A Distributed Search Engine Based on a Re-ranking Algorithm Model

Jingyong Wan, Beizhan Wang*, Wei Guo, Kang Chen, Jiajun Wang
Software School of Xiamen University
Xiamen, China
jywan@outlook.com, wangbz@xmu.edu.cn, guowei910516@163.com, checkking@foxmail.com, wangjiajun422@gmail.com

***Abstract*—With the rapid increase of websites and the explosive growth of Internet information, the centralized search engine will face great challenge in mass data processing and mass data storage. However, the distributed search engine can solve the problem effectively. In this paper, we describe the design and implementation of a distributed search engine that is based on Apache Nutch, Solr and Hadoop. Considering users click logs, we propose a re-ranking algorithm based on Lucene scoring. Our experimental results show that our approaches significantly satisfy users' massive data searching demand while maintaining high reliability and scalability.**

***Index Terms*—Distributed Search Engine, Hadoop, Re-ranking Algorithm.**

I. Introduction

With the rapid increase of websites and the arrival of Big Data, the centralized search engine will face great challenge in the mass websites crawling, indexing and the real-time searching. Search engine should have the distributed processing capabilities to enhance the system's ability of processing information. Distributed search engine can make up for the shortcomings of centralized search engine.

A distributed search engine can be divided into three modules: distributed crawling module, distributed indexing module and distributed retrieval module [1]. In this paper, we design a distributed search engine system, which is based on three open-source software: Nutch[2,3], Hadoop[4,5] and Solr[6]. We use the MapReduce parallel computing framework and distributed file system HDFS to implement our distributed search engine. In particular, we use MapReduce to crawl the web pages and store all the pages in the HDFS. Then the pages are submitted to Solr server to create index. In order to improve system reliability and performance, we set up three Solr search engine servers and use ZooKeeper[7] to manage them, which is a service for coordinating processes of distributed applications. ZooKeeper distributes the processing job of mass pages to multiple servers to create indexes in indexing phase. After that, every Solr server stores a part of the whole index files. It avoids the storage problem of only one machine and enhances the indexing efficiency because of distributed indexing. It is also because the index files are located in many machines, we need ZooKeeper to coordinate all search results in every Solr server, then return the final result to users. Figure 1 shows the system architecture diagram of the distributed search engine we designed.

Solr is a high performance search server built on Apache Lucene, so its scoring method is Lucene scoring actually. Lucene implements a variant of the TF-IDF scoring model [8]. This scoring strategy is determined only by the correlation of document and query. It is a prior ranking strategy. Since the users' demand is different although they give the same query, the search results based on this strategy cannot meet users' demand in many times. Since users' true intentions can often be seen from their click behaviors, we propose a novel re-ranking algorithm, which takes users' feedback into account according to users click logs, to make up for the inadequacy of Lucene scoring.

This paper is organized as follows. In Section 2, we describe the design of distributed search engine. Section 3 focuses on the re-ranking algorithm we proposed. Section 4 is our experimental process and results. We end this paper with some conclusions in section 5.

II. Distributed Search Engine Design

The distributed search engine includes web pages crawling and parsing, index creating and searching. Apache Nutch Wiki [9] describes the crawling process in detail. In this section, we

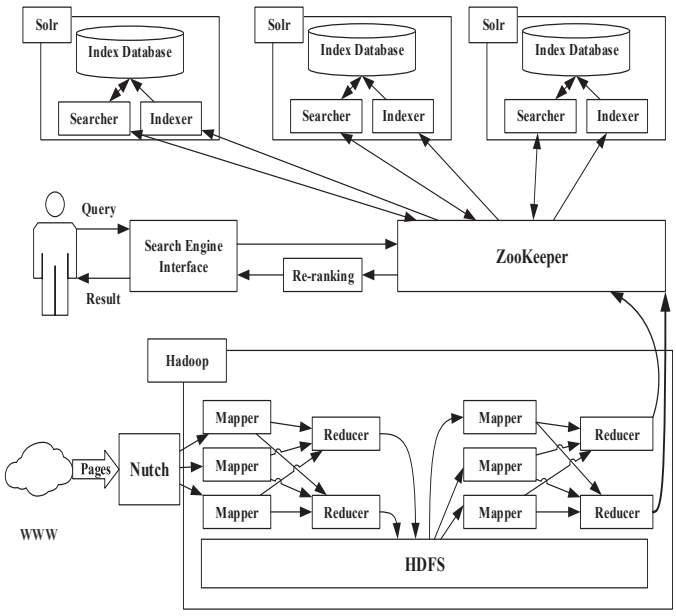

Fig.1 The system architecture diagram of the distributed search engine

describe the implementation of the distributed indexing module in detail.

The web pages contents, which are parsed in the previous stages, are stored in HDFS. We encapsulate the parsed web contents and related properties in *NutchDocument* class using MapReduce job. Then we create indexes by Solr interface. The MapReduce processes are as follows:

1) Map phase

Input: <url, Parse Contests>, "Parse Contents" represents the parsed web contents.

Process: Filtering. Encapsulating in *NutchWritable* Class.

Output: <url, *NutchWritable*>.

2) Reduce Phase

Input: <url, [*NutchWritable* Set]>.

Process: Parsing out the url, title, content, host, segment, boost, digest and tstamp fields. Encapsulating in *NutchDocument* Class.

Output: <url, *NutchDocument*>.

The *NutchDocument*, outputted by reducer, is indexed by Solr servers. We use ZooKeeper to manage multiple Solr servers, when indexing request is received, ZooKeeper will allocate the indexing requests to one of the multiple Solr servers. The indexing jobs will be assigned to multiple Solr servers and each server will store a part of the whole index files. That can reduce the burden to create and store the index files using only a single server.

III. Re-ranking Algorithm

A. Re-ranking Algorithm Model

In our model, we define some notations for brevity. The *click* represents whether the result is clicked or not. The *q* is short for query which represents the user's search query. The *pos* is the position in the given result page. Our distributed search engine system will calculate $P(click \mid q,url)$ and $P(click \mid pos)$, which represent the click ratio of the *url* when given the query and the click ratio in the position *pos* when given result page respectively. Our model is based on statistical probability theory. Only when the number of occurrences of a query is more than the threshold we set did we use our re-ranking algorithm. It makes our model more convincing.

Equation 1 is the scoring formula which consists of two parts: $ClickScore_{q,url}$ and $LuceneScore_{q,url}$, where λ is the weighting distribution parameters. The users' click score is denoted by $ClickScore_{q,url}$. The Lucene original score is denoted by $LuceneScore_{q,url}$.

$$Score_{q,url} = \lambda \cdot ClickScore_{q,url} + (1-\lambda) \cdot LuceneScore_{q,url} \quad (1)$$

In most times, the users click feedback implies their demand. So we give $ClickScore_{q,url}$ a weight higher than $LuceneScore_{q,url}$, in other words, $\lambda \in (0.5,1)$. The detailed calculation expressions are as follows:

$$ClickScore_{q,url} = 1/\left(1+\exp(-\alpha \cdot P(click \mid q,url)/P(click \mid pos))\right) \quad (2)$$

$$P(click \mid q,url) = Count(click \mid q,url)/Count(click \mid q) \quad (3)$$

$$P(click \mid pos = x) = Count(click \mid pos = x)/\sum_{i=1}^{N} Count(click \mid pos = i) \quad (4)$$

Where α is a constant parameter which is given initially. The *Count* function is to calculate the number of the users clicks over a period of time such as one day. $Count(click \mid q,url)$ indicates the number of clicks in an *url* when the query is given. $Count(click \mid q)$ represents the number of clicks of a query. And $P(click \mid pos = x)$ represents the priori click ratio in the position *x*, which is equal to the number of clicks in the position *x* divided by the total number of clicks. This value is updated once a month, that is, we calculate it every month.

When we calculate $ClickScore_{q,url}$, we normalize it into (0,1) interval. According to Eq. 1, our final score, which belongs to (0, 1) interval, is consistent with the original Lucene score. The above is our re-ranking algorithm model.

B. Re-ranking Algorithm MapReduce Process

The format of a record in users click log is as follows:

timestamp \t ip \t query \t click \t url \t position

We collect and save the click log into log files. Each row of the log files is called a row record. Now we describe MapReduce process of every step in detail.

Step 1: Calculating $Count(click \mid q)$.

1) Map Phase

Input: <Row Key, Row Record> from users click log files.

Process: Parsing out the query in each record.

Output: <query, 1>

2) Reduce Phase

Input: <query, [Click Count Set]>

Process: Summing the number of click with the same Key. Filtering out the total click below the threshold we set.

Output: <query, Sum of the click *query_cnt*>

The outputs of this step are the results we need. We save them to HDFS and use them in the later steps.

Step 2: Calculating $Count(click \mid q,url)$.

1) Map Phase

Input: <Row Key, Row Record> from users click log files.

Process: Parsing out the query \t url in each record. Filtering out the query if it is not in the output of the first step.

Output: <query \t url, 1>

2) Reduce Phase

Input: < query \t url, [Click Count Set]>

Process: Summing the number of click with the same Key.

Output: < query \t url, Sum of the click *url_cnt*>

Step 3: Calculating $Count(click \mid pos = x)$.

1) Map Phase

Input: <Row Key, Row Record> from users click log files.

Process: Parsing out the position in each record. Filtering out the position if it is out of the range we set.

Output: <position, 1>

2) Reduce Phase

Input: < position, [Click Count Set]>

Process: Summing the number of click with the same Key.

Output: < position, Sum of the click *position_cnt*>

Theoretically, we should scan the whole users log files and calculate the click times of all possible positions. In this case, the computation is very heavy. Fortunately, under the guidance of the recent-biased principle, we only need to care about a period of recent time, for example, one or three months. Meanwhile, less than 10% of search engine users click on search engine results pages that appear after the third page. To reduce computational complexity, we calculate the top 30 positions instead of all possible positions.

Step 4: Calculating $ClickScore_{q,url}$.

According to the Eq. 2, 3 and 4, we can calculate the click score of each *url* with given query. We save it in a dictionary for each query. The format is as follows:

query \t url1 clickscore1 \t url2 clickscore2 \t …\t urlN clickscoreN

Our search engine system stores these dictionaries into file system in the form of HashMap. When an user search a query, our system will determine whether it is in the HashMap. We extract the click score of the query if it is in the HashMap, and get the final score according to the Eq. 1. Then the results pages of the search query are re-ranked by the final score.

IV. Experimental Evaluation

In this section, we present the concrete implementation process of our distributed search engine, including experimental datasets and results. In our experiment, we use 11 ordinary Linux (CentOS) hosts to do our experiment.

A. Experimental Datasets

In our experiment, we crawl and parse data on the Software School of Xiamen University (SSXMU) related web pages using the distributed search engine we designed. A total of 41,529 pages about SSXMU are crawled and parsed in ten hours. The size of these data are about 3.02G and we store them in HDFS. We had used stand-alone mode to crawl and parse the pages in ten hours as a comparison. The result is only 20,653 pages were crawled and parsed. It shows that our distributed web crawling and parsing approach enhance the efficiency greatly.

B. Crawling and Parsing Evaluation

Table I shows the performance of crawling and parsing web pages with different data size of pages using two hosts. As table I shows, the average speed of crawling and parsing is 77.87MB/h with two hosts. When we gradually increase the number of cluster hosts, we calculate the average process speed of with different number of cluster hosts using the same way as the two hosts. As Fig. 2 shows that we draw a line chart with the average speed of different hosts.

We can see from Fig. 2 that the processing speed is not linear growth while the number of hosts grows. It is because the network bandwidth is limited, and there is a performance bottleneck of the crawled sites.

C. Indexing Evaluation

We test indexing performance in multiple sets of data and calculate the average indexing speed. Table II shows the indexing speed with a single search engine server. We can see from table II that the average indexing speed, with only one search engine server, is 95.30MB/h. Similarity, we get the average indexing speed with two and three search engine servers respectively. Then we draw a line chart with the average indexing speed with different number of hosts, as Fig. 3 shows. When the number of search engine servers increasing, the average speed of search engine indexing increase, but the increase speed gradually decreases.

D. Searching Evaluation

In order to test the performance of searching, we use http_load [10], a Linux performance testing tool, to test the throughput of the distributed search engine servers by simulating the concurrent users. Figure 4 shows our test results. Increasing the number of servers, the system throughput decreases only in the case of fewer concurrent users. With the increase of the number of concurrent users, the increment of the system throughput is not obvious. This is mainly because coordinating user query requests and merging multiple servers search result need to spend extra time.

TABLE I The performance of crawling and parsing pages with 2 Hadoop Cluster nodes

	Data size of pages(MB)	Process Hours (h)	Process Speed(MB/h)
1	22.10	0.30	73.66
2	10.34	0.20	51.70
3	50.10	0.56	89.46
Sum	82.54	1.06	77.87

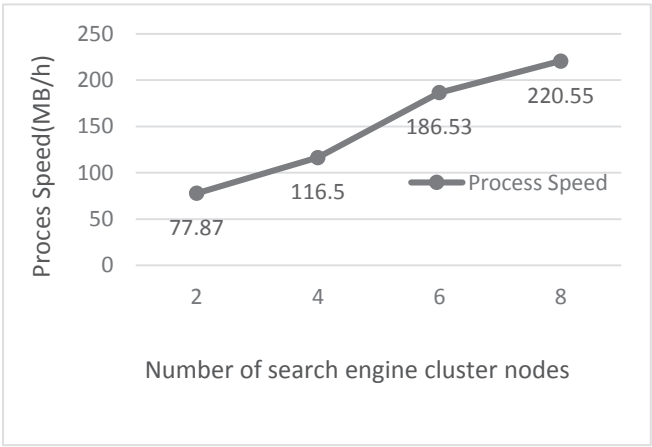

Fig.2. Average crawling and parsing speed of different hosts

TABLE II A single search engine server index test results

	Data size of pages(MB)	Process Hours (h)	Process Speed(MB/h)
1	22.10	0.17	128.23
2	10.34	0.09	116.44
3	50.10	0.52	96.79
4	98.53	1.12	87.97
Sum	82.54	1.90	95.30

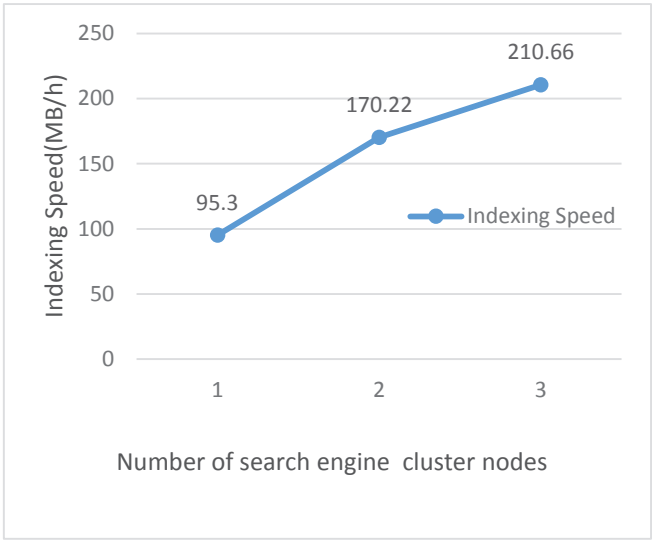

Fig.3. Indexing speed for different search engine cluster nodes

E. Re-ranking Algorithm Evaluation

In order to verify the reliability of our re-ranking algorithm model based on users click logs, we create a test web page with high Lucene score about the query "软件学院"(Software School) and put it into the web crawler database. Without using our re-ranking algorithm model, the search results top five are shown in Fig. 5. As we can see, the first position is our test page. We repeat the search query many times in the test page, so it ranks first because of high Lucene score. It reveals the drawbacks of the Lucene score strategy.

To simulate the users click behavior, we ask many students to search the query "Software School" and click the result they need. We collect the users' click log and save them into HDFS. Finally, we collect 30,000 click logs in total. A fragment of the users click logs is shown in Fig. 6.

Based on the click logs, we use the re-ranking algorithm model described above to re-rank the search results, and the final re-ranked results are shown in Fig. 7.

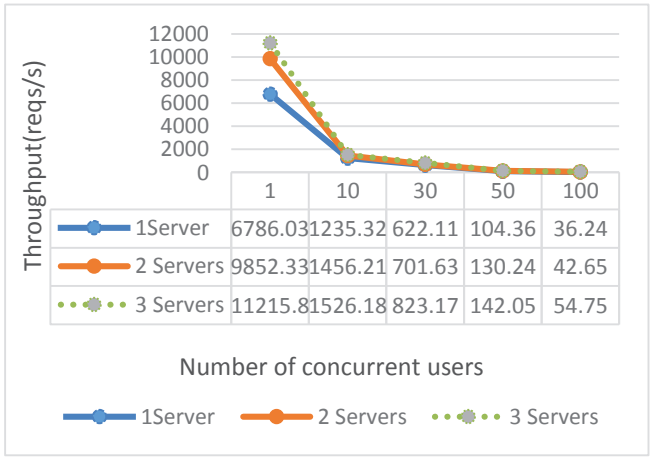

Fig.4 Throughput of different number of concurrent users

Fig.5. Top five of the search results without using re-ranking algorithm model

```
1422086247068  172.16.13.210  软件学院  click  http://software.xmu.edu.cn/View/english/faculty/qgh_en.htm  3
1422086247070  172.16.13.195  软件学院  click  http://localhost:8080/SoftwareSearchEngine/test.html  2
1422086247071  172.16.13.195  软件学院  click  http://software.xmu.edu.cn/View/english/faculty/qgh_en.htm  3
1422086247072  172.16.17.210  软件学院  click  http://localhost:8080/SoftwareSearchEngine/test.html  2
1422086247073  172.16.13.191  软件学院  click  http://localhost:8080/SoftwareSearchEngine/test.html  2
1422086247074  172.16.13.198  软件学院  click  http://software.xmu.edu.cn/View/index.aspx  1
1422086247075  172.16.13.198  软件学院  click  http://localhost:8080/SoftwareSearchEngine/test.html  2
1422086247077  172.16.13.210  软件学院  click  http://software.xmu.edu.cn/View/index.aspx  1
1422086247080  172.16.13.198  软件学院  click  http://localhost:8080/SoftwareSearchEngine/test.html  2
1422086247083  172.16.13.212  软件学院  click  http://localhost:8080/SoftwareSearchEngine/test.html  2
1422086247084  172.16.13.210  软件学院  click  http://localhost:8080/SoftwareSearchEngine/test.html  2
1422086247085  172.16.13.210  软件学院  click  http://software.xmu.edu.cn/View/index.aspx  1
```

Fig.6. Users click log

Fig.7. Top five of the search results using re-ranking algorithm model

We see that the original rank first result drop to the second position. With the accumulation of users click logs, the results we predict based on our model tend to be better. It proves that our re-ranking algorithm model have solved the problem that Lucene built-in scoring algorithms do not consider users' feedback. Our re-ranking algorithm takes the users' feedback into account and gives the better result.

V. CONCLUSIONS

In this paper, we designed and implemented a distributed search engine by using Nutch, Solr and Hadoop. We mainly described the MapReduce procedure of our system and proposed a new re-ranking algorithm model. The experimental results showed that our distributed search engine had good performances in crawling, parsing, indexing and searching, it solved the problem of massive data processing to some extent. The re-ranking algorithm makes the results pages more reliable and reasonable, because it takes the users' feedback into consideration.

REFERENCES

[1] Chen N, Xiangyang C. Investigation of Distributed Search Engine Based on Hadoop[J]. TELKOMNIKA Indonesian Journal of Electrical Engineering, 2014, 12(9): 6954-6957.

[2] Apache Nutch, http://nutch.apache.org/

[3] Khare R, Cutting D, Sitaker K, et al. Nutch: A flexible and scalable open-source web search engine[J]. Oregon State University, 2004, 32.

[4] Apache Hadoop, http://hadoop.apache.org/.

[5] Ding J, Yang B. A new model of search engine based on cloud computing[J]. International Journal of Digital Content Technology and its Applications, 2011, 5(6): 236-243.

[6] Apache Lucene, http://lucene.apache.org/solr/.

[7] Apache ZooKeeper, http://zookeeper.apache.org/.

[8] http://lucene.apache.org/core/5_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

[9] http://wiki.apache.org/nutch/Nutch2Crawling.

[10] Http_load, http://www.acme.com/software/http_load/.

.