International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 4 Issue: 2 315 - 319

Optimization Scheme for Storing and Accessing Huge Number of Small Files on HADOOP Distributed File System

L. Prasanna Kumar[1],
[1]Assoc. Prof, Department of Computer Science & Engineering.,
Dadi Institute of Engineering & Technology,
Visakhapatnam.
lpk.lakineni@gmail.com

Sampathirao Suneetha [2]
[2] M.Tech (CSE), 2nd year,
Andhra University College of Engineering, Andhra University, Visakhapatnam.
sampath.suneetha@gmail.com

Abstract:- Hadoop is a distributed framework which uses a simple programming model for the processing of huge datasets over the network of computers. Hadoop is used across multiple machines to store very large files, which are normally in the range of gigabytes to terabytes. High throughput access is acquired using HDFS for applications with huge datasets. In Hadoop Distributed File System(HDFS), a small file is the one which is smaller than 64MB which is the default block size of HDFS. Hadoop performance is better with a small number of large files, as opposed to a huge number of small files. Many organizations like financial firms need to handle a large number of small files daily. Low performance and high resource consumption are the bottlenecks of traditional method. To reduce the processing time and memory required to handle a large set of small files, an efficient solution is needed which will make HDFS work better for large data of small files. This solution should combine many small files into a large file and treat these large files as an individual file. It should also be able to store these large files into HDFS and retrieve any small file when needed.

Keywords: *Hadoop distributed file System, small file*

*****

I. Introduction:

Hadoop is an open source distributed framework which stores and processes large data sets and is developed by Apache Software Foundation. It is built on clusters of commodity hardware. Each single machine server stores large data and provides local computation which can be extended to thousands of machines. It is derived from Google's file system and MapReduce. It is also suitable to detect and handle failures. It can be used by the application which processes large amount of data with the help of large number of independent computers in the cluster. In Hadoop distributed architecture, both data and processing are distributed across multiple computers. Hadoop consists of the Hadoop Common package, HDFS and MapReduce engine. A Hadoop cluster consists of a NameNode and DataNodes. Function of NameNode is to manage the metadata of file system and the actual data is stored at the DataNodes.

It is obligatory to divide the data across different DataNodes when large amount of data is stored on a single machine. Distributed file systems are the ones which are responsible for the management of data storage over a network. The distributed file system used by Hadoop is called HDFS and it is a storage system. HDFS has been considered as a highly reliable file system. HDFS is a scalable, distributed, high throughput and portable file system programmed in Java for the distributed framework of Hadoop. HDFS has read-many-writeonce model that allows high throughput access, simplifies data consistency and eases concurrency control requirements. HDFS helps to connect nodes which are personal computers present in a cluster in which data is distributed. Then, the data files can be accessed and stored as an one seamless file system. HDFS work is done efficiently by distributing data and logic to on nodes for parallel processing. It is well grounded as it retains multiple copies of data and assigns processing logic in the event of failures on its own.

Hadoop also comes with the MapReduce engine. For writing applications, Hadoop MapReduce is used to process large data parallely on large clusters. A MapReduce job normally divides the input data into separate chunks. These chunks are then processed parallely by the map tasks. The framework classifies the results of the maps. These are fed as input to the reduce tasks. File system stores both the input and the output of the job. The framework manages scheduling of tasks, supervising them and also the unsuccessful tasks are reexecuted. It includes JobTracker and TaskTracker. Client applications sends requests of the MapReduce jobs to JobTracker and the JobTracker assigns these jobs to available task tracker nodes in the cluster. The JobTracker tries to keep the data and its processing in close proximity to each other. MapReduce processing need not be done in Java unlike Hadoop which needs Java base.

II. Background

A. Hadoop Distributed File System

The Hadoop Distributed File System provides several services like NameNode, DataNode, Secondary NameNode, JobTracker, TaskTracker, etc.

The NameNode is the main part of an HDFS. It keeps the metadata information which is the directory tree of all files present in the file system. It also tracks where the file data is kept across the cluster. However, the data of these files is not stored, but the metadata is stored. There is a single NameNode running in any DFS deployment. Namenode is the master of HDFS. It manages the slave

DataNode daemons to perform input output tasks at the low-level. It also manages and keeps the information about on which nodes the data blocks of file are actually stored, on what basis the files are split into file blocks, and the functioning of the distributed files ystem. In the HDFS Cluster, the NameNode is a single point of failure. Whenever client applications need to locate a file or say they want to add or copy or move or delete a file, they always talk to the NameNode. Then the NameNode responds to the valid and successful requests by returning a list of DataNode where the data actually resides.

In HDFS, DataNode stores data. Generally there are many DataNodes, with data replicated across them to recover the data in case of data failure. When the cluster is started, DataNode connects to Namenode and waits till the acknowledgement comes from the Namenode. For file system operations, the Datanode responds to requests from the NameNode. Once the NameNode has given the location where the data is stored, client applications can instantly talk to a DataNode. Whenever MapReduce tasks are submitted to TaskTracker which is near a DataNode, they immediately contact to the DataNode for getting the files. TaskTracker operations should generally be performed on the same machine where Datanode resides. This helps for the MapReduce operations to be executed in close proximity to the data. To duplicate the data blocks, DataNode communicate with each other and thus redundancy is increased. The backup store of the blocks is provided by the Datanodes. To keep the metadata updated, Datanodes constantly report to the NameNode. If any one DataNode crashes or becomes unreachable over the network, backup store of the blocks assures that, file reading operations can still be performed.

In HDFS, the Secondary NameNode acts as an assistant node to supervise the state of the cluster. In each cluster, there is one SecondaryNameNode. It resides on each machine in the cluster. The SecondaryNameNode differs from the NameNode in the context that, any realtime changes to HDFS are not received or recorded by this process. After definite time intervals defined by the cluster configuration, SecondaryNameNode communicates with the NameNode to get the instances of the HDFS metadata. It is a daemon that periodically wakes up, triggers a periodic checkpoint and then goes back to sleep. If the NameNode goes down, the SecondaryNameNode helps to lower the outage duration and data loss. If a NameNode fails and to use the SecondaryNameNode as the primary NameNode, we need to manually reconfigure the cluster.

The function of the JobTracker is to distribute the MapReduce tasks between particular nodes containing the data in the cluster. These nodes might be present in the same rack. Job tracker acquires jobs from the client applications. Once the code is submitted to the cluster, the JobTracker determines the enactment strategy by finding out which files to process. It then allocates nodes to different tasks and superintends all running tasks. To determine the location of the data, the JobTracker consults the NameNode. The JobTracker discovers TaskTracker nodes that available for processing the data. The work is then given to the located TaskTracker nodes. In case of task failure, that task is automatically instigated by the JobTracker on some other node. The number of retries has already been defined. The only JobTracker in Hadoop cluster is run on the master node.

The TaskTracker in the cluster is the one that receives tasks such as Map, Reduce and Shuffle operations from a JobTracker. The number of tasks that TaskTracker can accept is configured with a set of slots. To ensure that process failure does not bring down the task tracker, it manages a separate Java Virtual Machine (JVM) processes to do the work. It supervises these processes and captures the output and exit codes. It notifies the JobTracker whether the process completed successfully or not. After every fixed interval of time, the TaskTrackers also send out pulse messages to the JobTracker, to conform that the JobTracker is still alive.

B. Small File Problem

Normal approach for storing large number of small files includes directly storing the files in HDFS without any pre-processing done, this has many disadvantages, following are few amongst them:

1) In HDFS, if data in files is significantly smaller than the default block size, then the performance reduces drastically.
2) If small files are stored normally in HDFS, then it wastes a lot of space in storing metadata of all files. When small files are stored in HDFS, there will be lots of seeks and jumps from one datanode to other to get a file, which is ineffective data access method [12].

III. System Design

We have tried to eliminate the drawbacks of storing large number of small files in HDFS, so that the time required to read and write the files would be much less and also the metadata storage decreases significantly.

While writing the files into HDFS,first we have sorted the files and then stored them in order to get better prefetching. Our program stores the similar files (similarity of files is on the basis of their extension) zipped together into HDFS. Once the sorting has been done, we have zipped the small files till the size of zipped file i.e combined file is equal to 64MB or till the extension of small files change. This approach helps, as it leads to the concept of locality of reference i.e. user will refer similar types of files for later use. Storing files into HDFS requires less time since we are zipping the files. Therefore, for example, instead of writing thousands of small files into HDFS, we will write only few zip files into HDFS, each zip containing hundreds of small files. For the first read, a map gets created for each extension, then this map is searched for the given small file name and returns the combined file name .This combined file is copied to local machine. Since

whole combined file is being copied to local machine the similar files present in the zipped file are being prefetched. For the next reads, time required to get combined filename from small file name would be less since map is already created and search time in a map is o(1) i.e constant. If a similar small file is read then same combined file name is returned hence it takes further less time since the combined file is already present in local machine.

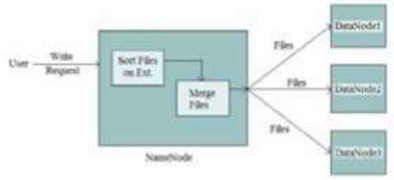


Fig. 1 : Write Operation

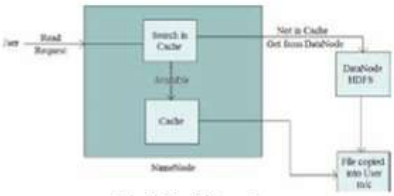


Fig. 2 : Read Operation

A. File Merging Algorithm

1) Start the program
2) Select the Write button to write the files into HDFS
3) Enter the file name (this file contains pathnames of all the files which you need to store in HDFS)
4) Our program reads the file ,sorts it on the basis of extension and stores it in output.txt file
5) initalisation of variable: index = 100 (some arbitary number)
6) Reads each line from output.txt
7) creates a zip file archive+index.zip
8) while(line != null)
9) if (sum(size of read files) is less than 64MB and extension doesn't change)
10) then a) : add files into zip
11) else
12) close zip
13) add entry into hashtable.txt on the basis of extension
14) copy zip file from local to HDFS
15) index++
16) create a new zip file archive+index.zip
17) end ifelse
18) read next line of output.txt
19) end while
20) close zip file
21) copy zip file from local to HDFS

B. File Reading Algorithm

1) Start the program
2) Select the Read button to read from HDFS
3) Enter the small file name
4) For the first read it creates a map by looking at the hashtable.txt created by merger program
5) For later reads it checks for the match in map (which requires O(1) time)
6) gets the archive index number
7) if (archive+index.zip is not in local machine)
8) copy the archive from HDFS to local machine
9) end if
10) read the small file from the zip present in local machine.

C. File Sorting Algorithm

1) Read the file contents from the filename (given by user)
2) Read a line
3) while (line ! = null)
4) split on . (split on extension)
5) interchange extension and name
6) add this to list
7) read next line
8) end while
9) use Colection.sort (in build java function) to sort list
10) for (each list item)
11) split on extension
12) intechange extension and name
13) write into output.txt
14) end for

/subsectionFile Searching Algorithm

1) find the smallest key which is greater than or equal to given target (small file name)
2) if (smallfilename lies within the range of { smallfilename1 } to { smallfilename2 g }
3) return the index
4) else
5) print file doesn't exist
6) exit program
7) end ifelse

IV. Evaluation & Results

A. Experimental Environment:

The multinode cluster had 4 nodes, one master and 3 slave nodes. Each of these machines had following configurations:

1) Processor: Intel Core i7-2600 CPU @ 3.10GHz
2. RAM: 4 GB
3. HDD: 500 GB
4. Operating System: Linux
5. Version: Ubuntu 12.04 LTS
6. Java Version: 1.6.0 24
7. Hadoop: 1.2.1 version
8. Eclipse IDE: Helios
9. Network Protocol: Secure Shell All the machines were connected through ethernet.

The time taken for read and write operations was measured for both the original HDFS and the proposed solution for single node as well as for multinode cluster. A set of 1000, 2000, 4000, 6000, 8000 and 10000 files. Here we have considered a mix of text and pdf files.These files

were first copied into HDFS and then read back to the local machine. The time taken to complete these operations was noted down and similar readings were obtained for the proposed solution[11]. This was repeated three times and an average of the results was calculated and used for analysis.

B. Single-Node Cluster

1) Read Operation: The results obtained for read operation are summarized below:

Comparison for Reading

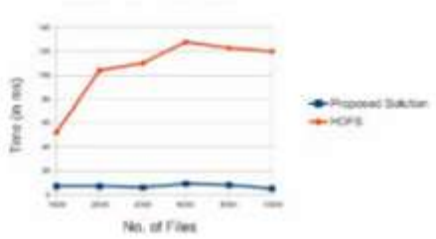


Fig. 3 : Time taken for read operation in single node cluster

2) Write Operation: The results obtained for write operation in single node cluster are summarized in a similar manner which is shown below:

Comparison for writing

The data obtained clearly indicates that the proposed solution is faster than the default original HDFS for read as well as write operation in single node cluster setup.

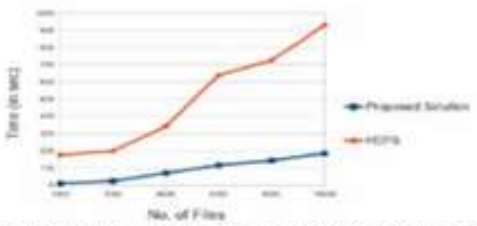
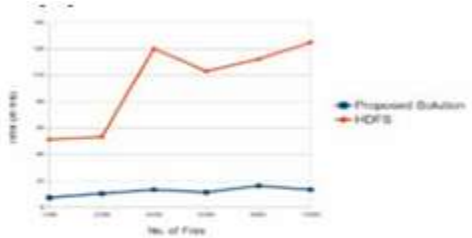


Fig. 4 : Time taken for write operation in single node cluster

C. Multi-Node Cluster:

1) Read Operation: In the following graph, we have compared the read time required in HDFS and in proposed solution for .txt files:



5: Time taken for reading .txt files in Multi-Node Cluster

In the following graph, we have compared the read time required in HDFS and in proposed solution for .pdf files:

Comparison for reading (pdf) files

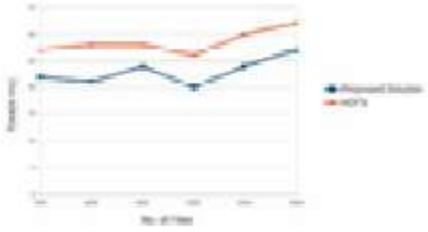
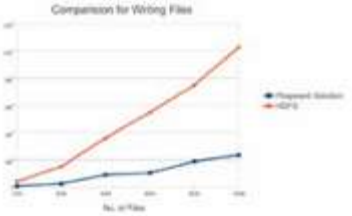


Fig. 6 : Time taken for reading .pdf files in Multi-Node Cluster

2) Write Operation:

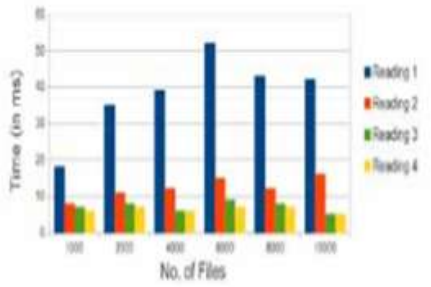
The results obtained for write operation in a multinode cluster are summarized in a similar manner which is shown below:



Time taken for Write operation in Multi-Node Cluster

D. READ Analysis:

The first read will take more time since the map is created but the next reads take significantly lesser time as depicted in the figure below:



V. Related Work:

Bo Dong1, Jie Qiu, Qinghua Zheng, Xiao Zhong, Jingwei Li and Ying Li in [1] designed a way of effective storage and access pattern for large number of small files in HDFS. For storing and accessing small files, correlations between the files and locality of reference remaining among small files in the context of PPT courseware are taken into account. In the first step, they tried to merge all the correlated small files of a PPT courseware into a single big file that can effectively reduce the metadata load on the NameNode. In the second step, they introduced a concept of two-level prefetching mechanism. Applying this mechanism, the efficiency of accessing small files is improved. The experimental results finally indicate that their presented design efficiently reduces burden on the NameNode. It also improves the efficiency of storing and accessing huge number of small files on HDFS. However, it does not take into account other types of files such as .txt, .pdf, .png, .odt, etc.

Chandrasekar S, Dakshinamurthy R, Seshakumar P G, Prabavathy B and Chitra Babu in [2] proposed a solution based on the works of Dong et al., where a set of correlated files is combined, as identified by the client, into a single large file to reduce the file count. An indexing mechanism has been built to access the individual files from the corresponding combined file. Efficient management of metadata for small files helps for greater utilization of HDFS resources. However, it does not sort the files on the basis of their extension which makes it difficult while reading similar types of files.

Yang Zhang and Dan Liu [3] proposed an approach for small files processing strategy and proposes files efficient merger, which builds the file index and uses boundary file block filling mechanism. It successfully accomplishes it's goal of effective files separation and files retrieval. Their experimental results clearly show that their proposed design has improved the storing and processing of huge number of small files efficiently in HDFS. However, it does not take into account file correlation mechanism which can reduce access time in HDFS. This solution can be enhanced further by effectively implementing file correlation mechanism.

VI. Conclusions

HDFS was originally developed for storing large files. When a large number of small files are stored in it, efficiency is reduced. The approach designed in this solution could improve small files storage and accessing efficiency significantly. Files are sorted on the basis of their extensions and then merged into zip files whose size does not exceed the HDFS block size. This helps to locate any small file easily. A file read cache has been established, so that the program can read a small file quickly. The experimental results show that the approach can effectively reduce the load of HDFS and improve the efficiency of storing and accessing small files. For 10,000 small files, the writing time is reduced by 80% and the reading time is reduced by 92% on a single-node cluster while for the multi-node cluster, the percentage decrease for writing is 77% and for reading text files is 89% and for .pdf files is 15%.

VII. Future Work

As for future work, this solution can be enhanced further to provide a more advanced file correlation framework. This framework should provide a mechanism to combine files of similar domain. Append operation can also be provided to add similar files into the existing combined files. It can also be extended for other types of file format.

References

[1] Bo Dong1, Jie Qiu, Qinghua Zheng, Xiao Zhong, Jingwei Li, Ying Li "Improving the Efficiency of Storing and Accessing Small Files on Hadoop: a Case Study by PowerPoint Files". In proceedings of Services Computing (SCC), 2010 IEEE International Conference, Miami, FL, July 2010, pp. 65 72.

[2] Chandrasekar S, Dakshinamurthy R, Seshakumar P G, Prabavathy B, Chitra Babu "A Novel Indexing Scheme for Efficient Handling of Small Files in Hadoop Distributed File System". In proceedings of Computer Communication and Informatics (ICCCI), 2013 International Conference, Coimbatore, Jan. 2013, pp. 1 8.

[3] Yang Zhang, Dan Liu "Improving the Efficiency of Storing for Small Files in HDFS". In proceedings of Computer Science and Service System (CSSS), 2012 International Conference,Nanjing, Aug. 2012, pp. 2239 2242.

[4] C. Shen, W. Lu, J. Wu and B. Wei, "A digital library architecture supporting massive small files and efficient replica maintenance", Proceedings of the 10th annual joint conference on digital libraries, ACM Press, QLD, Australia, June 21-25, (2010), pp.391-394

[5] Petascale Data Storage Institue, "NERSC file system statistics," World Wide Web electronic publication, Available: http://pdsi.nersc.gov/filesystem.htm, (2007).

[6] G. Mackey, S. Sehrish and J. Wang, "Improving metadata management for small files in HDFS", Proceedings of IEEE International Conference on Cluster computing, New Orleans, USA, August 31 - September 4, (2009).

[7] Shafer J., Rixner S. and Cox A., "The Hadoop Distributed File System: Balancing Portability and Performance", Proceedings of 2010 IEEE International Symposium on Performance Analysis of Systems & Software, White Plains, NY, USA, March 28-30, (2010), pp. 122-133

[8] J. Venner, "Pro Hadoop", Springer Press, (2009).

[9] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The Hadoop Distributed File System", Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies, Incline Village, NV, USA, May 3-7, (2010).

[10] "The Hadoop Distributed File System: Architecture and Design", available: http://hadoop.apache.org/common/docs/r0.20.1/hdfsdesign.html, (2010).

[11] "The major issues identified: The small files problem", available: http://www.cloudera.com/blog/2009/02/02/the-small-files-problem, (2010).

[12] A. Chervenak, J. M. Schopf, L. Pearlman, M.-H. Su, S. Bharathi, L. Cinquini, M. D'Arcy, N. Miller and D. Bernholdt, "Monitoring the Earth System Grid with MDS4", Proceedings of the Second IEEE International Conference on e-Science and Grid Computing. Washington: IEEE Computer Society, (2006).