A New Data Access Mechanism for HDFS

Qiang Li$^{1,2}$, Zhenyu Sun$^{1,2}$, Zhanchen Wei$^{1,2}$ and Gongxing Sun$^1$

$^1$ Institute of High Energy Physics, Beijing, China
$^2$ University of Chinese Academy of Sciences, Beijing, China

E-mail: liqiang88@ihep.ac.cn

Abstract. With the era of big data emerging, Hadoop has become the de facto standard of big data processing platform. However, it is still difficult to get legacy applications, such as High Energy Physics (HEP) applications, to run efficiently on Hadoop platform. There are two reasons which lead to the difficulties mentioned above: firstly, random access is not supported on Hadoop File System (HDFS), secondly, it is difficult to make legacy applications adopt to HDFS streaming data processing mode. In order to address the two issues, a new read and write mechanism of HDFS is proposed. With this mechanism, data access is done on the local file system instead of through HDFS streaming interfaces. To enable files modified by users, three attributes including permissions, owner and group are imposed on Block objects. Blocks stored on Datanodes have the same attributes as the file they are owned by. Users can modify blocks when the Map task running locally, and HDFS is responsible to update the rest replicas later after the block modification finished. To further improve the performance of Hadoop system, a complete localization task execution mechanism is implemented for I/O intensive jobs. Test results show that average CPU utilization is improved by 10% with the new task selection strategy, data read and write performances are improved by about 10% and 30% separately.

1. Introduction

In the era of big data, the amount of data is explosively growing. IDC predicts that by 2020 the digital universe will reach 40 zettabytes (ZB), but only 3% of the potentially useful data in 2012 is tagged, and even less is analyzed. With such massive data, how to effectively store and analyze is a huge challenge, big data processing technologies are emerging rapidly, and Hadoop [1] becomes the most popular data processing technology in the Internet domain.

Analysis of physical data in High Energy Physics is a typical big data mining application. In High Energy Physics, the volume and complexity of data had undergone a tremendous increase over the past decades. It means that data processing and analysis difficulty increase dramatically. In order to improve the efficiency of High Energy Physics data processing, Hadoop has been tried on High Energy Physics data storing and processing since 2009. For example, University of Nebraska has been trying to use HDFS [2] as storage system in Tier 2 of WLCG [3]. Fabian Glaser, researcher at the University of Iceland, proposed to use MapReduce instead of PROOF (Parallel ROOT Facility) to do parallel analysis in 2013 [4]. Because random access to event data is not supported by HDFS, the output of jobs can only be written to HDFS via temporary files. Jobs must write the output to the local File System first, and later upload it to HDFS.

The paper introduces the technical background first and then proposes a new data access mechanism for HDFS for High Energy Physics data processing. Our goal is to write the output of jobs to HDFS directly, and the user program can modify files which were stored on HDFS.
2. Background

Hadoop is a framework for the distributed processing of large data sets across clusters of computers using simple programming models. The Hadoop framework has two core parts: a distributed file system named HDFS and a distributed programming model named MapReduce [5]. They are based on Google’s two papers: The Google file system [6], MapReduce: simplified data processing on large clusters [7].

However, the HDFS is designed to process text data, only sequential read, write and append operations are supported. In HDFS, a file is split into one or more blocks, each block has three replicas by default. When a client creates an HDFS file, it computes a checksum of each block of the file and stores these checksums in a separate file called checksums file in the HDFS. The HDFS data read and write process are shown in figure 1. HDFS access data is through the Java streaming interface with no support of random write operation and file modification.

In HEP-like applications, the data are stored as objects in DST file, it’s not suitable to process HEP data the same way as to process text data. The general way to read data is through FUSE_DFS module when using Hadoop to analysis of HEP data, and write the result to HDFS via temporary local file. Test results show that the read performance is reduced by 10% through FUSE, write via temporary file resulting in one more data copy operation. There will be a lot of data copy operations under high concurrency, and it will result in very poor write performance.

3. Design and implementation

The new data access mechanism is only used to access local data, remote access to data is still using the original HDFS streaming interface.

![Diagram of Hadoop architecture](image-url)
requests from the HDFS clients. The Datanode also performs block creation, deletion, and replication
upon instruction coming from the Namenode. Blocks are stored on the Datanode’s Local File System.
Map/Reduce (M/R) task accesses file through HDFS streaming access interfaces, without knowing
where the file is.

The new data access method can make MapReduce programs read/write data on local File System.
In HEP, we set the block size (default 10GB) a little bigger than the file size when write file to HDFS,
so that the file has only one Block. For data reading and modification, Datanode provides API to get
Block absolute path on local File System, the MapReduce programs can read or modify the Block file
under the condition the Task executed locally. For data writing, the client (MapReduce programs) first
creates a file on Namenode, applies a new Block for the file, gets a temporary directory from the
Datanode, then writes the file under the temporary directory, calculates and generates the checksums
file, notifies Datanode and Namenode that the file write has completed.

3.1. Data Access
In the new data access mechanism, we developed the HDFSService module to communicate with the
Namenode and Datanodes. For data reading, we modify the Datanode source code so that Datanode
Daemon provides access API for the Blocks which were stored on it. The Mapper program can get the
file location through the API, then can read the data from local file system.

The data read process is shown in figure 3, the Client (Mapper program) can get the file path from
MapReduce framework, then calls HDFSService’s getFile() method to get the file storage path on this
Datanode. The getFile() method first calls the Namenode’s getLocatedBlock() method to obtain the
Block information, and then calls Datanode’s getBlockFile() method to get the Block absolute path on
this Datanode. Then the client can read the file from local File System. If the file does not stored on
this Datanode, the getFile() method returns NULL, the client would read the file through FUSE.

![Diagram of data read process](image)

**Figure 3.** The data read process.

According to the HDFS data storage policy, the first replica of the Block is stored on the client
machine when the client is one of the Datanodes. Therefore, we define the Block only has one replica
when the file is created, and the replica store on the Datanode that client running on. If the number of
replicas of the system configuration is greater than 1, the rest replicas should be produced by copy of
the first replica.

The data write process can be divided into seven steps, shown in figure 4, the client is a mapper or
reducer program, it can run on any slave node of Hadoop. The client creates a new file in the
Namenode's namespace by calling Namenode’s create file interface via the HDFSService (Step 1).
Then the HDFSService remote call Namenode’s addBlock() method applies a new Block object.
Namenode will create a new Block object and adds the BlockInfo to INodeFile. We add Permissions
(permission, owner, group) to the Block object when create the new Block object. And set the storage
location of the Block is the node where the client resides. After the file is created successfully, a
LocationBlock object (figure 5) will be returned (Step 2). Then the client calls HDFSService’s
getTmpPath() method to get the temporary storage directory, HDFSService’s getTmpPath() method
will get the temporary storage directory from Datanode. The Datanode first creates a new ReplicaInfo
object (figure 6) for the Block and adds the ReplicaInfo to structure Map<pool, Map<blockId,
ReplicaInfo>>. Then selects a volume for the Block, returns a Block temporary storage directory
(Step 3). The client writes the file to the temporary storage directory. The local File System of
Datanode is supports POSIX semantics, so can meet the client's random write requirements (Step 4). When the client writes complete, it calls HDFSService’s Complete() method to end the write. The Complete() method will modify the file name to Block name and calculate the checksum of the file, then notify the Datanode and Namenode that the file write has completed. The Datanode moves the Block file to the finalized directory, and sets Block file permissions through JNI [8]. The Namenode changes the file state to completed (Step 5). The Namenode will check the number of replicas, and starts the Block copy if not reach the number of replicas of the system configuration (Step6, Step7).

Figure 4. The data write process.

Figure 5. The LocatedBlock structure.  

Figure 6. The ReplicaInfo structure.

3.2. Data Modification

Modifying a file requires that one has permission to modify the file. In the original mechanism, the owner of the Block replica file on the Datanode is the user hdfs, only user hdfs has the right to modify the Block file on Datanodes. We add Permissions (permission, owner and group) to the Block object, shown in figure 7. The access rights for the replicas of the Block on the Datanode are the same as the file in the Namenode namespace. When client uses the Hadoop command to change the file permissions, the blocks Permissions are also changed. We add two queues for DatanodeDescriptor to keep the block which Permissions changed. The blocks will be added to these queues when their Permissions are changed. When the Namenode handles the heartbeat message from the Datanode, it gets the blocks from the DatanodeDescriptor and encapsulates it into a command returned to the Datanode. When the Datanode receives the Namenode’s command, it will change the Permissions of the Block replica files via JNI.
The file modification process is shown in the figure 8. The client first gets the LocatedBlock object according to file name, then gets the location of the Block from the Datanode, throws an exception if the location is null. So client can modify the Block file directly, regenerate checksums file and notify Namenode update the file information. The Namenode will refresh the file’s BlockInfo object (figure 4) and delete the old replicas on other Datanodes, then starts Block copy to produce the new Block replicas.

3.3. Fault Tolerance
For the write fault tolerance, we define two rules: Firstly, Namenode does not retain the uncompleted file, this means Namenode will delete the uncompleted file after an hour; Secondly, the temporary directory on the Datanode set timeout deletion mechanism, the file under this directory will be deleted if it is not accessed for a period of time. If the Datanode fails during the write process, the Map or Reduce tasks which execute the write operation failed too. MapReduce framework failure task retry mechanism will make the failure task to be re-executed. The file stored in the temporary directory will be deleted by Datanode after Datanode recovery. If the client fails, the file it has created on the Namenode and the temporary file on the Datanode will be deleted after a certain period of time according to the rules we have defined.
If an exception occurs while reading the file, it will cause the task execution to fail and MapReduce framework will re-execute the task. If an exception occurs while modifying the file, the modified file will be deleted because it does not match the checksums file.

4. Complete Localization Task Scheduling
To take full advantage of the Hadoop task local execution, a complete localization task execution mechanism is implemented on MapReduce 0.20. In MapReduce, the JobTracker associates Map task with the node (one of Hadoop slave nodes) based on the data split location processed by the Map task when the job is initialized. The Map task and node relationships are stored in the structure `Map<Node, List<TaskInProgress>>`, a TaskInProgress object is a Map task. According to the MapReduce task scheduling strategy, TaskTracker first selects the Map task from the `List<TaskInProgress>` associated with the node on which the TaskTracker resides when applying for a task. If the List is NULL, a Map task is selected from other nodes. When the Map task fails, the failed task is added to queue failedMaps, which has the highest priority. Failed tasks are no longer under consideration of data locally, they can be executed on any node.

We modified the Hadoop task scheduler source code to let the TaskTracker only get Map task from the node it resides. If the Map task fails, it will be associated with the node which the data split resides. This ensures the data analysis jobs are executed locally.

5. Evaluation
The system is evaluated by analyzing the data from BESIII experiment. The system is implemented on HDFS 2.6.0 and MapReduce 0.20. The testing environment is set up with 8 nodes, each node with 12 cores CPU of 2.4GHz, 24GB memory and 1000M Ethernet connectivity. The HEP application for complete localization task scheduling testing is the program of BESIII offline software [9] called Rhopi events analysis. The new data access mechanism test program named Event is one test program from High Energy Physics data process tool ROOT [10].

Figure 9 shows the test results of job running time of the original task scheduling algorithm and the complete localization task scheduling algorithm. Using complete localization task scheduling algorithm, the job running time is decreased by 10% compared to the original task scheduling algorithm.

![Figure 9. The job running time.](image)

Figure 10 shows the file read time of using HDFSService and FUSE. The results show that the read performance of using HDFSService is improved about 10% than FUSE.

![Figure 10. The file read time.](image)

In HEP, the normal way to write the file to HDFS is via temporary file. First write the file to local file system, then upload the file to HDFS. Figure 11 shows the file write time of using HDFSService and via temporary file. The results show that the write performance using the HDFSService is improved by more than 30%. Figure 12 shows the file write performance when executing concurrently. We test it by way of multiple Map tasks, each Map writes 500000 Events (37GB). We call the scenario as case A when writing data via temporary file and the scenario as case B when writing data through HDFSService. The test results show that the performance of case A decreases linearly as the
number of processes increases. When the number of Maps is 12, the case B is 2.5 times faster than the case A. This is because in case A there will be an additional data copy operation, the operation will take up a lot of disk IO resources.

![Figure 11. The file write time.](image)

![Figure 12. Concurrency test.](image)

6. Conclusion
The new data access mechanism for HDFS meets the needs of random access data to HDFS in these legacy applications. This method effectively improves the data access efficiency of Hadoop in High Energy Physics. This has great significance to extend Hadoop application domains, especially in High Energy Physics.

The complete localization task scheduling may cause the cluster load imbalance because of uneven data distribution. In HEP, data analysis is Dataset-based. The next work is to study Dataset-based data load balancing method of HDFS, so that the data of the Dataset can evenly distributed to the entire cluster, and Map tasks can be evenly distributed over the cluster.

7. Acknowledgment
This work was supported by the National Natural Science Foundation of China (NSFC) under Contracts No.11375223.

References
[1] White T, Cutting D. 2009 Hadoop: the definitive guide. O’reilly Media Inc Gravenstein Highway North, 215(11):1 - 4.
[2] Borthakur D. 2007 The hadoop distributed file system: Architecture and design. Hadoop Project Website, 11(2007): 21.
[3] Bockelman B. 2009 Using Hadoop as a grid storage element. Journal of Physics: Conference Series, 180, 1.
[4] Glaser F, Neukirchen H, Rings T, et al. 2013 Using MapReduce for High Energy Physics Data Analysis. Int. Conf. on Computational Science and Engineering. IEEE, Sydney, Australia, 1271-78.
[5] Holmes A. 2012 Hadoop in Practice. Manning Publications Co.
[6] Ghemawat S, Gobioff H, Leung S T. The Google file system. 2003 Acm Sigops Operating Systems Review, 37(5):29-43.
[7] Dean J, Ghemawat S. 2004 MapReduce: Simplified Data Processing on Large Clusters. Conf. on Symposium on Operating Systems Design & Implementation, Berkeley, USA, 107-113.
[8] Liang S. 1999 Java Native Interface: Programmer's Guide and Reference. Addison-Wesley Longman Publishing Co. Inc.
[9] Li W, Liu H, Deng Z, et al. 2006 THE OFFLINE SOFTWARE FOR THE BESIII EXPERIMENT. ResearchGate.
[10] Rademakers F, Brun R. 1998 ROOT: an object-oriented data analysis framework. Nuclear Instruments & Methods in Physics Research, 1998(1–2):81-86.