Research of Spark SQL Query Optimization Based on Massive Small Files on HDFS

Kefei Cheng[1, a], Xudong Chen[1, b], Ke Zhou[2, c], Xianjun Deng[1, d] and Zhao Luo[1, e]

[1]Key Laboratory of Network Intelligence and Network Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065，China

[2]Chongqing Municipal Public Security Bureau Cyber Security Department, Chongqing 401120, China

[a]chengkf@cqupt.edu.cn, [b]996468223@qq.com, [c]zkheartboy@qq.com, [d]406455850@qq.com, [e]240860747@qq.com

Abstract. This paper focuses on the low efficiency of Spark SQL reading massive small files on HDFS in 4G Industry Application Card (IAC) business analysis system. To solve this issue, we propose a Local Merge Storage Model (LMSM) for 4G IAC small files. In this model, locality is enhanced by exploring the type and time of small files. Then, Spark is used to merge small files into the Parquet column storage file and store them to HDFS. Finally, according to the experimental results, after merging partitioned storage of small files, Spark SQL query efficiency increases up to 60 times higher.

Keywords: Industry Application Card data**,** Spark SQL, HDFS, Small Files, Parquet.

1. Introduction

In recent years, with the rapid development of mobile network, 4G network is more and more mature. Lots of industry application cards (IAC) are gradually converted from the traditional GPRS network to 4G network. As a result, the business data generated by IAC users soars. In the case of a certain mobile carrier in one province, its business data is approximate 400GB per day with different types and various sizes. For example, S1MME data can be up to 10GB in a single file, but the other data like email, FTP are commonly only dozens of KB. As a result, storing these data efficiently to support the subsequent data analysis and mining is the key issue at 4G industry application card (IAC) business analysis system.

Google revealed its storage and computation strategy while dealing with big data in GFS [[1]] and MapReduce [[2]]. Inspired by GFS and MapReduce, Doug Cutting developed Hadoop [[3]], where its HDFS (Hadoop Distributed File System) is one open-source implementation of GFS. In 2009, AMPLab of the University of California, Berkeley developed Spark [[4]], which is a memory-based computing framework. Compared to MapReduce, Spark has the following advantages: 1) The intermediate result in the computing process is stored in the memory instead of the disk, and it is spilled to disk when the memory is insufficient. This reduces the disk I/O and improves the computing speed greatly. 2) A DAG (Directed Acyclic Graph) computing mode [[5]] is proposed. When some exception incurs a computing task failure, Spark will find the point where the computing fails based on the DAG generated by the computing task and restart from that point instead of the beginning. Spark SQL is widely used by more and more developers in the field of big data real-time query because it can use SQL for interactive query.

HDFS is designed to stream access the big dataset of TB or PB level. While for large amounts of small files, which are much smaller than the size of the HDFS block (in hadoop2.x and 3.x version the block size is 128M by default), its access is inefficient. As a distributed parallel computing framework, Spark does not provide file management functions, and it must rely on other file systems. HDFS has become the first choice for big data storing due to its low cost, high availability and scalability. And a huge ecosystem around HDFS has been build in which Spark has been fully integrated. Therefore, using HDFS to distribute data storage and using Spark for distributed

computing has become the industry's first choice when dealing with big data. When using Spark for computing, the data will be read from HDFS into memory, then the results of the computation are output or deposited into HDFS. Massive small files will bring the following problems for storage and computation: Firstly, according to the features of HDFS distributed storage, HDFS will allocate a block for each small file, and each block will be stored as an object with the size of 150 bytes. When storing massive small files, the NameNode of HDFS will store a lot of metadata for these small files, which will occupy a lot of memory such that the overall performance of HDFS is reduced. Secondly, when Spark SQL or other distributed computing framework read these small files, they need to read the block one by one. Each time a small file is read, a Spark task will be generated, which will create a great amount of disk I/O and Spark tasks, and thus greatly reduce the computing efficiency.

2. Related Work

For the problem of the low efficiency of storing large amounts of small files for HDFS, lots of research has been done by related researchers.

The Hadoop system itself provides some merge strategy of Hadoop archive (HAR)[[6]], SequenceFile[[7]], and MapFile[[8]] for massive small files. HAR uses a hierarchical structure to package and index small files into a HAR file, which will alleviate the memory pressure of the NameNode. The file structure is shown in Fig.1. In fact, this solution only solves the memory pressure brought by massive small files to the NameNode. When one file is accessed, it needs to be indexed twice to read the file itself, so the reading time will be longer. In addition, once the HAR is created, it cannot be modified, so it has to be re-indexed when the file needs to be changed. SequenceFile stores the file in the form of key-value. The file name is key and content is value, whose structure is shown in Fig.2. This strategy merges small files through file names. The biggest disadvantage is that the keys are not sorted. Therefore, when one file is accessed, the entire file needs to be traversed, resulting in low random reading efficiency. MapFile, another solution, is a combination of SequenceFile and index. In MapFile, it consists of data and index, where data stores content, and index stores index files. Because the file is indexed, MapFile's retrieval efficiency has an obvious promotion over SequenceFile. Its file structure is shown in Fig. 3.

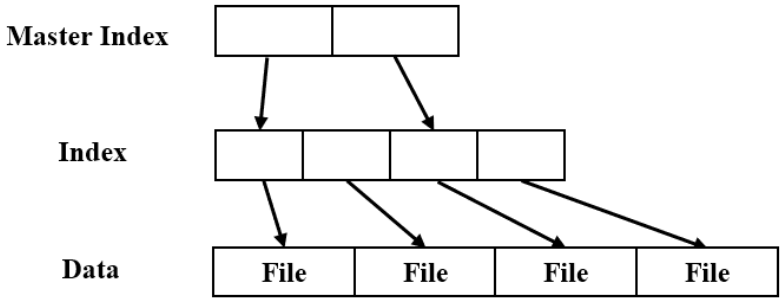

Fig. 1 HAR File Structure Figure

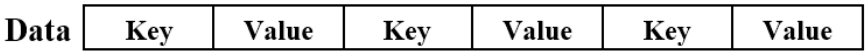

Fig. 2 Sequence File Structure Figure

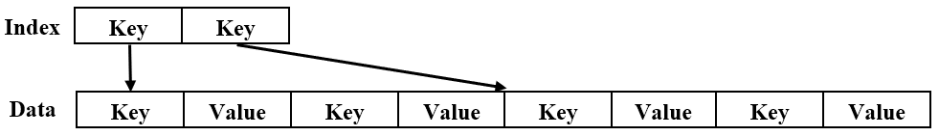

Fig. 3 MapFile File Structure Figure

Meng Li, Sheng Cao[[9]] proposed a method to improve storage efficiency of small educate resource files on HDFS. Their main idea was to classify the massive small files, merge them by classes, and index the merged files aiming at reducing the amount of index items in namenode and improving the storage efficiency. Yuwan Gu, Wenwen Wang[[10]] put forward an optimization of

massive small files storage and accessing on HDFS to improve the efficiency of HDFS. First, they got the relationship between small files by analyzing many history access logs, and then merged these correlative small files into a big file which would be stored on HDFS. When the client read data from HDFS, the system would prefetch the related files which were most likely to be visited next according to the relevance of small files. As a result, it could reduce the number of requests for NameNode and increase the hit rate and processing speed. This method needs to get access logs for analyzing large files in advance, which is not applicable to some small files that are stored in real time. Tong Zhen, Weibin Guo[[11]] proposed a merging strategy, which merged small files into a big file, and created the mapping relationship between small files and merged files at the same time. Then they stored the mapping relationship in HBase. When user need to get a small file, the meta information of the file would be retrieved at first, then the merged data would be read and target file will be returned to the user. This solution solves the storage problem of small files to a certain extent, but it relies on HBase, which will affect the performance of the overall cluster.

Above studies have solved the problem of small file low visit efficiency on HDFS in different ways. However, the object after the merge is the user rather than some computing framework, such as Spark and MapReduce, so the data mining and analysis in small files is limited. Besides, their merge strategy is not applicable to column structure data, such as Parquet[[12]], ORC[[13]] and so on, whose files have a unique identifier at the beginning or end for judging whether it is an illegal file. Above merge strategies will destroy the file structure. Columnar storage has several advantages for Spark SQL queries: 1) Since one field is the same type, so more efficient compression coding can be used to save storage space and thus reduce disk IO. 2) When reading the file, unnecessary columns can be skipped and vector operations is supported to get better scanning performance in columnar storage.

Based on the above research, combined with the features of 4G IAC data and the consideration of Spark SQL computation, we proposed a local merge storage model (LMSM) to cache 4G IAC small files. Our model used time interval T as the merge point and spark was used to merge small files according to T, then the merged files would be stored to HDFS by time partition. Partition was used to index the file and unnecessary files could be filtered according to the query condition of the users in order to improve the query efficiency of Spark SQL.

3. Traditional HDFS Storage Efficiency Analysis

3.1 4G IAC Data Feature Analysis.

1) The original data is text file, in which each field is divided by "|", and the name of files are named by file type and time.

2) Certain fields in the data can be used as a basis for determining the failure of the IAC user. For example, the APN bounded by the user can determine whether the user is abnormal due to an APN exception during a certain period.

3) Due to the wide range of services included in 4G IAC, the types of data generated are also numerous. For example, IM data generated by instant messaging, VoIP data generated by user calls, HTTP data generated by users surfing the Internet, and so on.

4) Files are usually generated at regular time intervals. Some of them are large files, such as S1MME and HTTP, which are beyond the scope of this paper. While others such as Email, FTP and RTSP are generally small, mostly around 1MB, and these data in this time period need to be merged into HDFS.

3.2 Direct Reading of Text File Efficiency Analysis.

When the massive small files are stored into HDFS directly, there will be the following problems:

1) Because the file type is text, when Spark SQL reads this file, it needs to read the entire file even if only a few of fields are needed. The total input data size that Spark needs to compute at this time is:

$$S_{total} = \sum_{i=1}^{n} S_i + Sc \tag{1}$$

where S_i is the size of i-th file, n means the number of files, and Sc means the size of the intermediate data generated by Spark during the computation.

2) Block is the smallest unit of HDFS distributed file storage. When a file is about to be stored on HDFS, HDFS will physically split the file into several data blocks for distributed storage according to the block size set by the user. When a file is less than a block size, it still needs a whole block but does not occupy the entire block space. Therefore, for a large number of small files, each small file needs to occupy a separate block. When Spark SQL reads these small files, it needs to get the small file information from the NameNode of HDFS one by one, the total time T_{total} required in this process is:

$$T_{total} = n * t(t > 0) \tag{2}$$

where t is a fixed value which is the time to get the information of a small file, n is the number of small files. Therefore, T_{total} is proportional to n, as a result, T_{total} will increase if n increase.

Besides, from the computation parallelism of Spark SQL, for a query, the average load for each node in ideal environment is:

$$B_i = \frac{T_1 + T_2}{n} (1 <= i <= n) \tag{3}$$

where T_1 is the tasks generated by Spark SQL traversing all the small files. Each time a small file is read, T_1 will add 1. T_2 is the tasks generated by Spark SQL itself during its computation process according to the query conditions, n is the number of compute nodes, and B_i is the load of the i-th compute node. For a query of Spark SQL, T_2 is a fixed value, in the case of n fixed, B_i will increase as T_1 increase, that is, the more small files are read means the more tasks for each compute node are generated; in the case of T_1 is fixed, that is, the number of small files is fixed. B_i will decrease as the number of compute node increases, that means more compute nodes we have, less tasks of each computing node are generated. Therefore, in the computation of massive small files, both adding the compute nodes in the cluster or merging small files can reduce the computational load of each node. Obviously, the cost of the former is much larger than the latter.

It can be seen from the above that it is a better strategy to merge small files if we want to improve its access efficiency.

4. Local Merge Storage Model

4.1 Ideas of Model Design.

From the above analysis and research, this paper proposes the Local Merge Storage Model (LMSM) for merge the massive small files whose main ideas go as follows: A merge and storage module are created in the client of Hadoop to cache the small files. In this module, a directory is created for each type of small files, which generated in the time interval T are cached under the related directory, then start Spark at T intervals to merge the cached small files, as is shown in Fig. 4.

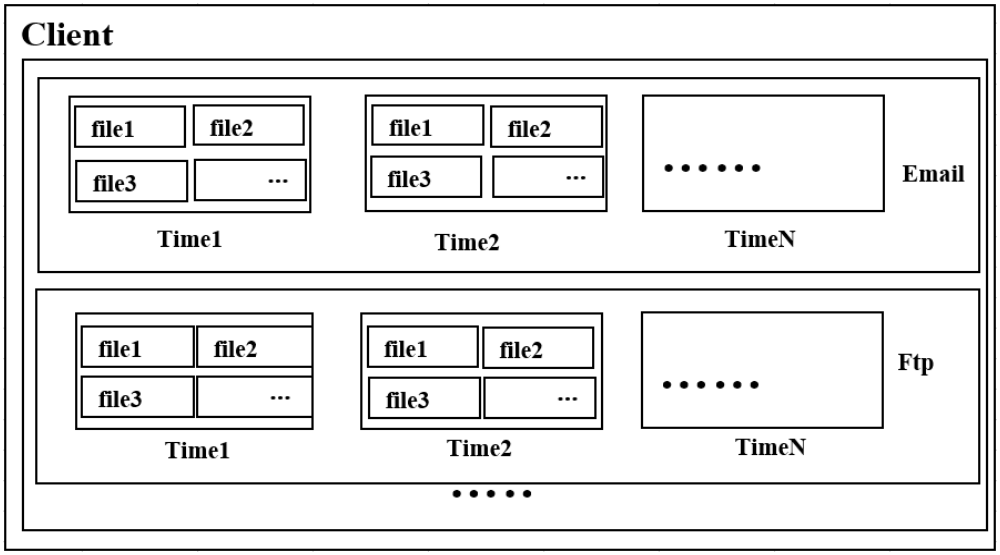

Fig. 4 LMSM Diagram

4.2 Selection of File Storage Format.

Parquet is a columnar storage format developed by Twitter and Cloudera for the Hadoop ecosystem. It is the best supported columnar storage format of Spark SQL. According to Xiaopeng Li，Wenli Zhou's research[[14]], under Spark SQL for the same query, the query speed of Parquet is 3-6 times that of ORC, 8-16 times that of RCFile, and 11-24 times that of Sequence File. Their experimental dataset is the traffic industry data, which includes the user's mobile phone number, terminal type, user IP and other information, and is very similar to the data features of the 4G IAC dataset. Based on the above analysis, Parquet is finally selected as the storage format of the files' merge, and the compression algorithm is the default Snappy compression.

4.3 Description of Model Process.

Firstly, the collected data is classified to get information such as its type and time. The algorithm is as follows:

File classification algorithm:

Input: small file to be uploaded, local directory to cache small files

Output: the location of the small file in the model

```
classificationFile(file，cacheDir){
    if(illegal(file)){
saveToIllegalDir();
} else {
//get file type and time through file name
    type=getTypeByFile(file);
    time=getTimeByFile(file);
    dstDir=cacheDir+"/"+type+"/"+time+"/"
    // determine if the destination directory exists
    if(!dstDir.exists()){
        dstDir.mkDir();
    }
    cache(file,dstDir+file);//cache the collected file
}
}
```

Then is the merge upload module. In this module, Spark is used to traverse the small files in the time interval T, then merge and convert these files into Parquet file and save to HDFS according its type. After the upload is completed, the source file will be backed up. The algorithm is as follows:

File merge and upload algorithm:

Input: directory to cache small files locally, array of small file types, spark

Output: HDFS directory
Merge upload algorithm:
Input: directory to cache small files locally, array of small file types, spark, backup directory
Output: HDFS directory

```
while(true){
mergeAndSaveToHdfs(cacheDir,type[],spark,bakDir);
sleep(T);
}
mergeAndSaveToHdfs(cacheDir,type[],spark,bakDir){
//traverse the file in different type
    for(int i=0; i<type.length;i++){
        fileType=type[i];
    files[]=listFiles(cacheDir+"/"+fileType);
        for(int j=0; j<files.length-1; j++){
    targetDir=files[j];
    time=getTimeByFile(targetDir);
hdfsDir="..."+"/"+fileType+"/"+time+"/";
    mergeToHdfs(spark,targetDir,hdfsDir); //merge the files in this period to HDF
backup(targetDir,bakDir+"/"+fileType+"/"+time); //back up the uploaded file and delete it in LMSM
        }
    }
}
```

The whole model flow is shown in Fig. 5.

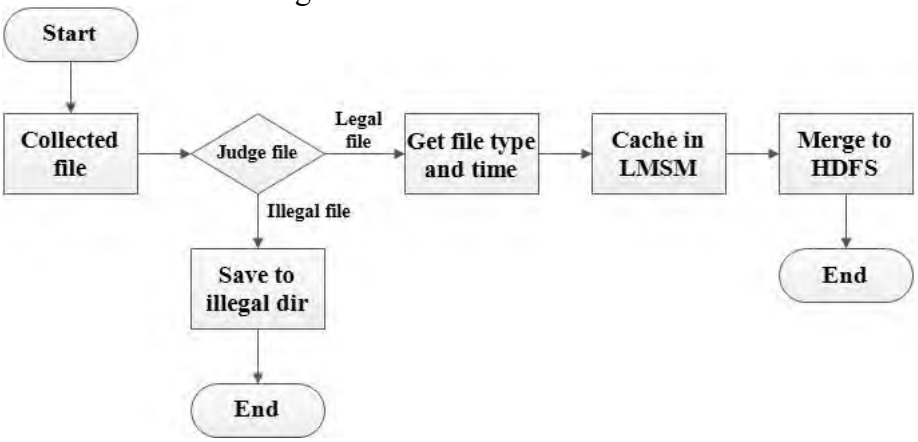

Fig. 5 File Cache and Merge Flow Diagram

5. Experiment and Result Analysis

5.1 Introduction of Experimental Environment.

The experimental environment for our work was based on the Windows Server 2008 R2 on Dell PowerEdge R720 server. VirtualBox was installed in this server and five virtual machines were created in VirtualBox, where the server memory was 128G, CPU was Intel E5-2650 with 32 cores and 2.6GHZ. The operating system for each virtual machine was CentOS7, and memory was 8GB, 10 cores, 455GB hard disk. Based on the 5 virtual machines, Hadoop and Spark clusters were created. The first virtual machine was the NameNode of the Hadoop cluster and the Master of the Spark cluster, while the remaining four were DataNodes and Workers. The version of the Hadoop was 3.0.0 with the 128MB block size and 3 replica of each block. The version of Spark and JDK were correspondingly 2.2.0 and 1.8.0_131.

In our experiment, the dataset was the Email data of IAC, which included 9 fields, and 44592 small text files generated within two months were selected for this experiment. The size distribution

of dataset is show in Fig. 6.

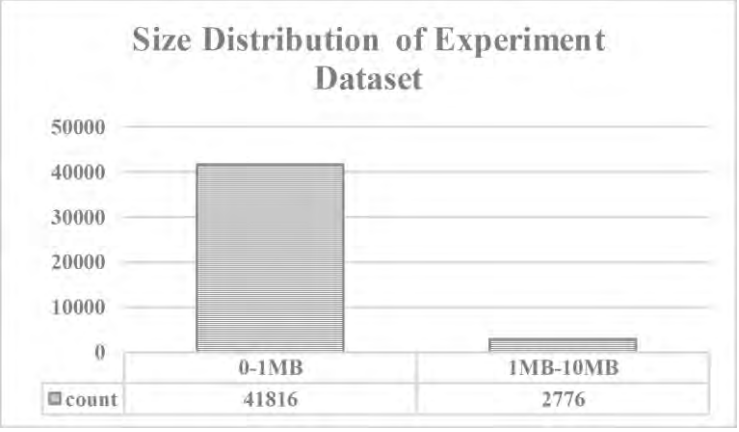

Fig. 6 Size Distribution of Experiment Dataset

5.2 Experimental Steps.

In our experiment, 10000, 20000, and 30000 small files were selected randomly from the 44592 small files for the comparative experiment. In the first experiment (E1), small files were uploaded to HDFS directly; In the second experiment(E2), small files were merged to a large Parquet file and stored to HDFS; Considering that data analysis was often required in hours for 4G IAC business analysis system, therefore, the third experiment (E3) used the time field of the file as the partition field, merged the files into Parquet in one hour, and then saved to HDFS by partition. Finally, six commonly SQL statements were selected for testing like Table 1:

Table 1 Types of Query

No	SQL
Q1	select count(*) from table;
Q2	select * from table where imsi = '46007***';
Q3	select imsi, apn, areaName, TcpLinkState from table where imsi='46007***';
Q4	select count(1) from table where imsi like '46007%';
Q5	select TcpLinkState, count(*) from table group by TcpLinkState;
Q6	select company,TcpLinkState, sum(ResponseDelay), count(*) from table where imsi ='46007***' and time > 1499155200000 and time < 1499529600000 group by company, TcpLinkState;

As is shown in Table 1, Q1 is used to count the number of rows in the table, Q2 is used to query all the information of the user whose IMSI is 46007***, Q3 is used to query partial information of the user whose IMSI is 46007***, Q4 is a fuzzy query, Q5 is a grouped aggregation query, and Q6 is used to query one user's information within five days, involving grouping, aggregation, filtering and other operations. For each data set, the average query time for Q1 to Q6 is shown in Fig .7:

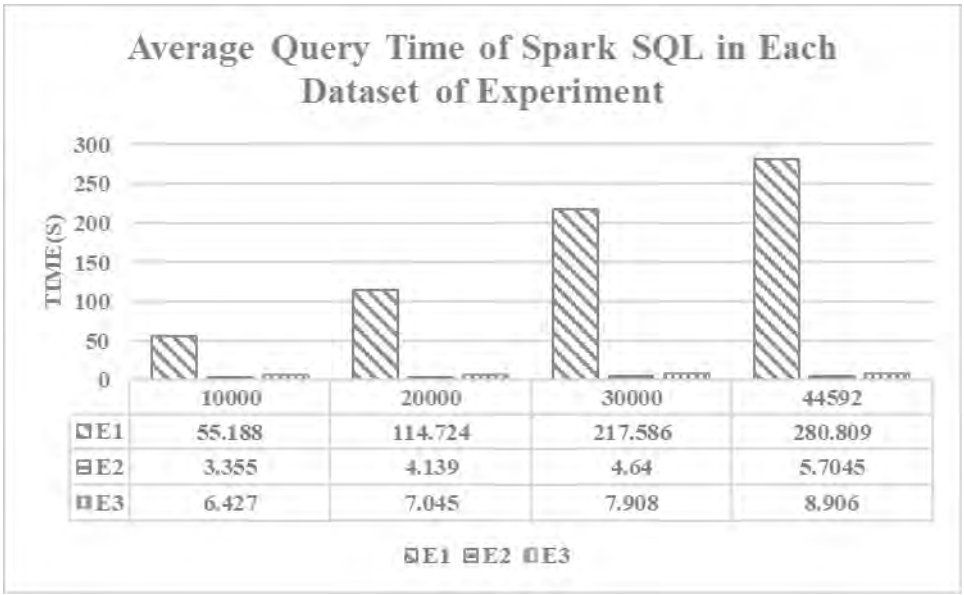

Fig. 7 Average Query Time of Spark SQL in Each Dataset of Experiment

As we can see from Figure.7 the query efficiency has been significantly improved after merging massive small files.

5.3 Analysis of Experimental Results.

The following is a detail analysis of the query of 44,592 small files. The dataset's block and storage distribution on HDFS is shown in Fig. 8 and Fig. 9.

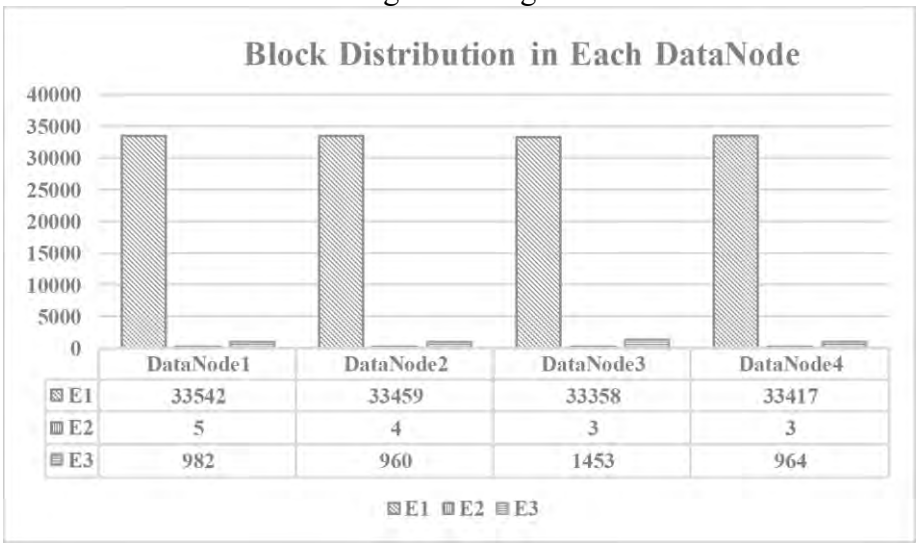

Fig. 8 Block Distribution in Each DataNode

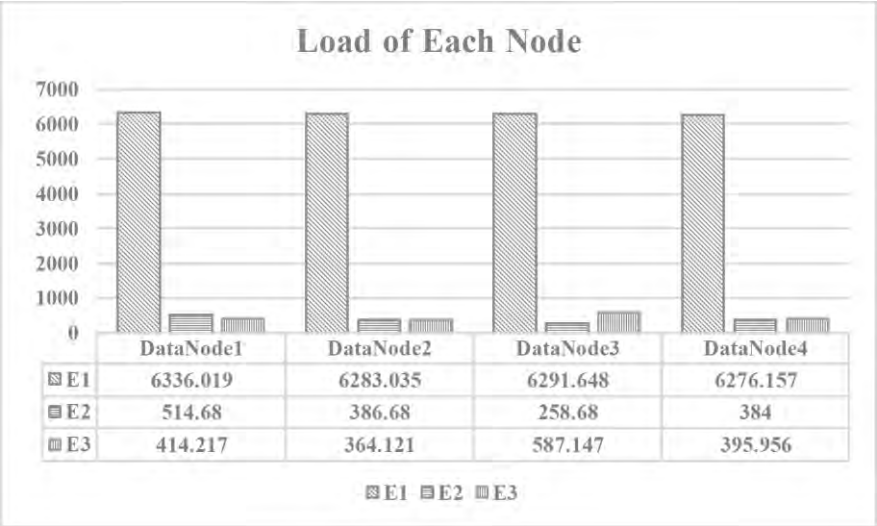

Fig. 9 Load of Each DataNode

As can be seen from Fig.8 and Fig.9, there is a significant reduction in block distribution and load of each node after these small files are merged and converted. The detail query time of this dataset is shown in Fig. 10:

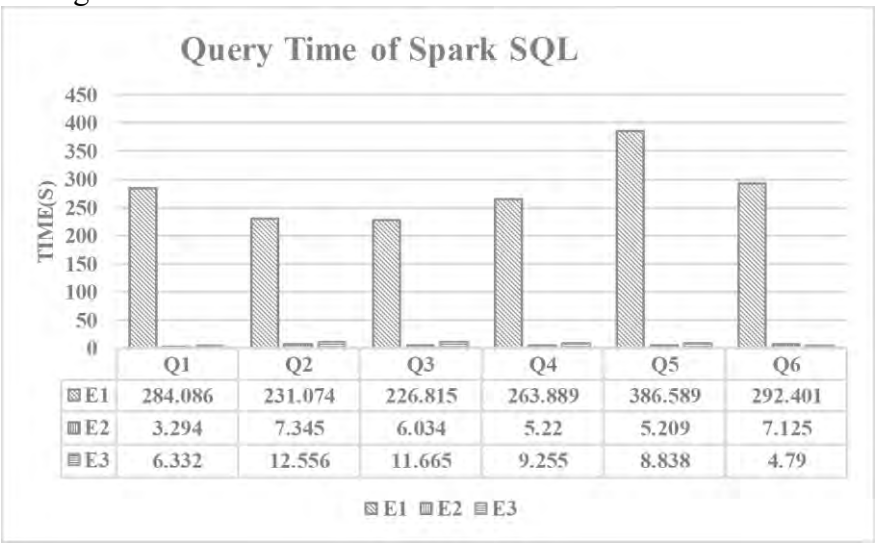

Fig. 10 Query Time of Spark SQL

Spark provides a web client to monitor the detail running information for a Spark application. Table 2 shows the statistics of the number of Tasks in each query Spark, and Table 3 shows the statistics of the data size of each query Spark.

Table 2 Tasks of Spark in Each Experiment

experiment / SQL	E1	E2	E3
Q1	44592	21	1503
Q2	44592	20	1502
Q3	44592	20	1502
Q4	44593	21	1503
Q5	44792	220	1702
Q6	44792	220	339

Table 3 Data Size of Spark Read in Each Experiment

experiment / SQL	E1	E2	E3
Q1	8.8 GB	81 KB	2.1 MB
Q2	8.8 GB	456.1 MB	536.6 MB
Q3	8.8 GB	285.2 MB	399.3 MB
Q4	8.8 GB	218.4 MB	335.11 MB
Q5	8.8 GB	1.36 MB	5.3 MB
Q6	8.8 GB	289.5 MB	5.3 MB

The following will analyze the query efficiency of Spark SQL in terms of the compute task and the input data size of Spark in each query.

For the first experiment, as can be seen in Table 2 and Table 3, Spark SQL will read all the small files for computation in any query, so the number of the tasks and the total input data are very large. As a result, the query efficiency is very low.

In second and third experiments, taking Q2, Q4, Q6 as examples: for Q2, E2 and E3, they will scan the full table. Because E3 is partitioned storage, Spark SQL will traverse files in all partitions at first, so the number of its task is more than E2, which makes its query efficiency lower than E2; for Q4, Spark SQL in E2 will only read the IMSI field for computation, while E3 will traverse all the partition at first then IMSI field will be selected for computation, which makes its query efficiency lower than E2. For Q6, E2 will still read all the data in the corresponding field in the file for computation, while for E3, due to the created partition, Spark SQL will read the fields that match the corresponding file under the partition condition instead of the full table scan, which makes the amount of data read in E3 much smaller than E2 and the query efficiency higher than E2 even when its tasks are higher than E2.

From the above experimental results and analysis, after the massive small files are merged, the compute task and input data size of Spark SQL have been significantly reduced. As a result, its query efficiency are improved significantly. It can also be seen from E2 and E3 that E2 has higher query efficiency for full table scan queries but E3 performs better than E2 if we want to get the information in a certain period, such as Q6. It can be predicted that with the amount of data increasing or the time range of the query reducing, the advantage of the partition will be more obvious. It is often necessary to analyze the information in units of hours in 4G IAC business System, which is very suitable for partition storage.

6. Summary

This paper introduces and analyzes the relationship between HDFS distributed storage and Spark SQL computing at first. For the problem of low query efficiency of Spark SQL reading massive small files, LMSM is proposed to merge and store 4G IAC small files based on its features. After merging these small files, the efficiency of Spark SQL has got a significant improvement. However, this paper only optimizes the storage of the data itself, and it does not consider how to store the merged data in a distributed manner, so it can be evenly distributed and stored in each DataNode of the Hadoop cluster to make full use of Spark SQL's advantages, which is also the focus of research in the future work.

Acknowledgements

West Project of the National Social Science Fund,17XFX013.

Demonstration Projects of Technology Innovation and Applicationof Chongqing Science and Tec hnology Committee. cstc2018jscx-msybX0332.

References

[1]. Ramesh D, Patidar N, Kumar G, et al. Evolution and analysis of distributed file systems in cloud storage: Analytical survey[C]// International Conference on Computing. 2017.

[2]. Smith C, Albarghouthi A. MapReduce program synthesis[J]. Acm Sigplan Notices, 2016, 51(6):326-340.

[3]. Apache Hadoop.http://hadoop.apache.org/

[4]. Gittens A, Rothauge K, Wang S, et al. Alchemist: An Apache Spark, MPI Interface[J]. 2018.

[5]. Bin Liao, Tao Zhang, Jiong Yu, et al. Optimization of collaborative filtering algorithm based on DAG Spark scheduling. ACTA SCIENTIARUM NATURALIUM UNIVERSITATIS SUNYATSENI. 5.2017, 56(3):46-56..

[6]. Renner T, Müller J, Thamsen L, et al. Addressing Hadoop's Small File Problem With an Appendable Archive File Format[C]// Computing Frontiers Conference. 2017.

[7]. Gao K, Mao X. Research on massive tile data management based on Hadoop[C]// International Conference on Information Management. 2016.

[8]. Sethia D, Sheoran S, Saran H. Optimized MapFile based Storage of Small files in Hadoop[C]// IEEE/ACM International Symposium on Cluster. 2017.

[9]. Meng Li, Sheng Cao, Zhiguang Qin, et al. Storage Optimization Method of Small Files Based on Hadoop. Journal of University of Electronic Science and Technology of China. 2016, 45(1):141-145.

[10]. Yuwan Gu, Wenwen Wang, Yuqiang Sun, et al. Optimization of massive small files storage and accessing on HDFS. Application Research of Computers. 2017, 34(8):2319-2323.

[11]. Tong Zheng, Weibin Guo, Guisheng Fan, et al. Research on Optimization Method of Merging and Prefetching for Massive Small Files in HDFS. COMPUTER SCIENCE. 11.2017, 44(b11):516-519.

[12]. Boufea A, Finkers R, Van Kaauwen M, et al. Managing Variant Calling Files the Big Data Way: Using HDFS and Apache Parquet[C]// IEEE/ACM International Conference. 2017.

[13]. Plase D, Niedrite L, Taranovs R. Accelerating data queries on Hadoop framework by using compact data formats[C]// Advances in Information, Electronic & Electrical Engineering. 2017.

[14]. Li X, Zhou W. Performance Comparison of Hive, Impala and Spark SQL[C]// International Conference on Intelligent Human-Machine Systems and Cybernetics. IEEE, 2015:418-423.