality for retrieving and analyzing the contents of the files stored in the library [1]. Although it supports normal SQL (structured query language)-based queries well, it is hard to meet the requirements of a search engine. First of all, a search engine needs to search large amounts of data, and the storage format is relatively simple. How to use a database to reasonably and effectively manage this data is a difficult problem. Second, the demand of a search engine is to give an accurate response within a short time for a large number of users' query requests. Therefore, the time consuming work should be completed during the index building phase. In other words, before run time. Apparently, the index structure in the database system does not meet this need. Therefore, it is necessary to establish an efficient index library for massive data.

Lucene is a subproject of the Apache software foundation [2]. It is an open source java-based full-text search engine architecture. It provides a complete query engine, index engine and part of the text analysis engine [3]-[4]. Thus, software developers can easily achieve full-text search function in the target system. As an excellent full-text search engine architecture, Lucene has greatly improved the retrieval efficiency, by using highly optimized inverted index structure [5]. A main advantage of Lucene is that the format of the indexed file is independent of the application platform. It defines a set of 8-bytes-based index file formats so that compatible systems or applications of different platforms can share the establishment of the index file [6]-[7].

The full-text retrieval system is a software system, aiming to provide full-text retrieval services based on full-text search theory. There are two parts to complete full-text search. One is to build and maintain the index library. The other is an efficient and accurate retrieval mechanism. Lucene has provided calling interfaces for both parts. However, in practical applications, Lucene has a problem that should be noticed. The size of the index file is linearly increasing as the number of files needing to be indexed increases. When the primary requirement is to do interval filter for a specific field first, and then do keyword search, whether the search engine needs to search the entire index file every time is worth studying. Based on this, we propose a scheme to create a block index based on this field. We split the GigaByte-level index file into multiple small index files according to this field. Then, we only need to search on small index files which satisfy the query range and merge the search results during the searching phase.

When an index file reaches the GigaByte-level, for a single server, if the filter interval given by the query is small, then the block index may significantly improve the searching speed. However, when the range of the filter field in the query is large, integrated index may perform better than block index during the searching phase. As using block index needs to read multiple index files, there is a need for frequent I/O (Input/Output) operations. As a conclusion, choosing different index scheme according to different search situations may improve the average response time of the search engine as a whole.

The target searching dataset for this work is a large email dataset. Each email contains an eight-digit batch attribute. The primary user requirement is to search for a specified keyword within some batch of messages. Therefore, it is of great significance to improve the search efficiency by implementing the full-text search system which realizes coexistence of the block index and the integrated index. It is very meaningful to improve the search efficiency for different search requests.

Given the email dataset stored in MySQL - a commercial relational database management system [8], the idea of this work is to create a block index and an integrated index for each attribute of each record/email. The search function is completed based on the establishement of the index file. Thus, the user could query the system. After the basic structure of the email retrieval system is completed, the optimization by using block index is introduced. Evaluation is performed to help choose the appropriate indexing strategy based on users' searching needs. Finally, the strategy is used to improve the searching efficiency of the email retrieval system.

In the remainder of this paper, we first give a brief introduction about Lucene [2] in Section II. We will introduce the index engine and search engine of Lucene. Next, we will show our system design method and implementation details in Section III. We will explain in details how to accomplish the index building for our email retrieval system, and how to perform the searching process based on the established index. Evaluation results is shown in Section IV. We perform comprehensive experiments to select the best index strategy for different searching requirement. Section V gives the conclusion of the paper.

II. Preliminaries— A Review of Lucene Theory

Lucene consists of eight packages, each of which is invoked with other packets. They have specific functions, such as text analysis, index creating, index read-write, index structure management, and search requests parsing, etc. [9]. Lucene works by converting other data formats into text, extracting the index entries and related information from the word breaker, and then writing the information to the index file, and saving it to disk or memory. We will introduce Lucene theory from both the index engine and the search engine.

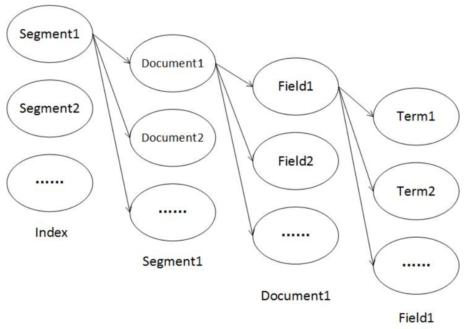

Figure 1. Lucene Index Conceptual Relationship

A. Lucene's Index Engine

There are five basic concepts in Lucene, including **Index**, **Document**, **Field**, **Term** and **Segment**. The relationships are shown in Figure 1.

- An **index** consists of multiple documents.
- A **document** consists of multiple files. It is similar to a record in relational database, which is mainly responsible for domain management [10].
- A **field** consists of several terms. Each field usually has four attributes: name of the field, value of the field, whether it is necessary to be stored in index file and whether it is indexed.
- A **term** is a string that is obtained by lexical analysis and language processing of the text, which is the searching unit. It has two attributes: the name of the term and the value of the term.
- A **segment** can be considered as a tiny index. It includes all documents needed by the index. When adding new documents to the index being searched, Lucene usually creates a new segment to avoid the cost of rebuilding the index.

When creating an index, it is not that every record is immediately added to the same index file. They are first written to a different small file, and then merged into a large index file [11]. The source provided by the user is a record in the database table. A record is indexed and then stored in the index file.

Lucene holds only one buffer when building index. However, it provides three parameters to adjust the size of the buffer and the frequency to write the index file to disk [2]. These three parameters are:

- **Merge factor**: this parameter controls the timing of merging index files on a disk into large index files and the number of documents that can be stored in each index block.
- **Minimum number of merged documents**: this parameter is the minimum value that the number of documents in memory want to write to disk. If there is enough memory, increasing this parameter could significantly improve the index efficiency.
- **Maximum number of merged documents**: this parameter is the maximum number of documents an index block can store. Appropriate increase in this parameter value can speed up the index building process and shorten the response time.

B. Lucene's Search Engine

Lucene supports a variety of query methods. Combining them allows developers to customize the queriers needed. We need three kinds of queries in this work.

- **TermQuery**: it allows to search for keywords for specified field.
- **BooleanQuery**: it supports query combination. By adding a variety of query objects and designating their logical relationship as ”and”, ”or”, ”non”, it can link the queries together.
- **RangeQuery**: it supports range search on a specific field. It can also be used together with BooleanQuery.

Based on this search engine, the searching process includes five steps. The first step is to read the index file. Lucene provides an IndexReader. After its open() method has been invoked, it will find the latest segment from the index file. It will load the meta data of this segment into memory, and further open each segment and the documents within each segment.

The second step is to build the search tree. Given the structure of the query object, Lucene will parse it into a query tree based on the logic of the query. When multiple search conditions are used, Boolean queries are usually used as logical join queries. In this case, a Boolean query can be used to represent the entire query tree.

The third step is to evaluate the weight. Weight is the factor used to calculate the score. When the query tree is obtained, the first operation is to rewrite the query tree. The purpose is to change the query tree according to the need of searching keywords change. Then, use the recursive creation of the weight tree based on the newly obtained query tree, and calculate the value of the public part of each document in the scoring formula.

The fourth step is to calculate the document score. The matching is performed by calculating the similarity between the query and the document. Each search result will be given a score. The higher the score, the higher the degree of matching. The scoring function is the **score** method in class **Scorer**. It traverses all the resulting documents to calculate the score. The scoring mechanism it used is the TF/IDF (Term Frequency/Inverse Document Frequency) [12] algorithm. The tf-idf weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. We have discussed in index engine that the documents are divided into words when creating an index. Word segmentation is also performed in the searching phase. TF/IDF algorithm considers how many times the word appears and how many words appear in the document. The TF frequency of the keyword is calculated as follows: $tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$ [12], where $n_{i,j}$ is the number of occurrences of the keyword in the document d_j, and $\sum_k n_{k,j}$ is the summation of the number of occurrences of all keywords in document d_j. For a specific document, the greater the proportion of the number of occurrences of a keyword to the number of occurrences of all keywords, the stronger the ability of this keyword to distinguish between the attributes of the document, and the greater the value of the calculated TF value [13]. The IDF value is computed as $idf_i = \frac{|D|}{|\{d:d \in t_i\}|}$ [12], where $|D|$ is the total number of documents, and $|\{d : d \in t_i\}|$ is the number of documents which contains the keyword t_i. The greater the number of documents that contain a keyword in the document set, the weaker the ability of the keyword to distinguish the attributes of the document category, resulting in a smaller corresponding IDF weight. Finally, the TF-IDF is computed as $(tf - idf)_{i,j} = tf_{i,j} * idf_i$ [12]. The resulting score reflects the ability of a keyword to reflect the document subject. The larger the final score, the better the effect of the keyword that reflects the subject of the document. The document score is computed as the summation of the score of each word segmentation term. A priority queue is used to store the resulting documents, which has a sorting function at the same time.

The last step is to return the query result. After computing the score of each searched document, Lucene will return the query results in decreasing order.

III. System Design and Implementation

The email retrieval system of this work is implemented based on JavaWeb [14] and Lucene. We have introduced the theory of index building and query processing in previous sections. We will discuss how to build the index of a dataset based on the interfaces Lucene provides, and how to accept users' searching queries and return the searching results based on JavaWeb in this section.

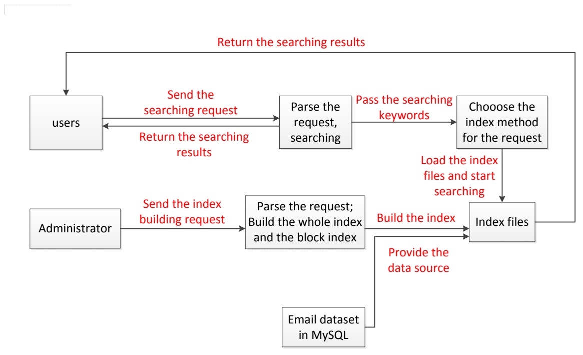
Return the searching results
users
Send the searching request
Return the searching results
Parse the request, searching
Pass the searching keywords
Chooose the index method for the request
Load the index files and start searching
Administrator
Send the index building request
Parse the request; Build the whole index and the block index
Build the index
Index files
Provide the data source
Email dataset in MySQL

Figure 2. System Design Flow Chart

The system design flow chart is shown in Figure 2. As we are trying to develop an email retrieval system, the datset we use is an email dataset. The emails have already been parsed based on receiver, sender, copy recipients, email contents, etc., and stored in MySQL. Each email has an eight digits batch label, so that users could search on specified batches. Since users have a great need to search keywords on emails with specified batches, we build two kinds of indexes for the input dataset:

- **Integrated index**: this kind of index builds index for each attribute of the email, and finally there is only one index files folder, which includes all index files needed for query processing.
- **Block index**: the blocks are divided according to the batch label of each email. We build an index files folder for each batch of the emails. The name of the index files folder is the batch name and the number of the index files folders equals the number of email batches.

After finishing the index building, the user could launch the searching request according to personal needs. The system server then decides which index method to use according to the specific request. Based on this, the server then loads the index files for searching and returns the search results.

A. *Index Building*

Index building is an important part of system implementation. Before we build the index, we need to confirm how to do the word segmentation for the documents awaited to be analyzed, which information should be stored for future use, and which fields will be used for future queries. We will introduce how to build integrated index and block index based on relevant kernel classes and the call graph. Figure 3 shows the kernel classes needed for index building and the call graph.

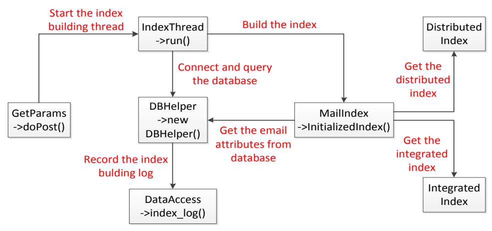
Start the index building thread
IndexThread ->run()
Build the index
Distributed Index
Connect and query the database
Get the distributed index
GetParams ->doPost()
DBHelper ->new DBHelper()
Get the email attributes from database
MailIndex ->InitializedIndex()
Get the integrated index
Record the index bulding log
Integrated Index
DataAccess ->index_log()

Figure 3. Kernel Classes Needed for Index Building and the Call Graph

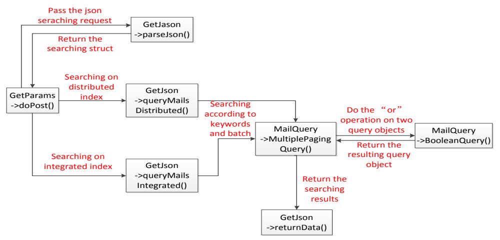
Pass the json seraching request
GetJason ->parseJson()
Return the searching struct
Searching on distributed index
GetParams ->doPost()
GetJson ->queryMails Distributed()
Searching according to keywords and batch
Do the “or” operation on two query objects
MailQuery ->MultiplePaging Query()
MailQuery ->BooleanQuery()
Return the resulting query object
Searching on integrated index
GetJson ->queryMails Integrated()
Return the searching results
GetJson ->returnData()

Figure 4. Kernel Classes Needed for Searching Process and the Call Graph

- Class **GetParams**: this class is inherited from HttpServlet. After the doPost() function of this class receives the request for index building, it will create an object of class IndexThread. It will then call the start() method of IndexThread to start the thread, and start the index building.
- Class **IndexThread**: because the data are stored in MySQL, after the thread is started, the run() method of this class will first load all batch information from the database. It will then create index file folder according to the information and name each index file using the corresponding batch name. After that, it will pass the file folder path and the current batch information to the InitializeIndex() method of the MailIndex class. If we are building integrated index, then the passed batch information is a null string. So, we could build the two kinds of indexes at the same time.
- Class **MailIndex**: since our email retrieval system needs to do full-text search, the InitializeIndex() method of this class will create an object of the DBHelper class, to connect to the MySQL database, read all attributes of the dataset, and build the index based on the interfaces Lucene provides. This method accepts two parameters, which are index building path and the email batch information.
- Class **DBHelper**: the simplest nonparametric constructor function of this class is able to connect to the database. The one-parametric constructor adds a function to get ready to execute the statement.

Since we need to build the block index at the same time when we build the integrated index, we use one parameter of the InitializeIndex() method to accept the email batch. After receiving the email batch needed for index building, we use JDBC (Java Database Connectivity) [15] to connect to MySQL, and select all emails which have this specified batch, and then create an IndexWriter object of Lucene to build the index.

B. *Searching Process*

The searching process includes parsing the search request, opening the index files for searching, and returning the searching results to users. We will show the call graph according to these three steps and explain the details. Figure 4 shows the call graph.

- Class **GetParams**: users send the post request to JavaWeb server using browser. The searching request is passed by json. In the meanwhile, http request is acquired and encapsulated by Servlet container to the HttpServlet object. The doPost() method of this class is responsible for receiving the searching request, and passing the json string to the GetJson object created. It will call the GetJson methods for future parsing, querying, and results returning.
- Class **GetJson**: the parseJson() method of this class is responsible for parsing the json string, and generate the searching structure. It has two parameters. The first is the json string for the searching request. The second is whether should use the block index to parse the searching request. The generated searching structure is returned to GetParams. After receive the searching structure, GetParams would call different GetJson methods for integrated index and block index. The block index will call queryMailsDistributed() method, which needs to do searching on every index file within this batch, while integrated index will call queryMailsIntegrated() method, which only does searching on the single index file. These two methods will both call MultiplePagingQuery() method of MailQuery.
When accepting the request, the relevant information is received. Thus, after getting the searching results, the returnData() method of this class will call the relevant methods, to generate the response data and pass the response to the user.
- Class **MailQuery**: this class will do searching according to the received request. The MultiplePagingQuery() method and the BooleanQuery() methods are two kernel methods.

After receiving the searching keywords and the batch information, the system will use QueryParser in MultiplePagingQuery() to construct the searching object according to the keywords. It will use BooleanQuery to do the ”or” operation on the two searching objects: one is the QueryParser of the keywords, while the other is the RangeQuery of the batch. Further, it will use IndexReader and IndexSearcher to open the index, and perform the searching process.

IV. Evaluation

We have discussed how to build the two kinds of indexes, how to perform the searching process and how to return the results. However, we are still not sure when to use which kind of index under what scenario. Thus, we need to do the searching evaluation. We will discuss the design of the

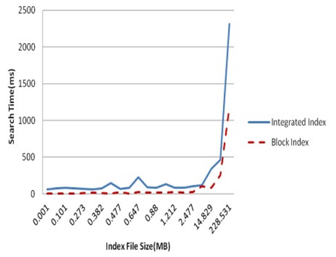

Figure 5. Partial index files searching time comparison

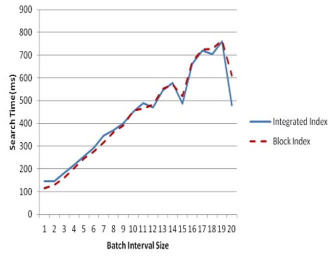

Figure 6. Searching time comparison for batch interval in 1-20

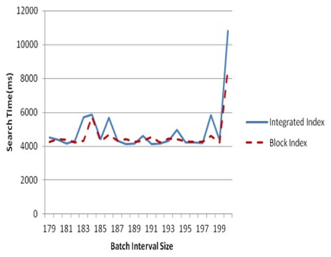

Figure 7. Searching time comparison for batch interval in 179-200

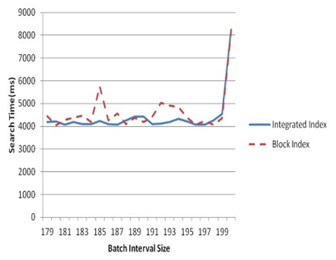

Figure 8. Searching time comparison for exchangeable search and batch interval in 179-200

searching evaluation, searching results analysis and searching strategy development based on experimental results in this section.

A. *Evaluation Design and Implementation*

The batch interval decides how many block index files needed to be loaded into the disk. The size of a block index depends on the amount of data needed to be loaded into the memory. Different keywords will result in different sizes of result sets. They will all affect the searching efficiency. Thus, we perform the experiments by varing parameters from these three aspects.

The searching evaluation uses URL and URLConnection classes of Java [16] to send searching requests to Class GetParams. In order to get the statistics of the searching time conveniently, we modify the searching code so the server only return the searching time, instead of returning the searching results. We then output the returning searching time directly to an excel file, and finally get the statistics for analysis.

B. *Evaluation Results and Analysis*

The effect of the size of index file: since the number of emails in different batches is different, and the size of each block index file is different, in order to evaluate the effect of the size of index file to searching efficiency, we need to search on each batch. The partial average result is shown in Table I.

TABLE I. THE EFFECT OF SIZE OF INDEX FILE TO SEARCHING EFFICIENCY

size of index file(MB)	block index time(s)	integrated index time
0.001	0.006	0.061
0.406	0.009	0.073
1.107	0.012	0.077
4.157	0.034	0.109
22.015	0.123	0.260
51.934	0.254	0.463
94.237	0.372	0.880
228.368	1.250	2.208

We perform the searching evaluation on 200 batches. We show partial results according to certain interval for clarity. The comparison of integrated index and block index is shown in Figure 5. We can see from the experimental results that when we search on one batch, block index is obviously better than integrated index. The smaller the index file, the more apparent difference between the two methods. When the size of the index file is less than 1MB, the searching efficiency of block index is 10 times better than integrated index. As the size of the index file increases, the advantage of the block index is decreases.

The effect of the batch interval: We have seen from the previous test that when the batch interval is 1, the block index has significant advantages. However, as the batch interval is increasing, the I/O cost of block index is increasing as well. So the advantage is expected to decrease. So, the assumption is there is a cross point that before this point, the block index is better than integrated index. While after this point, the integrated index is better than block index. We perform the experiments on 200 batches, and extend the batch interval from 1 to 200 gradually. The partial average searching time is shown in Table II.

TABLE II. THE EFFECT OF SIZE OF SEARCHING BATCH INTERVAL

size of batch interval	average block index searching time (s)	average integrated index searching time (s)
1	0.115	0.146
10	0.391	0.399
20	0.765	0.756
50	1.699	1.621
90	2.846	2.769
140	4.613	4.152
200	8.590	10.811

Because it is hard to present the 200 test results in a single figure, we show the first part of the results in Figure 6. We can see that block index is better than integrated index when batch interval is less than 11. It means that when users' searching request includes less than 11 batches, we should use block index. The integrated index performs better than block index in the middle part.

However, the last experiments did not perform as expected. In our expectation, since the block index needs to load multiple index file folders to memory, the frequent I/O operation becomes the bottleneck. Thus, the searching efficiency should be worse than integrated index. However, as we seen in Figure 7, the block index performs close to or even better than integrated index in most cases. Since we did the experiments on block index and integrated index respectively, it is possible that the system did not do the whole I/O operations in every searching. Thus, we perform experiments for block index and integrated index alternatively. The result is shown in Figure 8. We can see that in this case the integrated index performs better than block index. So, we should still use integrated index when the batch interval is large.

The effect of the size of the results set: Under the single server environment, the searching results are stored in the Java stack. When the size of the returning results is large, continuous searching will result in high memory usage. This will further affect the data exchange between disk and memory, resulting in low searching efficiency.

We perform a test on the effect of different keywords and the results are similar to previous experiments, thus we omit the figure here. The cross point of graph appears in the range 10 to 15. It means the search efficiency of block index and integrated index is equal when the size of batch interval is 10 to 15. In other words, block index performs better in small batch interval, while integrated index performs better in large batch interval.

C. *Searching Strategy Development*

We can see from the experimental results, when the batch interval is less than 10, the searching efficiency of block index is better than the integrated index in most cases. However, when the batch interval is large, say, approching 200, the searching efficiency of integrated index is better than block index. When the batch interval is in the middle, the searching efficiency of the two methods are close. However, due to the uncertainty of searching request, there could be great differences among the two continuous searching requests. Thus, to reduce the disk I/O operation, it is recommended to use integrated index.

Moreover, considering a large batch will decrease the searching efficiency of block index, we could record the large batches in advance. When the searching request touches on many those batches, we could use integrated index.

V. CONCLUSIONS AND FUTURE WORK

Compared to the traditional SQL language, Lucene has unbeatable searching efficiency. It provides friendly interfaces and clear documentation. So, the software engineer could develop a search engine in a short time. However, when the amount of data increases sharply, the linear increase of the size of the index files lowers the efficiency of searching within a specific range.

Our system focuses on a specific email dataset. We need to satisfy the requirement to searching keywords within some specific batches. So, we need to realize the function for fast search under multiple restrictions. We propose to divide the index files according to batch labels, in other words, block index. We develop the system based on JavaWeb [17] and Lucene. We implemented both integrated index and block index, to accpet the user requests and return the results to users. In order to decide the index using strategy, we perform comprehensive searching experiments, considering the problem from three aspects: the size of a index file, the searching batch interval, and the keywords differential. We develop a meaningful method to compute the average searching time. We perform comprehensive experiments and the evaluation results show our scheme has significantly improved the searching efficiency of the email retrieval system compared to the basic system which does not allow hybrid index structure. We compare the searching efficiency using both tables and figures, and show the strategy at the end. We have reduced the users' waiting time while at the same time when satisfying users' requirement.

Although the block index has increased the efficiency to some extent, we could still improve in some aspects. Currently, we only use the batch interval to select different index methods. We could consider more factors for choosing the index methods. When there are lots of searching results, it will occupy a lot of memory so the searching efficiency will decrease. We could try better results returning methods. We use single machine in this work. However, when the data size further increases, we could try to deploy the system in a distributed manner. In that case, how to merge the searching results and do the scoring is another problem worth considering.

ACKNOWLEDGMENT

This work was supported in part by Natural Science Foundation of Tianjin, under Grant No. 17JCYBJC23800; National 863 Program of China, under Grant No. 2015AA015401; and Research Foundation of The Ministry of Education and China Mobile, under Grant No. MCM20150507. Chunyao Song and Yao Ge contribute equally to this work.

REFERENCES

[1] Y. Xu, Y. Zhu, C. Li, and W. Wang, "The design and implementation of lucene based full-text retrieval on massive database," *Journal of Hunan University of Technology*, vol. 25(2), pp. 81–84, 2011.

[2] Lucene. http://lucene.apache.org/. [accessed: 2017-06-06].

[3] K. Yang, X. Shi, and E. Tang, "Reptile based software defects prediction," *Journal of Nanchang College of Education*, vol. 31(6), pp. 125–128, 2016.

[4] J. Zhang and J. Wang, "Discussion about the integration of lucene in haobai searching engine," *Science & Technology Information*, vol. (21), pp. 12–12, 2012.

[5] H. Tang, Y. He, X. Xu, and C. Xu, "Lucene based distributed parallel index," *Computer Technology and Development*, vol. 21(2), pp. 123–126, 2011.

[6] H. Wu, "Lucene based email forensics technology," *Netinfo Security*, vol. 10, pp. 181–184, 2013.

[7] L. Yuan, "Discussion about the functions and applications of lucene based full-text index," *Science and Technology of West China*, vol. 11(5), pp. 37–38, 2012.

[8] https://www.mysql.com/. [accessed: 2017-06-06].

[9] X. Shi and Z. Wang, "An optimized full-text retrieval system based on lucene in oracle database," *Enterprise System Conference*, pp. 61–65, 2014.

[10] R. Gao, D. Li, W. Li, and Y. Dong, "Application of full text search engine based on lucene," *Advances in Internet of Things*, vol. 02(4), pp. 106–109, 2012.

[11] S. Yue, W. Li, L. Wang, and S. Guang, "Index for database retrieval based on lucene," *Journal of Jilin University (Science Edition)*, vol. (5), pp. 995–1000, 2014.

[12] A. Rajaraman and J. Ullman, "Miing of massaive datasets," pp. 1–17, 2011.

[13] X. Wang and G. Ren, "An improved wpr algorithm based on the most recent searching period's referencing frequency," *Computer Science*, vol. 43(2), pp. 86–88, 2016.

[14] http://docs.oracle.com/javase/tutorial/deployment/webstart/. [accessed: 2017-06-07].

[15] https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/. [accessed: 2017-06-06].

[16] https://www.java.com. [accessed: 2017-06-07].

[17] http://www.vogella.com/tutorials/javawebterminology/article.html. [accessed: 2017-06-06].