HDFSX: Big data Distributed File System with small files support

Conference Paper · December 2016

DOI: 10.1109/ICENCO.2016.7856457

CITATIONS
7

READS
740

3 authors:

Passent Elkafrawy
Menoufia University
51 PUBLICATIONS **229** CITATIONS
SEE PROFILE

Amr Mausad
Menufiya, Egypt
11 PUBLICATIONS **29** CITATIONS
SEE PROFILE

Mohamed M Hafez
Menoufia University
2 PUBLICATIONS **8** CITATIONS
SEE PROFILE

HDFSX: Big Data Distributed File System with Small Files Support

Passent M ElKafrawy
Faculty of Science
Menofia University
Menofia, Egypt
basant.elkafrawi@science.menofia.edu.eg

Amr M Sauber
Faculty of Science
Menofia University
Menofia, Egypt
amr@science.menofia.edu.eg

Mohamed M Khalil
Faculty of Science
Menofia University
Menofia, Egypt
m.hafez@science.menofia.edu.eg

Abstract— **Hadoop Distributed File System (HDFS) is a file system designed to handle large files - which are in gigabytes or terabytes size - with streaming data access patterns, running clusters on commodity hardware. However, big data may exist in a huge number of small files such as: in biology, astronomy or some applications generating 30 million files with an average size of 190 Kbytes. Unfortunately, HDFS wouldn't be able to handle such kind of fractured big data because single Namenode is considered a bottleneck when handling large number of small files. In this paper we present a new structure for HDFS (HDFSX) to avoid higher memory usage, flooding network, requests overhead and centralized point of failure (single point of failure "SPOF") of the single Namenode.**

Big Data; Hadoop; HDFS; Small Files; HDFSX

I. INTRODUCTION

Nowadays, applications such as search engines, online auctions, webmail, online retail sale and others need to access data sets ranging from a few gigabytes to several terabytes or even petabytes. A distributed file system has been established to scale these data sets like Google's Distributed File System (GFS), Hadoop Distributed File System (HDFS) and Amazon Simple Storage Service (S3) [3]. HDFS components originally derived respectively from Google's MapReduce and Google File System (GFS) [4]. Hadoop is a portable, high reliability, high throughput, open source distributed file system. A Hadoop cluster scales computational power from small number of machines to large data centers and cloud systems. It consists of the HDFS as file system and a Programming framework, called MapReduce. They work as a Master Slave architecture; where a Metadata server, Namenode, serves a number of Data server, Datanode. The Namenode is responsible for managing all data files and organizes files access by storing their metadata in the server memory [8]. Files are split into one or more blocks and these blocks are stored in a set of Datanodes. Blocks replicated on Datanodes are managed by the Namenode for higher reliability and data availability. This replication helps rapid accessing of data, as distributed on the system and provides fault tolerance [2].

The Namenode is considered a bottleneck when handling large number of small files as metadata for each file is kept in memory to serve clients requests. Each file Metadata requires 125 bytes of space, however, consider applications as in biology generating 30 million files with an average size of 190 Kbytes; in astronomy exceed 20 million image files with an average less than 1 Mbyte; and in surveillance cameras, images of 1MB are created every second. Unfortunately, HDFS wouldn't be able to handle such kind of organized data, as the Namenode memory will exceed the addressing capacity of its memory, as it eats up the Namenode memory. This problem is called ***Higher Memory Usage***, which is addressed in the literature significantly [1].

Processing many small files increases the number of file access requests that are needed to run a job, such problem called ***Flooding Network Overhead***. To access the files, the metadata needs to be checked first to reach the data file. Too many metadata requests per second are issued which can flood the networking of the Hadoop cluster. This creates an overhead on the Namenode, hundreds of Small files will require hundreds of requests to the Namenode more than large files. Compare having a 1GB file broken into 64 MB blocks (i.e. 16 files) requiring 16 files requests; on the other hand, if there exists 1GB of 10,000 small files each with 100 KB size. The 10,000 files will require 10,000 files requests thus the job time might be hundreds of times slower than the equivalent with a single input file [2]. Moreover, the master/slave architecture is ***a Centralized point of failure (single point of failure "SPOF").*** The Namenode is the only node that holds the metadata of all files in the cluster. When this server fails then the whole cluster accessibility fails until the Namenode is up and running again. This centralization of processing is a bottleneck for the whole system known as a major problem. For the previous issues, we propose a modification for the HDFS architecture, in order to handle both large and small Big Data files, called HDFSX.

The rest of the paper is divided into the following, section 2 gives more details about HDFS architecture. Section 3 lists the solutions in the related literature. Section 4 presents the proposed system in more details. Finally, section 5 is a conclusion with future work.

II. HDFS ARCHITECTURE

HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern ***(Streaming data access)*** [8]. A dataset is typically generated or copied from source, then various analyses are performed on that dataset over time. Also, it doesn't require expensive, highly reliable hardware to run on. It's designed to run on clusters of ***Commodity hardware*** for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

HDFS cluster has two types of nodes operating in a master-worker pattern, a Namenode (the master) and a number of Datanodes (workers). The Namenode manages the filesystem namespace. It maintains the filesystem tree and stores the metadata for all the ***files*** (size, modified date, Authors…), ***directories*** (path of file: /home/etc/...) and ***blocks*** namespaces (Number of replicas, path of replicas, mapping from files to blocks and blocks to file). The Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the Namenode), and they report back to the Namenode periodically with lists of blocks that they are storing [4].

The HDFS include three kinds of file operations ***Write***, ***Read*** and ***Delete*** [5]. When client needs to Write or store data, it sends a write request to the Namenode then it will generate a Block ID. Consequently, the Namenode finds three Datanodes to save the data then the client will send the data to these Datanodes according to data flow; and notify the Namenode to save the metadata if successfully write. When reading data, client sends a read request to the Namenode then it will take several steps to tell the client where to read the data: First find the corresponding file in the directory tree. After that find the blocks belonging to the file according to the file block map. Then find the blocks needed in the whole file according to the block offset and length and find the Datanodes, which have stored these blocks. Sort the Datanodes of each block according to the distance between the client and then, send the sorted Datanodes of each block to the client [8]. When deleting files, client sends a delete request to Namenode then it finds out blocks belong to the file and delete the metadata only, the real data will be delete regularly.

III. RELATED WORK

Grant Mackey, Saba Sehrish and Jung Wang in 2009 [1] provided a solution to reduce the metadata footprint in Namenode's main memory, by archiving groups of files into a single file, named "***Harball archive***". In the structure of a Harball archive data is not compressed, files do not need to be expanding for accessing [2]. This system introduces another file layer on top of the HDFS, by creating an archive entry for a given file by storing its metadata into the Harball archive index. This index is considered as a meta-metadata layer for the files to increase the possible address space of the Namenode without increasing RAM. The archive was designed to have two index files (Master index, and Index) and a collection of part files. The Master index stores hashes, offsets and pointers to the files positions in the Index. The Index stores the file's status and the actual directory paths. The Part files store actual files data [2]. This method has some drawbacks: a slight overhead in referencing files, due to the fact that a request must now go through the metadata of the index to access the metadata of the required file [2]. Moreover, the Harball Archive is immutable after it has been created, as in order to add or remove files the archive must be re-created [8]. Finally, the flooding of the network overhead when serving file requests from the Namenode is not solved.

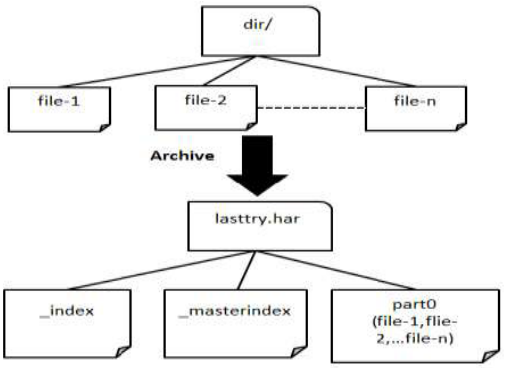

Fig. 1 Harball archive architecture

Yang Zhang and Dan Liu in 2012 [3] proposed another approach for small files on Namenode, which works as an independent Engine on the HDFS. This engine can reduce the overhead of HDFS effectively by merging small files. The model generates a file indexers and uses file block filling mechanism to accomplish files separation and files retrieval. This approach will not change the existing HDFS system but it improves the efficiency of HDFS by dealing with small files in a more efficient way. This Small file processing engine uses the Reactor multiplexed I/O, which can process massive request tasks. It contains two modules: Small Files Written and Merger to merge small files into a larger file then write it to HDFS, and Small Files Read and Separation to access small files when required [4]. This method has some drawbacks: that is the engine works outside the HDFS, just use HDFS for storing files in it. Also the merger size is not always equal to the file chunk size, fragmentation is developed in the HDFS block. Plus, the overhead of merging and separating files when written or required.

Liu Jiang, Bing Li and Meina Song in 2010 [5] provided another solution by allowing one block to save many small files and let the Datanode save some metadata in its memory. Typically, one block is mapped to have only one file, while a file could be stored in one or more blocks. They proposed the idea of storing multiple files in one block and replacing the file-to-blocks mapping by a new mapping from blocks-to-files. A file can only be mapped to one block, and one block may include many files.

Fig. 2 mapping from file to block

Each file has one bit to mark if it is valid, 1 is valid and 0 is invalid. Size is the block length. The researcher let the Datanode cache part of the metadata in their memory to overcome the bottleneck of the single Namenode. The metadata cached by Datanode include: the mapping from file to block and block to files and the locations of each block's replicas. Datanodes only caches block that has two copies in its own rack to reduce the amount of metadata each Datanode is caching. This method has some drawbacks, the new optimized HDFS does not effectively deal with large files because the new structure can only map a file to one block, hence, the file cannot exceed the block size. Also, the proposed method does not provide a solution to the Namenode memory that is highly occupied, it just distributes the requests overload to the Datanodes memory.

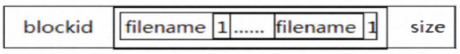

Fig. 3 mapping from block to files

IV. PROPOSED METHODOLOGY

We will propose a new structure for Hadoop Distributed file system (HDFS) to handle large number of small files addressing all the raised fall-offs, namely; Large amount of metadata, Flooding Network Overhead and Centralized point of failure. The new HDFS system will have five modifications in its infrastructure. First, the Namenode will store metadata until a certain threshold to avoid ***Higher Memory Usage***. Second, the extra metadata is stored in another secondary media, as Namenode main memory extension (like flash drive). Third, a metadata Indexer is added to the Namenode to locate the metadata when exceeded on the flash drive. Forth, each Datanode holds a copy of its own files metadata, to handle access requests of the same data without returning to the Namenode to reduce ***Network Flooding***. Finally, each metadata partition is linked with pointers to the head and tail of the next Datanode metadata, so as when the Namenode is down requests can be handled instantly to overcome the ***Centralization point of failure***.

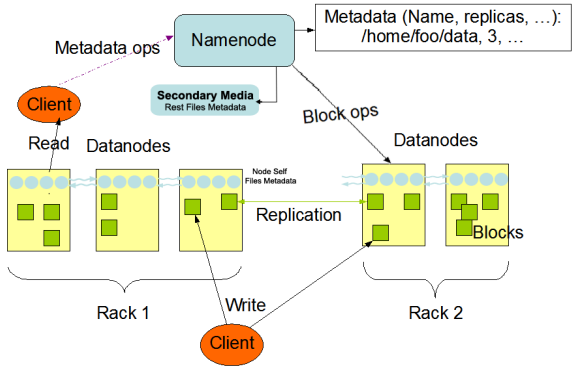

Fig. 4 HDFSX: New structure for current HDFS

A deeper discussion of the proposed system covering the three fall-offs is as follows:

A. *Large amount of metadata:*

This scenario's main problem is the vast amount of metadata that may equal or in some cases surpass the storage amount itself, which may occupy the whole memory capacity resulting numerous performance issues. We recommend a threshold to be used to determine how much metadata is stored in the main memory and if reached, the rest metadata is stored on a secondary media (flash drive is suggested) to simulate a virtual memory. This flash storage layer will be accessible from Namenode to read and write metadata. Furthermore, indexers will be added to the Namenode memory for faster accessing and enhanced performance.

B. *Flooding the networking overhead in the single Namenode:*

Accessing small files in HDFS is considered a bottleneck, the request of accessing files from Datanodes need to go through the Namenode for reading the metadata in order to find the blocks of files. The number of requests increases more with large numbers of small files. We will let the Datanodes save (cache) metadata of their files in its memory, then when MapReduce jobs in Datanodes need to access a file in the same Datanode; can smoothly find the metadata of the file in the local memory that make accessing the file faster and reduce the overhead of single Namenode.

C. *Avoid the Centralized failure of Namenode:*

The single Namenode architecture, while simple and handy, takes an obvious risk as the life cycle of the whole storage system is bound to the status of the Namenode. If it is down or not available even for a very short window, the system is down as the Datanodes need it for each access operation. We propose pointer-based solution; pointers are to be added in the beginning and the end of each Metadata partition stored in the RAM of the Datanode (as proposed in (II)). This way we link the Metadata sub-partitions all together to handle Namenode failure. Consequently, if the Namenode is down, requests can access the other metadata without the need for the Namenode.

V. PROPOSED DESIGN

The proposed design has a new structure for both Namenode and Datanodes:

A. *Namenode:*

However, the structure of files metadata in the Namenode memory stay as is, when store new metadata; the size of the metadata must be checked if exceed a predefined threshold which sited previously by the system administrator. To overcome the Namenode memory limitation, secondary media of high speed flash drive is used in addition to the Namenode. When the Namenode memory used size exceeds a certain threshold then any metadata will be redirected to flash drive. This is to be accomplished by replacing metadata in Namenode memory with an Indexer of the metadata as

stored in flash drive. The metadata itself is stored in flash drive and an index file will use to directly find such metadata is placed in Namenode memory, thus no search for files metadata at file requests is necessary. Flash drives high speed is specifically used as a virtual memory mechanism for Namenode without the page swaps mechanism as small metadata of file metadata is stored at the Namenode - i.e. the index to the file metadata. When a client request files metadata, the Namenode will search in its memory and deliver them. If Namenode could not find this metadata in its memory, it will search the indexer and get direct reference to the metadata in the secondary media then deliver it to the client.

Using a threshold will simulate the original HDFS as long as the metadata is in manageable size and the proposal methodology will work with no overhead since auxiliary storage media is used.

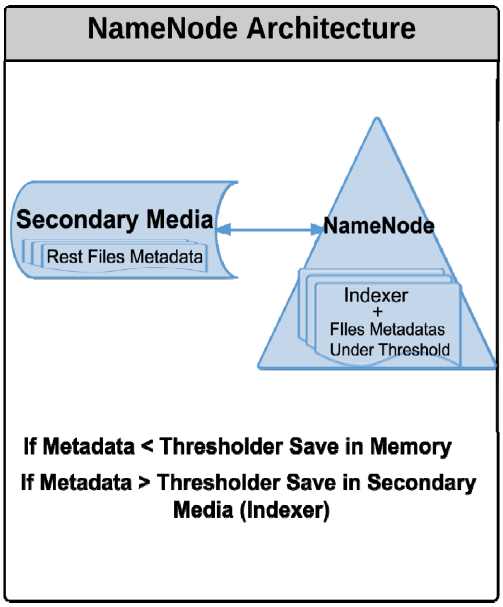

Fig. 5 A new Namenode structure

B. *Datanodes:*

We added a self-describing nature to every Datanode by distributing replica of Metadata among them and using pointer system to address the next and previous Datanodes. The client after the first read request will not need to return to the Namenode, since the Datanodes will store its files metadata. Self-node files metadata are in its memory and will handle the read requests normally. To help Datanode handle more client requests, pointers will be added in the beginning and the ending of the cached metadata to refer to metadata parts in other Datanodes. Consequently, other Datanode will deliver the required metadata from its cache or redirect the request to the Namenode if not found in the required metadata.

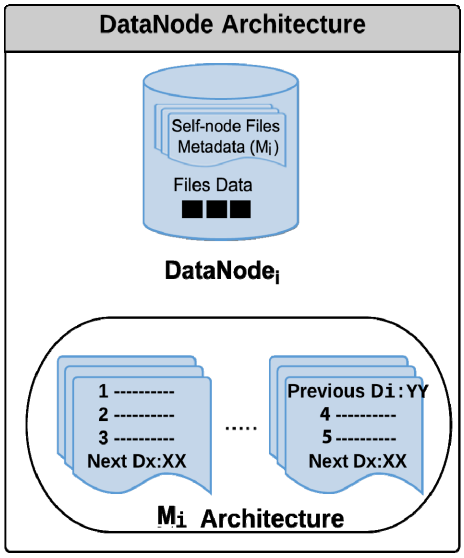

Fig. 6 A new Datanode structure

VI. PROPOSED TECHNIQUE

The new HDFS structure proposes a new technique for the three main operation ***Write***, ***Read*** and ***Delete*** for both clients and MapReduce jobs. Our modifications in HDFS infrastructure will handle requests efficiently and avoid previous fall-off.

When a ***Read*** file is required, just first request from client will go to Namenode for handling it normally. Namenode will start search in memory to find file metadata from directory tree then find the blocks that belong to this file. Namenode will search into indexer, if file metadata not found in memory then get the file metadata from secondary media and get the blocks details. Finally, blocks are sorted according to distance to Datanode, nearest block is sent to the client. Usually the client will read the file from first Datanode. In case the client needs to read another file, he will not send a request again to the Namenode but will send it to the current Datanode. Related files can be found in the same Datanode, because the files are previously organizing in a hierarchical file structure before being written to HDFS. The Datanode will handle read request normally like the Namenode by searching in its memory to find the file. Grant file access to the client and send reports to the Namenode after the client session. This will add self-describing nature to every Datanode such that communication will not require every time with the Namenode to acquire request access, since needed metadata can be found in the current Datanode. If Datanode couldn't find the file metadata in memory, the client request will be redirect to the next or previous Datanode according to the order of the requesting file to the previous file requested. Respectively if metadata not found in the previous or next Datanode, the Namenode is invoked to redirect the request to right Datanode and contain executions. If the Namenode is down (failure scenario), The request will be redirect automatically to other next or

previous Datanode according to direction ordered. Furthermore, all Datanodes are connected in circular queue manner.

When ***Write*** files are required, first, the client might organize the files in a hierarchical file structure (directory). This will help in storing the related files in the same Datanode and thus reducing file access overhead. Second, the client shall send a write request to the Namenode for locating the free blocks in the Datanodes; the Namenode will handle it like a normal process. Third, the client sends the files to free blocks in Datanodes. Finally, after the data is stored successfully, the metadata will be stored in memory of both the Namenode and Datanode. When the size of metadata exceeds the limit of Namenode memory (according to a certain threshold), coming metadata shall be stored in secondary media and defined into the indexer to ***avoid Higher Memory Usage***. To maintain the self-describing nature of every Datanode, we shall add pointers to the next Datanode address.

When ***Delete*** files are required, the client sends a delete request to the Namenode. The Namenode shall find the blocks that belongs to the file and delete the metadata only; the real data will be deleted regularly. Then the Namenode should inform the Datanodes to remove the file's metadata form its memory.

VII. Conclusions and Future Work

Small files management is one of the most obvious weaknesses of HDFS; however, small files are required for many applications such as biology, astronomy and scientific systems … etc. HDFS must be enhanced to satisfy these applications. In this paper, we proposed an enhanced HDFS (HDFSX) architecture, which is compatible with both small and large files. HDFSX solve the three issues that make the current HDFS fail with small files; 1-***Namenode Higher Memory Usage***: threshold is set and secondary Media is used, 2-***Namenode Overhead***: a metadata replica is distributed among the Datanodes and pointers are utilized to reduce the need of client to return to Namenode unless required, 3-***Centralized point of failure***: the client operation can work isolated from Namenode as long as the current Datanode contains the required metadata.

As for future work, we will implement the proposed technique in real environment with two possible big data sets, huge number of small files and large data files. To study the effect of the amendments for the two cases and analyze experimental result for increasing levels of technique effectiveness. Hence, we can point out that in the future we need to implement the proposed technique in real environment by applying changes in the Hadoop core code and libraries. Analyze experimental result for the new technique in huge number of small files. Study the effect of the technique amendments for large Big Data files. Study technique performance with Namenode failure and side effects, once and Datanode failure and its effects on the whole system. In other words, the performance analysis of the systems need to be developed in the next research publication.

VIII. References

[1] Grant Mackey, Saba Sehrish, Jung Wang: Improving Metadata Management for Small Files in HDFS, 978-1-4244-5012-1/09 ©2009 IEEE.

[2] Vaibhav Gopal Korat, Kumar Swamy Pamu: Reduction of Data at Namenode in HDFS using harballing Technique, International Journal of Advanced Research in Computer Engineering June 2012, 2278 – 1323 ©2012 IJARCET.

[3] Yang Zhang, Dan Liu: Improving the Efficiency of Storing for Small Files in HDFS, International Conference on Computer Science and Service 2012, 978-0-7695-4719-0/12 ©2012 IEEE.

[4] Bo Dong, Jie Qiu, Qinghua Zheng, Xiao Zhong, Jingwei Li, Ying Li: A Novel Approach to Improving the Efficiency of Storing and Accessing Small Files on Hadoop: a Case Study by PowerPoint Files, 978-0-7695-4126-6/10 ©2010 IEEE.

[5] Liu Jiang, Bing Li, Meina Song: The Optimization of HDFS Based on Small Files , 978-1-4244-6769-3/10 ©2010 IEEE.

[6] 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, 978-1-4673-2907-1/13 ©2013 IEEE.

[7] Ankita Patel, Mayuri A. Mehta: A Novel Approach for Efficient Handling of Small, International Advance Computing Conference (IACC), 978-1-4799-8047-5/15 ©2015 IEEE.

[8] Tom White. Hadoop Book: The Definitive Guide. O'Reilly Media, Inc Third Edition 2012.

[9] Small File Problems in HDFS Available: http://www.cloudera.com/blog/2009/02/the-small-files-problem/

[10] Hadoop har archive files idea. Available: http://developer.yahoo.com/blogs/hadoop/posts/2010/07/hadoop_archive_file_compaction/

[11] [10] Apache Hadoop 2.4.1, HDFS Federation: http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/Federation.html

[12] Data Science and Big Data Analytics Guide, 2012 EMC Corporation.