Optimal Design Method of Merging Disk Files Based on Hot Data

Lin Qian, Mingjie Xu, Jun Yu, Weichao Li, Hao Yuan, Jingbin Ren, Zhu Mei, Hengmao Pang
1State Grid Electric Power Research Institute (NARI Group Corporation), Nanjing, China
2State Grid GanSu Information&Telecommunication Company Lanzhou, China
3Huawei Technologies Co., Ltd Nanjing, China

*Corresponding author e-mail: qianlin@sgepri.sgcc.com.cn

Abstract. In recent years, with the rapid development of big data technology, users are more and more inclined to solve the problems of large amount of data and complex business scenarios with big data platform. When the system is faced with the scene of data hotspot and frequent data modification, it will produce a lot of useless data and cause data hotspot problems, which make a lot of access to the level of disk storage. Unreasonable data allocation of HFile will affect I/O performance and reduce system availability. This thesis proposes a comparison strategy based hot data. According to the frequency of data access, this strategy change the selection method of merging subsequence of Exploring Compaction Policy, so as to achieve a more suitable effect for hot data query business.

1. Introduction

With the rise of big data industry, centralized single machine storage or master-slave cluster storage is no longer the first choice for large-scale data processing in industry. With the rapid development of big data technology, the ideas of the three papers of Big table, MapReduce, and GFS have been widely adopted. Many open source organizations started to research and then implemented a set of open source big data processing frameworks. The Hadoop framework, led by Hadoop components, has gradually become a big data ecosystem with many big data components including HBase, Zookeeper, Spark and other contributions from many open source communities and individuals [1]. HDFS is Hadoop Distributed File System, a distributed file system based on Hadoop, the core modules include HDFS and MapReduce [2].

HDFS uses a master-slave architecture, and its master and slave nodes have meticulous fault tolerance mechanisms. One name node [9] and two data nodes can form a minimal Hadoop distributed file system. Name node is the core node in Hadoop cluster, it is usually set up with hot and cold dual-machine backup in industry. The name node is responsible for managing the namespace in HDFS and storing the index of HDFS data block access. The naming operation of Hadoop file system is usually performed in the name node.
MapReduce [3] is a parallel programming model. Hadoop MapReduce computing framework can support offline batch processing applications. The core idea of MapReduce is to segment data, collect the calculation results of multiple data segments, and merge to generate the final results.

In the Hadoop framework, the most common database for large-scale datasets is Hadoop Database, HBase. HBase is based on the columnar storage model [4]. Its query speed is very fast under large-scale data sets, but it is not compatible with SQL. HBase is derived from the Google paper Bigtable by Chang et al: A Distributed Storage System for Structured Data. Bigtable provides big data storage and query services based on GFS. Similarly, HBase relies on the Hadoop file system and provides basic functions similar to big data storage [5]. HBase, as a columnar database, is quite different from relational databases such as MySQL. It provides a storage model based on column family [6], which is compatible with structured data and unstructured data.

The design of HBase Compaction always pursues a balance point. On the one hand, it is necessary to ensure the basic effect of Compaction, on the other hand, we don’t want to bring serious I/O pressure. However, at present, there is no design strategy that can be applied to all application scenarios and all data sets. Therefore, in version 0.96, HBase made some adjustments to the architecture and provided the interface of the Compaction plug-in. Users only need to implement these specific interfaces to customize specific Compaction strategies according to their own application scenarios and datasets [7-12].

Because the big data system is faced with the scene of data hotspot and frequent data modification, which is dynamic and frequent, it will produce a lot of hot data. When the access volume is too large, the access of these hot data will break through HBase’s two-level cache and reach the HFile level. In this case, the unreasonable organization of HFile containing hot data results in a sharp increase in data's reading and writing I/O, which reduces the availability of system.

This paper proposes a Compaction strategy based on hot data, which changes the selection method of merging subsequences of the ExploringCompactionPolicy according to the frequency of data access. The idea is to increase the field of access frequency in the memory location of HFile, and then the Compaction strategy is determined according to the access frequency. At the same time, it is also necessary to consider the key parameters of ExploringCompaction to make HFile with high access frequency perform Compaction less frequently, and HFile with low frequency perform Compaction more frequently.

2. Main work
Firstly, the queue, which records the recent access timestamp, with fixed length is added to the HStoreFile. The popularity of this HFile is assessed by the time value of the queue. Then, we need to rewrite the merge method based on existing one ExploringCompaction. Specifically, calculate the total popularity of the merge group when selecting the HFile merge group in ExploringCompactionPolicy, and select the most suitable merged group for Minor Compaction by referring to this parameter. HstoreFile is the mapping structure of HFile in the memory level. GetReader() method which stored in HStoreFile should be executed every time before doing HFile reading operation. Similar to the getReader() method, there are two methods to read HFile, which are getPreadScanner() and getStreamScanner(). These two methods focus on HBase storage and Compaction, which will be called in the HStore class. The scan operation of HBase cannot represent the real popularity in the hot-spot access scenario, so we don’t need to consider the case where the scan operation is performed only once to get a large amount of disk I/O. At the same time, although the getReader() method will be called in Minor Compaction and Major Compaction, the original HFile will be deleted after the call, so it will not affect the number of getReader() calls, nor will it affect the calculation of HFile popularity.

2.1. Algorithm Flow
(1) Declare a TreeSet <Long> treeSet in the HStoreFile class to record timestamp.

(2) When getReader() is called in the object, check the size of the TreeSet and compare it with the customized LIMITSIZE.

2
(a) If `treeSet.size() < LIMITSIZE`, execute `treeSet.add(currenttime)`. The `currenttime` is obtained by `System.currentTimeMillis()`.
(b) If `treeSet.size() >= LIMITSIZE`, `treeSet.pollFirst()` deletes the element with the smallest timestamp before executing `treeSet.add(currenttime).

(3) Modify the `ApplyCompactionPolicy()` method of `ExploringCompactionPolicy`, and select the HFile merge subqueries that meet the ratio condition, then weight the number of files contained in the subqueue, the total file size, and the total file heat to obtain the best subqueue to do Minor Compaction.

2.2. Calculation Method (Model)

The related terms and definitions used in this paper are shown in Table 1.

| Variable Name | Variable Meaning |
|---------------|------------------|
| `filenum`     | The number of HFiles in the current subsequence to be merged. |
| `filecapacity`| The total offset of the HFile in the current subsequence to be merged. |
| `dtimehot`    | Popularity of access in a single HStoreFile. |
| `cmpratio`    | A parameter that measures the priority value in the current subsequence to be merged. |

The formula of `cmpratio` is shown in equation (1) where `f()`, `g()`, and `h()` can be replaced by appropriate functions to obtain formula (2).

\[
\text{cmpratio} = \frac{f(\text{filenum})}{g(\text{filecapacity}) \times h(\sum \text{dtimehot})} \quad (1)
\]

\[
\text{cmpratio} = \frac{\text{filenum} \times \text{const}}{\ln(\text{filecapacity}) \times \sum_{i=0}^{\text{sublist.size}} \text{sublist(i).dtimehot}} \quad (2)
\]

The choice of function is determined by experiments. Such a combination of functions has better discrimination. Theoretically, merging as much file data as possible is the aim of HBase and is also a critical part which should be paid attention to; in this paper, we don’t pay much attention to `filecapacity`, representing, the total capacity of the file, which is relatively large in numerical terms; `dtimehot` is the core of this algorithm, which is given a higher weight.

This algorithm does not want to merge hot data with other data. The first reason is that the access will increase delay after the merge. The second reason is that Minor Compaction will affect the I/O efficiency of the currently selected HFile from the perspective of system availability. Therefore, a larger weight is given to `dtimehot`. `filenum` and `filecapacity` here have already been implemented in the `ExploringCompactionPolicy`, so we don’t need to reconstruct them.

`dtimehot` is obtained from the `treeSet` which holds the timestamp in the second step of the algorithm flow, and its calculation is shown in equation (3).

\[
\text{dtimehot} = \sum_{i=0}^{\text{LIMITSIZE}-1} \frac{\text{LIMITSIZE} - i}{\log \left( \frac{\text{currenttime} - \text{treeSet.get(i)}}{\text{treeSet.get(i)}} \right) + 1} \quad (3)
\]

Where `i` is the `i`-th element in the `treeSet`. Intuitively, `treeSet.get(i)` represents the iterative process of the iterator. `LIMITSIZE` represents the maximum length of the `treeSet`. The denominator represents how
fresh the data is. The newer the data, the smaller the denominator, and the larger this accumulation term; meanwhile, due to the characteristics of Java timing, `currenttime-treeset.get(i)` represents the number of milliseconds between them. So, we need to reduce the magnitude, and plus one in order to prevent the exception of illegal denominators. The numerator represents the weight of the accumulation term. Here, the data with a smaller `i` is given a greater weight, so that the data that is frequently accessed is more distinguished from the data that has been accessed only a few times recently.

The timestamp recording method of the file is shown in Figure 1. A rectangle represents an access, and the color of the rectangle represents the distance of the access time. The type of Long numbers in the rectangle are timestamps of Java system, which can accurately record the access time. Each file contains a set of timestamps.

![Figure 1. Timestamp recording method](image)

The setting of `LIMITSIZE` and `const` should be based on the actual business scenario. `const` can make `cmpratio` more numerically easier to express. `LIMITSIZE` is related to the system's actual access to the amount of data that penetrates the cache. It is set to 1000 in subsequent experiments. If this value is too large, you can choose to counter intermittently when performing `getReader()`.

3. Experiment Design and Analysis

3.1. Experiment Environment
The experiment environment of this thesis is built on a 6-node distributed cluster of Hadoop 2.6.5. The deployment environment in server is configured with 64-core CPU, 128G memory, 8-disc (1 system disk, the remaining data disks) hard disks, CentOS release 6.6 (final) Operating system, 2.6.32-504.el6.x86_64 system kernel, 6 or more clusters.

The experiment data is based on the blood pressure record data, heart rate record data, personal data, video and audio data, and wearable device data of armed police, which are stored in HDFS. The amount of data is about 20T.

3.1.1. Experiment Scheme 1: Build HBase system, and carry out comparison experiments between the merge strategy before improvement and the one after improvement. First, select the HBase cluster that has been used for a period of time to ensure that there is a sufficient amount of data in it.

The interval of Major Compaction should be set as 80 minutes to facilitate the verification of the experiment results. Then, insert data to HBase gradually and query hotspot data at 10-minute intervals.
Meanwhile, parameters in the HBase cluster are monitored to obtain the CPU utilization rate and write efficiency of every server.

3.1.2. Experiment Scheme 2: Clear the initial data in HBase, and then carry out comparison experiments between redundant-data-based Compaction strategy and the DateTieredCompactionPolicy merge strategy. In the following three days, data should be written to the system continuously. 100 thousand pieces of data should be added to HBase at 10-minute intervals and 100 thousand pieces of data should be queried in HBase at 4-hour intervals, and record the efficiency of querying data.

3.1.3. Experiment Scheme 3: On the basis of Experiment 1, experiment should carry out about stress test of before and after the improvement to measure read latency. With dividing the runtime data into three stages, run model in prototype system.

In the first phase, 200 thousand pieces of data were added to HBase, and delete 20 thousand pieces of data that were added previously, meanwhile query the latest 100 thousand pieces of data in five tables;

In the second phase, 500 thousand pieces of data were added to HBase, and delete 50 thousand pieces of data that were added previously, meanwhile, the latest 250 thousand pieces of data were queried in five tables;

In the third stage, 1 million pieces of data were added to HBase, and 10w pieces of data that were add before were deleted, while querying the latest 500 thousand pieces of data in five tables. Perform the experiments 5 times to take the average as result.

3.2. Experiment Result

3.2.1. Experiment Result 1: While increasing useful data continuously and deleting useless data, carry out query experiment to simulate the real scenario of hotspot data and achieve cache breakdown, which can make sure the operation of data querying reaches the level of HFile. It also amplifies the impact of Major Compaction operation. The experiment results are shown in Figure 2.

Figure 2 depicts that the improvement of read performance compared with the previous experiment to some degree.

At 10 minutes, both are stored in HFile, and the query speed is slow. At 20 minutes, the data stored in the first and second-level caches from the first 10 minutes is queried and the query speed is fast. At 40 minutes, HBase performed the Minor Compaction operation. At this time, different merge strategies before and after the improvement affected the data storage form.

Before the improvement, HFile was merged according to the ExploringCompaction method. The latest hotspot data can be easily merged into the previous HFile as it is stored in small file and the file generation time is short. Although the sequential Compaction operation is performed, the improvement of query speed may be unobvious or even the query speed slowed down. The improved strategy
guarantees that hotspot data, namely, data blocks that are frequently queried, will not be merged into large HFiles. Take the internal index organization form of HFiles in to consideration, it is bound to speed up some query speeds. At 80 minutes, HBase performed a compulsory Major Compaction operation, which makes HFiles in the same Store merged into a single file, and the query speed was greatly slowed. However, the efficiency of the read operation after the improvement is lower than that of the read operation before the improvement. The mechanism of Major Compaction should be considered, the operation of the Major Compaction under the original strategy is faster than that of the improved Major Compaction. The pause of RegionServer caused by Major Compaction affects query speed.

In practice, the period of Major Compaction is relatively long, usually one week or even longer. Operators and maintainers often choose to perform Major Compaction operations manually instead of the default Major Compaction cycle mechanism, which makes Major Compaction controllable. It can be avoided for the disadvantages about poor performance of the improved strategy after Major Compaction.

There are no obvious differences in the CPU utilization rate and data writing speed of the HBase cluster server, which is following the query time before and after the improvement in this experiment. This is because the improvement strategy didn’t affect the merger trigger mechanism of the original ExploringCompaction. During the 80-minute Major Compaction period, the data writing rate of before and after the improvement decreased both, the CPU utilization rate of server is increased, and the fluctuation ranges of them were almost equal. Although there may be a gap in the number of HFiles under the Store, in general, the time of the merger occurs before and after the improvement and the CPU utilization rate were almost equal.

3.2.2. Experiment Result 2: Clear the data in HBase to simulate the characteristics of hotspot data query in the business of this system, and also adapt to the merge model of DateTieredCompactionPolicy merge strategy which group according to period of time. As the DateTieredCompactionPolicy strategy does not support deleting data, the query performance of hotspot data cannot be verified. The experimental results are shown in Figure 3.

![Figure 3. Comparison of query efficiency](image)

According to Figure 3, in the fourth hour, the merge strategy based on hotspot data was used in data query, which makes the query speed faster than that using the DateTieredCompactionPolicy merge strategy. It took different hours when Minor Compaction was carried out by native strategy and DateTieredCompactionPolicy strategy. In Date Tiered CompactionPolicy strategy, the execution condition of Minor Compaction is to reach threshold of Hfile amount which can be promoted to a higher level at least. However, in ExploringCompaction, the execution condition of Minor Compaction is memstore flush or CompactChecker triggers periodically in which the threshold is relatively low. The differences in the execution times of the Minor Compaction results in the improved strategy gaining more fine-grained HFiles, and the query result of the improved strategy is better than DateTieredCompactionPolicy.
In the later 8 hours to 48 hours, there was enough data in HBase, and the DateTieredCompactionPolicy merge strategy divided the data according to time period more clearly. The experimental results show the advantages of DateTieredCompactionPolicy in which HFiles is stored according to time periods, and the cost of time in querying is significantly lower than that using the improved strategy. After 52 hours, HBase stored a larger amount of data. The DateTieredCompactionPolicy merge strategy made the HFiles inconsistent in size, and oversized HFile blocks may appear, which greatly reduced the query speed.

The improvement strategy based on ExploringCompaction itself is to reasonably organize the relationship between the granularity and quantity of HFiles. The impact on the query speed caused by the increase in the amount of data is weakened to a certain extent compared with the former. On the whole, DateTieredCompactionPolicy is not suitable for the business situation of this system. This is mainly because DateTieredCompactionPolicy cannot perform Major Compaction and does not advocate data deletion operations. It is possible to modify the DateTieredCompactionPolicy to some degree, such as redefining the Major Compaction of this strategy, which can make the scope of Major Compaction reduced from the entire Store to a time zone.

3.2.3. Experiment 3: In order to research the impact of Minor Compaction on the query speed in the condition of large-data write operations in short time, this experiment performs insert operation, while the query process is also executed. It exacerbated the specific differences as the operation of pulling a large amount of data from HDFS to HBase before the experiment when the two strategies were merged, thereby magnifying the gap in read speed. When the amount of data added is less than 200 thousand, the query speed before and after the improvement is basically the same. This is because the large quantities of query data is cached in the memstore, and the two are also at the same storage level without going through Minor Compaction.

When the amount of added data reached 500 thousand, Minor Compaction occurred in the system after configuration. In the improved strategy, the hotspot HFile has not been merged in a large amount, and the strategy before improved made the recently added HFile merged with the previous HFile, which makes the difficulty of the data query is greatly increased. As the query data is a part of the data that was recently added, the improved strategy made a significant performance improvement. When adding more data to 1 million, more times of Minor Compaction occurred, and more file fragments appeared. At this time, the selection of the merged file has more obvious impact on the hotspot data query. Adopting the improved strategy is nearly 5 seconds faster than before. The experimental results are shown in Figure 4.

![Figure 4. Comparison of average query speed](image-url)
According to experiment scheme 3, it shows the read performance of hotspot data in this platform will be improved to a certain extent when using the hotspot data-based merge strategy for the file merge process in HBase, but it will also increase the server pressure during Major Compaction. With the increase Minor Compactions times, the read performance of hotspot data under this strategy is further improved than that of the original strategy; the read performance of hotspot data before and after the improvement of strategy can’t reach at the same level again until Major Compaction occurs.

4. Conclusion
This thesis gives a thorough analysis about the bottlenecks of query performance in big data platforms according to the native merge mechanism of HBase. Then it proposes a disk file merge strategy for the requirement of this platform, and gives experimental verification. This thesis puts forward a new disk file strategy, which takes the file popularity into consideration. It can improve the query efficiency of the hotspot data query scenario in this platform compared with merge mechanism that only considers the total file sequence size and the total number of files in HBase.

Acknowledgments
This work was financially supported by the science and technology projects of headquarters of State Grid Corporation (Research and application of non-stop state Adaptive upgrade technology for information system).

References
[1] Afzali M, Singh N, Kumar S. Hadoop-MapReduce: A platform for mining large datasets [C]. IEEE, 2016.
[2] Pandagala S. Hadoop-HBase for finding association rules using apriori MapReduce algorithm [C]. IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology, 2017: 795-798.
[3] Borthakurd. The Hadoop distributed file system: architecture and design [C]. Hadoop Project Website, 2007, 11 (11): 1-10.
[4] Shahabinejad M, Khabbazian M, Ardakani M. An Efficient Binary Locally Repairable Code for Hadoop Distributed File System [J]. IEEE Communications Letters, 2014, 18 (8): 1287-1290.
[5] George L. HBase: The definitive guide: Random access to your planet-size data [C]. Cambridge: OReilly Media, 2011.
[6] Fengd, Zhu L, Zhang L. Research on improved apriori algorithm based on MapReduce and HBase[C]. Advanced Information Management, Communicates, Electronic and Automation Control Conference. 2017: 887-891.
[7] Shafer J, Rixner S, Cox A L. The Hadoop distributed filesystem: Balancing portability and performance [C]. IEEE, 2010: 122-133.
[8] Gema B, Jason J, David C. Social big data: Recent achievements and new challenges [J]. Information Fusion, 2016 (28): 45-59.
[9] Chandarana P, Vijayalakshmi M. Big data analytics frameworks [C]. Circuits, Systems, Communication and Information Technology Applications (CSCITA), 2014 international conference on. IEEE, 2014: 430-434.
[10] Zhou W, Li R, Yuan S, et al. MetaSpark: a Spark-based distributed processing tool to recruit metagenomic reads to reference genomes [J]. Bioinformatics, 2017, 33 (7).
[11] Mundy J, Thomthwaite W. The Microsoft data warehouse toolkit: with SQL Server 2008 R2 and the Microsoft Business Intelligence toolset [M]. John Wiley & Sons, 2011.
[12] Bontempo, Charles, Zagelow, et al. The IBM data warehouse architecture [J]. Communications of the Acm, 1998, 41 (9): 38-48.