International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 06 Issue: 2 | Feb 2019 www.irjet.net p-ISSN: 2395-0072

A Novel Approach to Process Small HDFS Files with Apache Spark

Priyanka N Dukale[1]

[1]*Department of Computer Engineering, SPPU University / Vishwabharati college of Engineering, Ahmednagar, India*

Abstract - *Hadoop is an open source distributed computing platform and HDFS is Hadoop Distributed File System. The HDFS has a powerful data storage capacity. Therefore, it is suitable for cloud storage system. However, HDFS was originally developed for the streaming access on large software; it has low storage efficiency for massive small files. To solve this problem, the HDFS file storage process is improved. The files are judged before uploading to HDFS clusters. If the file is a small file, it is merged and the index information of the small file is stored in the index file with the form of key-value pairs. The MapReduce based simulation shows that the improved HDFS has lower NameNode memory consumption than original HDFS and Hadoop Archives (HAR files). This memory consumption can be optimized significantly if MapReduce based file processing is reduced by Spark based file processing. Thus, it can improve the access efficiency as well.*

Key Words: **Hadoop, Cloud storage, Map Reduce, Apache Spark, Name Node/Data Node**

1. INTRODUCTION

The Hadoop Distributed File System (HDFS) is the primary storage system used by Hadoop Applications. HDFS is a distributed file system that provides high-performance access to data across Hadoop clusters. Like other Hadoop-related technologies, HDFS has become a key tool for managing pools of big data and supporting big data analytics applications.

HDFS is typically deployed on low-cost commodity hardware, so server failures are common. The file system is designed to be highly fault-tolerant, however, by facilitating the rapid transfer of data between compute nodes and enabling Hadoop systems to continue running if a node fails. That decreases the risk of catastrophic failure, even in the event that numerous nodes fail.

When HDFS takes in data, it breaks the information down into separate pieces and distributes them to different nodes in a cluster, allowing for parallel processing. The file system also copies each piece of data multiple times and distributes the copies to individual nodes, placing at least one copy on a different server rack than the others. As a result, the data on nodes that crash can be found elsewhere within a cluster, which allows processing to continue while the failure is resolved.

HDFS is built to support applications with large data sets, including individual files that reach into the terabytes. It uses a master/slave architecture, with each cluster consisting of a single NameNode that manages file system operations and supporting Data Nodes that manage data storage on individual compute nodes.

HDFS is originated primarily to tackle the problem of handling large files. As per core design, HDFS processes large files with exceptional performance. However, problem arises while processing large number of small files. Performance of HDFS decreases significantly while handling such usual cases.

Various approaches have been proposed to address this issue through heterogeneous aspects. Some of them have been discussed in next section as well. A novel approach is proposed in this paper on a top of all these existing approaches. This approach looks towards brand new Spark technology as a more optimal way to process HDFS files with the in memory computation capabilities of Spark.

2. LITERATURE REVIEW

2.1 Harball Archive

Harball archive contains the metadata entry for the index of file it contains. This index serves as a meta-meta data layer for the data in the archive, causing slight overhead in referencing files. File referencing request goes through the metadata of the archive to the index of metadata that the archive contains. The overhead of file referencing is negligible as it is done in main memory. [1]

2.2 HDFS I/O Speed Optimization

HDFS is designed to store large files and suffers performance penalty while storing large amount of small files. This performance penalty can be reduced drastically by optimizing HDFS I/O speed of small files based on the original HDFS. The basic way is let one block save many small files and let the datanode save some meta-data of small files in its memory, this will reduce the read and write request received by the namenode; furthermore, by sorting files with directory and file name when reading and writing files will further optimize increase the speed of reading. [2]

2.3 Prefetching Technique

Access patterns play crucial role in performance if HDFS is being accessed by heterogeneous users. These performance issues have the remedy of an efficient Prefetching technique for improving the performance of the read operation where large number of small files is stored in the HDFS Federation. This Prefetching algorithm learns through the files access patterns progressively. Even though some existing solutions incorporated prefetching, it was purely based on the locality of reference. The proposed solution based on the file access patterns improves the read access time by 92 percent compared to the time taken with the prefetching based on the locality of reference and 94 percent compared to the time taken without prefetching. The files are prefetched into the cache on webserver, which eliminates cache coherence problems. [3]

2.4 New Hadoop Archive (NHAR)

NHAR redesigns the architecture of HAR in order to improving performance of small-file accessing in Hadoop. Since reading file from HAR need to access 2 indexes which affect access performance To resolve this problem, the architecture of HAR has been modified to minimize the metadata storage requirements for small files and to improve the access performance. This approach can achieve obvious improvements on small I/O performance. [4]

2.5 Enhanced HDFS

In enhanced HDFS, the performance of handling interaction-intensive tasks is significantly modifications to the HDFS are:

(1) Changing the single namenode structure into an extended namenode structure;

(2) Deploying caches on each rack to improve I/O performance of accessing interactive-intensive files; and

(3) Using PSO-based algorithms to find a near optimal storage allocation plan for incoming files.

Structurally, only small changes were made to the HDFS, i.e. extending single namenode to a hierarchical structure of name nodes. However, the experimental results show that such a small modification can significantly improved the HDFS throughput when dealing with interaction-intensive tasks and only cause slight performance degradation for handling large size data accesses. [5]

3. PROPOSED SYSTEM

The proposed approach highlights the usability of Spark far faster file processing. The architectural model can be divided into three layers i.e. User Layer, Data Processing Layer and Storage Layer.

3.1 User Layer

This layer provides user with an ability to perform certain operations on HDFS les. User layer gives end user an ease of access to add/ update/ delete les of HDFS. User interface of Ambari can be used to access the HDFS in a user friendly manner. Ambari has an authentication-oriented mechanism to protect HDFS les from unauthorized access.

3.2 Data Processing Layer

This layer represents an operational unit of the complete architectural model. This unit can be split into four sequential parts as explained below:-

3.2.1 File Judging Unit

This unit categorizes an input le based on its size. It determines whether an input file is small or big against the predefined le size threshold value. If the input le size value is below threshold level, it is sent to le merging unit, whereas for above threshold value, file is straightaway sent to HDFS client for processing it in a traditional HDFS way, as the conventional HDFS has best possible capability to process big size HDFS.

3.2.2 Spark based File Processing Unit

This unit receives files small size les from File Judging Unit, calculates file size and starts creating incremental oset values until the pre-defined block size exceeds. Spark is used here for le retrieval and in-memory oset calculation. The calculated files are the passed to Spark Based File Merging Unit.

3.2.3 Spark based File Merging Unit

After receiving computed les, this unit performs merging of small computer les using datasets of Spark. This merging operation refers predefined block sizes and offset values to assure creation of optimal le block sizes.

3.2.4. HDFS Client

File blocks having reached oset size are received by HDFS client and then stored in HDFS. This blocks are accompanied by the big sized les, already landed into HDFS directly from File Judging Unit.

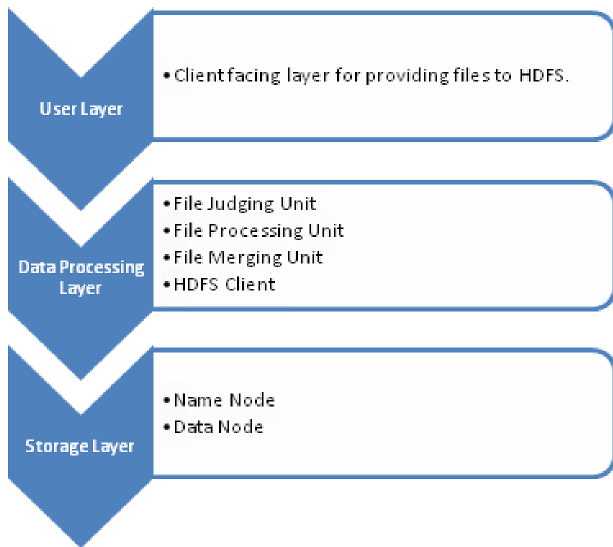

Fig -1: Proposed System

4. SYSTEM ARCHITECTURE

A description of the program architecture is presented.

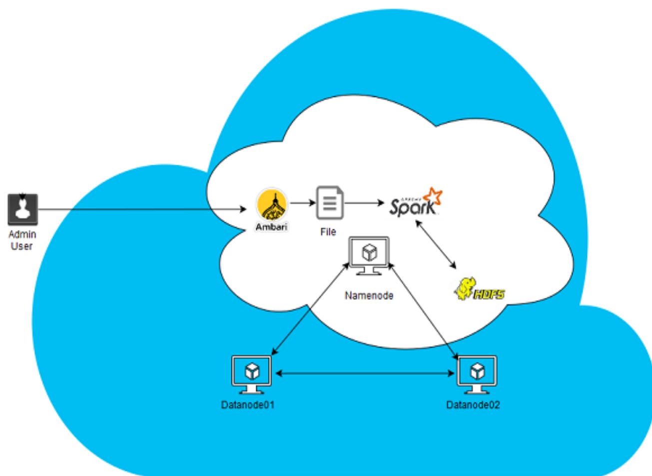

Fig -2: System Architecture

The overall project architecture comprised of three cloud instances i.e. one namenode and two datanodes. All the cloud instances are capable to communicate with each other through password-less SSH mechanism to have seamless communication. HDP repositories are pre-loaded on namenode. Service of HDFS(Hadoop File System) is up and running on namenode.

Admin user is able login to namenode instance through interactive user interface of Ambari service. User provides file to the namenode using this interface. Once file is received, Spark-based project code is invoked. It starts analysing file received, using File Judging Code.

If file size is greater than or equal to the predefined threshold value, it is categorized as big file and then sent directly to HDFS storage. But if le size falls below this predefined threshold value, it is marked as small size file, which is sent to File Processing Code to set the file-block oset value. This value remains as a reference while the le is being merged by File Merging Code. This merging activity keeps going on for further incoming files until the le-block oset value becomes greater than or equal to predefined threshold value. Once this value reached/ exceeded, the generated le block is uploaded to HDFS storage.

All this processing is performed by Spark code through in-memory computation. The computation is distributed across namenode and datanodes to enhance the processing speed, response time and performance without compromising on efficiency of file processing.

5. RESULTS

5.1 Table -1: Results

File Size	Time taken by conventional Hadoop System	Time taken by Spark Hadoop System
57.3 Mb	102.74sec	50.43sec
110 Mb	200.65sec	101.23sec
40 Mb	80.32sec	25.21sec

5.2 Snapshot of Result

```
[13514998532377]Execution Started.
log4j:WARN No appenders could be found for logger (org.apache.hadoop.metrics2.lib.MutableMetric
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
[13617746093182]Execution Completed.
Total Execution Time (seconds) : 102.747560805
```

Result with conventional haddop system

```
.8/12/23 10:57:05 INFO Executor: Finished task 98.0 in stage 0.0 (TID 98). 1
.8/12/23 10:57:05 INFO TaskSetManager: Finished task 98.0 in stage 0.0 (TID
.8/12/23 10:57:05 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks h
.8/12/23 10:57:05 INFO DAGScheduler: ResultStage 0 (saveAsObjectFile at File
.8/12/23 10:57:05 INFO DAGScheduler: Job 0 finished: saveAsObjectFile at Fil
7399104875996]Execution Completed.
'otal Execution Time (seconds) : 50.434645043
.8/12/23 10:57:08 INFO SparkContext: Invoking stop() from shutdown hook
.8/12/23 10:57:08 INFO SparkUI: Stopped Spark web UI at http://192.168.43.25
.8/12/23 10:57:08 INFO DAGScheduler: Stopping DAGScheduler
.8/12/23 10:57:09 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMaste
.8/12/23 10:57:09 INFO Utils: path = C:\Users\lenovo\AppData\Local\Temp\spar
.8/12/23 10:57:09 INFO MemoryStore: MemoryStore cleared
8/12/23 10:57:09 INFO BlockManager: BlockManager stopped
```

Result with Spark

Fig -3: Screenshots of Results

6. CONCLUSIONS AND FUTURE SCOPE

Aiming at the low store and access efficiency on small files in HDFS cloud storage, the HDFS file stored process is improved. If the file is a small file by judging before uploading to HDFS clusters, it is merged and the index information of the small file is stored in the index file with the form of key-value pairs. The file storage and access efficiency is analyzed through Spark. The results show that the improved NameNode memory consumption of the

Spark based Hadoop Processing is the least. It can also gives 100 times faster performance than Map reduce. It can save the NameNode memory when storing the small files. Thus, the Spark based Hadoop Processing can optimize the access efficiency of small files.

REFERENCES

[1] A. K, A. R. A, S. M. C, C. Babu and P. B, "Efficient Prefetching Technique for Storage of Heteregeneous small files in Hadoop Distributed File System Federation," in Fifth International Conference on Advanced Computing (ICoAC), 2013.

[2] C. Vorapongkitipun and N. Nupairoj, "Improving Performance of Small-File Accessing in Hadoop," in 11th International Joint Conference on Computer Science and Software Engineering (JCSSE), 2014.

[3] G. Mackey, S. Sehrish and J. Wang, "Improving Metadata Management for Small Files in HDFS," in IEEE International Conference on Cluster Computing and Workshops, 2009.

[4] L. Changtong, "An Improved HDFS for Small Files," in 18th International Conference on Advanced Communication Technology (ICACT), 2016.

[5] M. A. Mehta and A. Patel, "A Novel Approach for Efficient Handling of Small Files in HDFS," in IEEE International Advance Computing Conference (IACC), 2015.

[6] P. Gohil, B. Panchal and J. S. Dhobi, "A Novel Approach to Improve the Perform

[7] X. Hua, W. Hao and S. Ren, "Enhancing Throughput of Hadoop Distributed File System for Interaction-Intensive Tasks," in 22nd Euromicro Intternational Conference on Parallel, Distributed, and Network-Based Processing, 2014.