Improving Latency for HDFS using IDecider Component

Jigar S. Rudani and Prof. Roy H. Campbell
Dept. of Computer Science
University of Illinois, Urbana Champaign
{rudani2, rhc}{at} illinois{dot}edu}

Abstract - HDFS is a distributed file system designed to be fault tolerant, highly scalable and distributed in nature. It is mostly suitable for Write Once Read Many applications where number of reads are more as compared to writes. Applications that run on HDFS need streaming access to their data sets. HDFS is designed more for batch processing rather than interactive use by users and the main emphasis is on data throughput not on latency. But if we can improve the latency then it will extend the application of HDFS from WORM to different type of applications. There are various approaches to improve the latency like, we can have entire data reside in memory or we can have most frequently used data into memory and others on disk instead of entire data resulting into better memory utilization. Based on this principle we have designed an approach which helps to improve the latency for HDFS. The idea is to provide different levels of storage like Memory, SSD and Disk. At top level it appears to be one logical storage block but physically it will be three levels of storage component. We are planning to design a component known as Intelligent Decider which will decide the storage location for files. IDecider will consider runtime characteristics of file like frequency of access, volume, time to run and other basic parameter. Based on this factor it will decide whether to keep file in Memory, SSD's or Disk. Integration of IDecider will help to improve the latency and also throughput of HDFS.

Keywords – HDFS, SSD, Memory, Distributed File system, Latency, Data Throughput

I. INTRODUCTION

Distributed File System is designed to store huge amount of data and provide data access to any requestor, distributed across the network. There are number of distributed file systems which solve this problem in different ways. NFS which is ubiquitous file system helps to solve this problem by providing remote access to a single logical volume stored on a single machine. But it has some limitations like amount of data stored is limited to one machine and also files in NFS reside on a single machine. This results into single point of failure and provides less reliability.

HDFS, which is an open source distributed file system. It is designed to be robust to a number of problems that other DFS's such as NFS are vulnerable to. HDFS is designed to store huge amount of data (terabytes to petabytes) by spreading across large number of machines. This achieves high reliability and throughput. HDFS also integrates well with Hadoop MapReduce allowing data to be read and computed upon locally. Many commercial companies like Yahoo, Google [4] and Facebook have deployed HDFS as their massive storage system. HDFS is a part of famous open source project Hadoop. Hadoop is mainly comprises of two component - HDFS and Map Reduce. HDFS is highly fault tolerant and runs on inexpensive commodity server. HDFS is mostly suitable for WORM (Write Once Read Many) type of applications where batch processing is the main goal. However HDFS is very scalable, reliable and fault tolerant. It fails to provide efficient support for applications which require random read access, as its basic storage system is disk. There are many ways to improve the latency. One such approach is to use RAM-HDFS [1]. The idea is to store the entire data into memory i.e. entire HDFS directory into memory and use disk for backup and failure recovery. However this approach has some drawbacks like

1. The storage capacity is limited to the total memory of all cluster nodes. Also, out of total memory only fraction on RAM is available for computation. Consider an example, we have 5000 commodity server and each contribute 30GB RAM-HDFS. So total available memory is 150TB. Considering the default replication factor we hardly have 50 TB which is not sufficient for large data sets.
2. If suppose entire RAM is occupied and new data is requested by user. We have to replace some data to make room for new process. What kind of data to be swapped out. There is no distinction between most frequently used data and last used data.

We have presented an approach to improve the latency and also better utilization of memory. We have planned to abstract the HDFS storage as three layers of storage – Memory, SSD [3] and Disk. The idea is to devise some logic to classify data based on runtime characteristics. Some of the characteristic include Frequency (How frequently file is being accessed), Time to run (Running time), Volume (Size of the file) and Access Pattern. Based on the statistical information collected, we can

identify what data needs to be stored at different level of storage. In this way we can improve the latency by keeping data which is frequently used in memory [13] [14]. This also helps to improve the memory utilization by keeping only useful data into memory. If suppose memory is full and we need to store new data. If we have space in SSD then we can swap data between memory and SSD. This will make room for new data into memory. If we don't have space in Memory and SSD then instead of running replacement policy for Memory and SSD, we can directly store data into disk as time required for two replacement policies is approximately equal to one disk write operation. The component which helps to decide about storage location for data is known as IDecider (Intelligent Decider)

The arrangement in this report is as follows: Section 2 gives an overview of HDFS architecture. Section 3 describes the approach we propose to improve the latency for HDFS. Section 4 describes evaluation of experiment Section 5 describes conclusion

II. HDFS ARCHITECTURE

HDFS is a master/slave architecture as shown in Fig 1. HDFS is designed to support applications that uses very large files. The three main component [5] [6] [7] which make HDFS suitable for such applications are

1. Name Node (Master Node)
2. Data Node (Slave Node)
3. Secondary Node

HDFS is Block Storage in nature [2]. Each file is broken into multiple blocks of size 64 MB (default) each. The block size is configurable. Each block is stored on a Data Node. It is also replicated to other Data Node. The replication factor is also configurable, by default it is 3.

Name Node
When DFS client needs to write/read data to/from HDFS, It needs to know about the location of data blocks in HDFS. Name Node (NN) basically stores the metadata about the data blocks. So once client is aware about the location, it can directly write/read data to/from desired data block. The responsibility of maintaining metadata resides on NN. The NN exposes the file system namespace and allows data to be stored on a cluster of nodes while allowing user to have single logical view of the file system. The NN manages all the File/Directory operations such as open, close, rename and so on. The Data Nodes (DN) are responsible for actual storage of file data. Some of the metadata information stored by NN are

1. File/Directory name and its location relative to parent directory
2. File/Directory Ownership/permission.
3. Replication factor.
4. File Name of individual blocks.

Each blocks is stored as a file in local file system of DN. The NN [8] [9] [10] doesn't actually store the location of blocks. During cluster startup it gets the information about the location from DN. The metadata is stored on the disk but is loaded into memory during cluster operation for faster access.

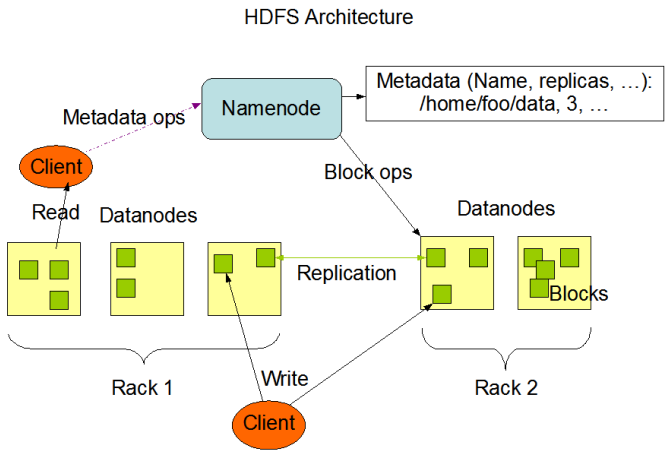

Fig. 1

Data Node
Data Node (DN) is slave node responsible for the actual storage of file data based on the instruction from NN. All file data actually flows through DN during write and read operation. DN are daemons which periodically send the heartbeat to NN. This enables NN to remain aware of health of each DN and not direct any client request to failed DN. During write operation DN receives data from client in small chunks. The writing of data is done in pipeline fashion. First DN receives data from client. It writes into its disk and also stream the data to other DN. The second data also writes the chunks received into its disk and streams down to other DN. After successful completion, each DN sends acknowledgment to previous one in pipeline fashion. When first DN receives acknowledgement, it sends the acknowledgement to the client and the client will send acknowledgement to NN. NN then commits the file after which, it is visible to client in file system namespace. During read operation client gets the list of blocks and their location from NN. Client then directly contacts the desired DN. DN serves the read request. In case of failure/corrupted file data, the read request is served with the replicated copy.

Secondary Name Node
Secondary Name Node is not failover node. NN maintains metadata in memory. All operations performed in Hadoop system are recorded into FsImage file which is used

during failure of NN. In order to avoid data loss, edit log file is maintained. The edit log captures all the updates performed during system operation. During restart, edit file content has to be rolled back to FsImage file. This affects the performance during startup. So Secondary Name Node was created to handle such situations.

III. IDECIDER APPROACH

We have discussed about the architecture of HDFS. To improve the latency for HDFS, We have proposed one approach. The approach is to consider the statistical characteristics of data while accessing it. In this section, we present the design and architecture of proposed solution

Architecture

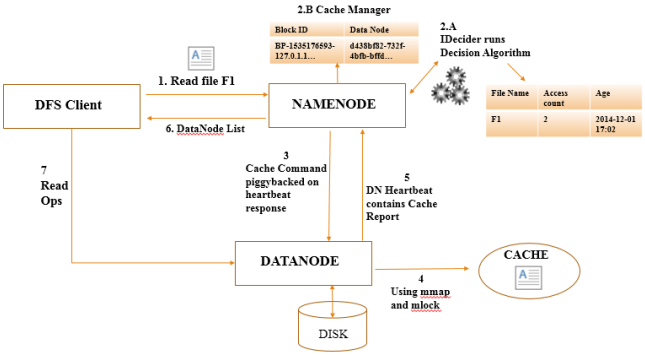

Fig. 2

Fig. 2 shows the architecture of IDecider in HDFS. IDecider is placed at Name Node. The advantage of having IDecider in Name Node is that IDecider can easily interact with Name space and Scheduler. Files in HDFS consist of blocks. The information about any particular file is contained in the HDFS namespace. Each block is replicated and stored on multiple Data Nodes. The mapping from block replicas to Data Nodes is stored in the INode data structure. Name Node has two critical table managed as HDFS Name space.

1) file name → block sequence (Namespace)
2) block replica → machine list (INodes)

The Namespace is persistent while INodes are not. They are created in the event of a NameNode restart with the help of block report send by Data Nodes. IDecider component keeps the track of statistical information about the file. It stores information like File name, Access count and Age field [19]. Access count is incremented every time file is read. Age field indicates how long the file is in memory or other storage location. When file is read first time the age is initialized to current time stamp. Suppose one file is being access frequently, but after certain rounds file is not accessed so we can swap out this file in case of RAM not available. We can achieve this by comparing the Age field with current timestamp. After comparing with current timestamp the LRU [18] list is prepared in descending order of difference. The file which has higher difference value is selected for eviction. The eviction process is shown in fig. 3 .After analysis, IDecider will inform name node to store file in memory or not.

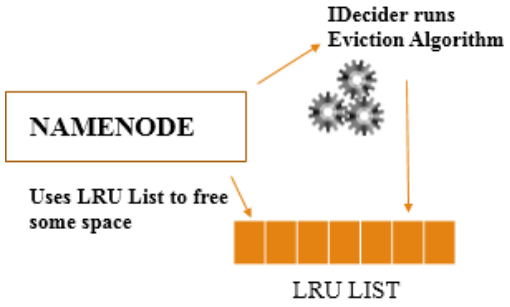

Fig. 3

Name Node maintains 2 new in-memory data structures [15] [17]

1. List of file directives which are yet to be stored in memory
2. Cache Manager which maps blocks to DNs that have the block in cache.

Name node also persists the request to the edit log. This lets the Name Node to re-cache the file on restart. As the rate of cache request is low (on the order of seconds between requests), so there is not a significant increase in edit log or fsimage size from this change [16]. After receiving, validating, and persisting a cache request, the Name Node uses Data Node heartbeat responses to tell the Data Nodes which replicas to cache and un-cache.

Data Nodes use the mmap and mlock system calls to pin blocks into memory. This requires that data node already has the block stored on disk. Data node also maintain cache map which includes information about data blocks which are cached. Based on information available in cache map cache report is send to Name Node. Once collocated with a memory replica, clients proceed with the normal HDFS read path without any extra RPCs.

Implementation

We have built a prototype using cacheadmin which is an api [11] implemented in hadoop 2.3.0. The cacheadmin command helps to specify the directive path (file/directory) which is to be cached. So based on the IDecider decision, cacheadmin api is called with the

required file name. The decision algorithm is as shown in fig. 4. As part of eviction process we are preparing LRU list as shown in fig. 5.

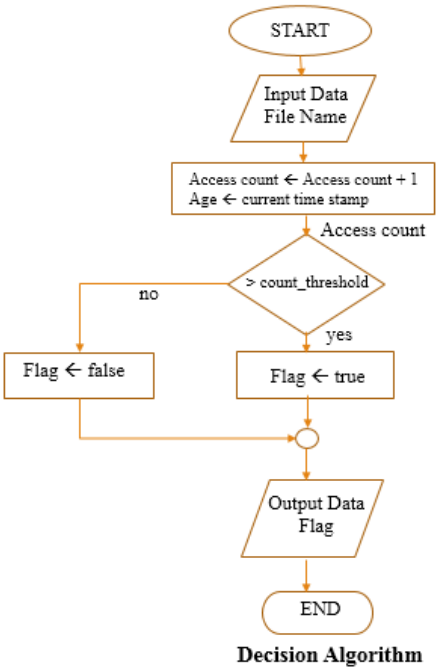

Fig. 4

Algorithm: - Decision Algorithm

Input: - File Name
Output: - Flag to indicate whether to cache data or not
Steps: -

1. Initialize the Age field to current timestamp and increment Access count by 1
2. Check whether Access count > count_threshold.
3. If yes then Flag ← True
4. Else Flag ← False
5. Output Flag value

Algorithm: - Eviction

Input: - File name
Output: - LRU List
Steps:-

1. Calculate the Age for given file name
2. Age ← abs (Age – current time stamp)
3. LRU List ← Sort in descending order of new Age value
4. Output LRU lists

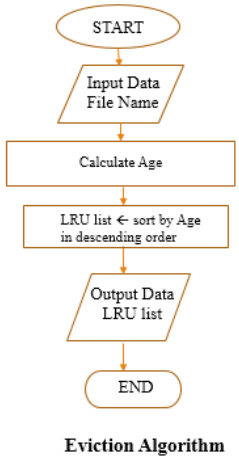

Fig. 5

IV. EXPERIMENT EVALUATION

Hadoop 2.3.0 is running on Single node setup on a machine which has Corei5-3210M CPU @ 2.50 GHz with 4 GB RAM and 500 GB IDE disk. Operating system used is Ubuntu 64 bit Linux OS. The number of mapper used is 3 and reducer's count varies. We have used workload which has paper titles from 5 different conferences. The title belongs to 5 different Computer Science domain – Information Retrieval, Data Mining, Theory, Machine Learning and Database.

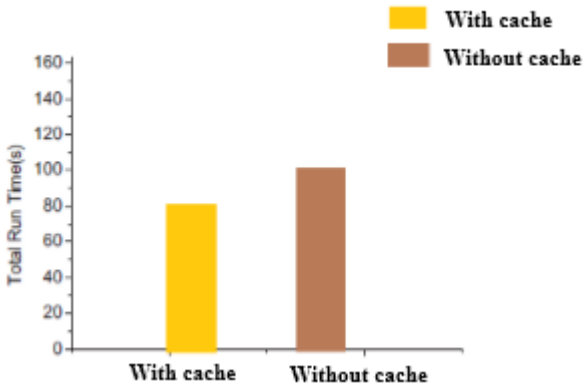

Fig. 6

We have use benchmark [12] wordcount and grep along with Apriori pattern matching algorithm. Apriori helps to find the patterns by counting the number of occurrences for each pattern. wordcount counts the number of occurrences of each word in the given dataset. Apriori along with wordcount is I/O intensive application that requires loading and storing a sizeable amount of data during pattern matching. Results is shown in fig. 6. We

have run the application 10 times and took average of the results. We have tested with cache and without cache. The total average time taken without cache was 104 seconds and with cache was 81 seconds. We have also collected cache statistics as shown in fig. 7. There was significant improvement in the results.

$ hdfs cacheadmin -listPools -stats inMemory
Found 1 result.

NAME	**inMemory**
OWNER	**adminuser**
GROUP	**adminuser**
MODE	**rwx r-x r-x**
LIMIT	**unlimited**
MAXTTL	**never**
BYTES_NEEDED	**4096**
BYTES_CACHED	**4096**
BYTES_OVERLIMIT	**0**
FILES_NEEDED	**1**
FILES_CACHED	**1**

 # Look at the data node stats
$ hdfs dfsadmin -report

Live data nodes:
Name: 127.0.0.1:50010 (local host)
...<snip>...
Configured Cache Capacity: 65536 (64 KB)
Cache Used: 4096 (4 KB)
Cache Remaining: 61440 (60 KB)
Cache Used%: 6.25%
Cache Remaining%: 93.75%
Last contact: Mon Dec 01 16:18:42 CST 2014

Fig. 7

V. CONCLUSIONS

We present the approach and evaluation of caching frequent data with the help of IDecider. We propose IDecider component that helps to distinguish between frequent and infrequent data. Our approach require only slight modification in Hadoop code. The experiments conducted shows decent result. In the future we are planning to run on real cluster with 5 nodes. We are also planning to use SSD along with Memory to cache data.

VI. ACKNOWLEDGEMENTS

We are thankful to Prof Roy Campbell for supporting our research idea and also thanks to System Research Group (SRG) group at UIUC.

IV. REFERENCES

[1] Yifeng Luo,Siqiang Luo, Jihong Guan,Shuigeng Zhou "A RAMCloud Storage System based on HDFS: Architecture, implementation and evaluation", Journal of Systems and Software Volume 86 Issue 3, March, 2013
[2] Pro Apache Hadoop, 2nd Edition, By Sameer Wadkar, Madhu Siddalingaiah,Jason Venner ISBN13: 978-1-4302-4863-7 Publication Date: September 15, 2014
[3] Xiangyong Ouyang, Nusrat S. Islam, Raghunath Rajachandrasekar, Jithin Jose, Miao Luo, Hao Wang and Dhabaleswar K. Panda,"SSD-Assisted Hybrid Memory to Accelerate Memcached over High Performance Networks*",2012 41st International Conference on Parallel Processing
[4] Sanjay Ghemawat, Howard Gobioff,Shun-Tak Leung. The Google file system. Proceedings of the 19th ACM Symposium on Operating Systems Principles. New York: ACM Press, 2003:29-43
[5] Tom White. Hadoop:The Definitive Guide[M]. United States of America: O'Reilly Media, Inc. 2009.
[6] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[7] https://developer.yahoo.com/hadoop/tutorial/module2.html
[8] http://hortonworks.com/hadoop/hdfs/
[9] http://hadoop.apache.org/docs/r0.18.3/hdfs_design.html
[10] http://static.googleusercontent.com/media/research.google.com/en/us/archive/gfs-sosp2003.pdf
[11] http://hadoop.apache.org/docs/current/api/
[12] http://hadooper.blogspot.com/
[13] Shubin Zhang, Jizhong Han ; Zhiyong Liu ; Kai Wang, Shengzhong Feng, Inst. of Comput. Technol., Chinese Acad. of Sci., Beijing, China, "Accelerating MapReduce with Distributed Memory Cache". Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on 8-11 Dec. 2009
[14] Jing Zhang, Gongqing Wu ; Xuegang Hu ; Xindong Wu, Dept. of Comput. Sci., Hefei Univ. of Technol., Hefei, China, "A Distributed Cache for Hadoop Distributed File System in Real-Time Cloud Services". Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on 20-23 Sept. 2012.
[15] http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-common/releasenotes.html
[16] https://issues.apache.org/jira/secure/attachment/12610186/caching-design-doc-2013-10-24.pdf
[17] https://archive.apache.org/dist/hadoop/core/hadoop-2.3.0/

[18] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, Ion Stoica, “PACMan: Coordinated Memory Caching for Parallel Jobs”
[19] Abad, C.L, Yi Lu, Campbell, R.H, Dept. of Comput. Sci., Univ. of Illinois at Urbana-Champaign, Urbana, IL, USA, “DARE: Adaptive Data Replication for Efficient Cluster Scheduling”, Cluster Computing (CLUSTER), 2011 IEEE International Conference on 26-30 Sept. 2011