Hadoop NameNode: Single Point of Failure

Snehal Vathiyath
Student
Department of Computer Application
SIES College of Management Studies Nerul, Navi Mumbai. Maharashtra

Abstract— Nowadays, Companies generates large amount of unstructured data in the form of logs, comments, chats etc. So there is a need to process Multi Zattabyte datasets efficiently. Here the best solution would be to choose Hadoop, Open Source Apache Software Foundation. Basically, it's a way of storing enormous datasets across distributed clusters of servers and then running "distributed" analysis applications in each clusters which attracts it the most. Hadoop is a popular open-source implementation of MapReduce for the analysis of large datasets. The problem defined in this paper is that of Single Point of Failure of NameNode which is master of the HDFS cluster. The actual challenge is I/O speed for analysing the data and not the storage capacity. This paper develops a novel for overcoming the Single Point of Failure in NameNode. The optimal solution to this problem is the minimal overhead on NameNode. The novality is in achieving a hierarchical storage of the HDFS system. The conceptual result illustrates the faster and accurate service of proposed framework.

Key words: HDFS, NameNode, Secondary NameNode, ZattaByte, Overcome Single Point of Failure, BigData

I. INTRODUCTION

BigData is the data which is beyond to the storage capacity and processing power. This Bigdata is generated from CCTV Cameras, Social Media, Ecommerce sites, Airlines, Hospitality data, Stock market etc. The data generated from these sources are very large and are in unstructured format. Such Bigdata are only used for analysis and decision making for future by data analysts. To process such BigData in less time Hadoop has been introduced. Hadoop is a "flexible and available architecture for large scale computation and data processing on a network of commodity hardware". As Hadoop is an open source framework for processing, storing and analyzing massive amounts of distributed unstructured data. Originally created by Doug Cutting at Yahoo!. Hadoop was inspired by MapReduce, a user- defined function developed by Google in early 2000s for indexing the Web. It was designed to handle petabytes and Exabyte's of data distributed over multiple nodes in parallel. Hadoop is now a project of the Apache Software Foundation, where hundreds of contributors continuously improve the core technology. The basic concept of Hadoop is that rather than dumping away huge block of data in a single machine, Hadoop breaks up Big Data into multiple parts so each part can be processed and analyzed at the same time.

II. BACKGROUND

Basically, Hadoop can be simply defined as combination of HDFS and MapReduce. Hadoop uses HDFS to manage storage resources across the cluster and MapReduce for processing that large datasets. HDFS is written in Java and designed for portability across heterogeneous hardware and software platforms. MapReduce parallel programming model. Hadoop is composed of a MapReduce engine and a user-level filesystem, HDFS that manages storage resources across the cluster. For portability across a variety of platforms — Linux, FreeBSD, Mac OS/X, Solaris, and Windows — both components are written in Java and only require commodity hardware. Both are platform independent since they are written in Java.

A. HDFS

The Hadoop Distributed File System (HDFS) is a file system designed for storing very large files with streaming data access patterns, returning clusters on commodity hardware. It is designed as a massive data storage framework and serves as the storage component for the Apache Hadoop platform. Data is distributed over several machines, and replicated to ensure their durability to failure and high availability to parallel application. HDFS is designed for very large file (in GBs, TBs). The Hadoop Distributed File System (HDFS) provides global access to files in the cluster. For maximum portability, HDFS is implemented as a user-level filesystem in Java which exploits the native filesystem on each node, such as ext3 or NTFS, to store blocks i.e 64MB and each block is stored as a separate file in the local file system.

In addition to a centralized NameNode, all remaining cluster nodes provide the DataNode service. Each DataNode stores HDFS blocks on behalf of local or remote clients. Each block is saved as a separate file in the node's local filesystem. Blocks are created or destroyed on DataNodes at the request of the NameNode, which validates and processes requests from clients. Although the NameNode manages the namespace, clients communicate directly with DataNodes in order to read or write data at the HDFS block level. The file system is based on hardware and provides a highly reliable storage and high throughput of global access to large data sets. Reason is that before the start of an I/O transmission, there are some necessary initialization steps need to be done, etc. data location retrieving and storage space allocation.

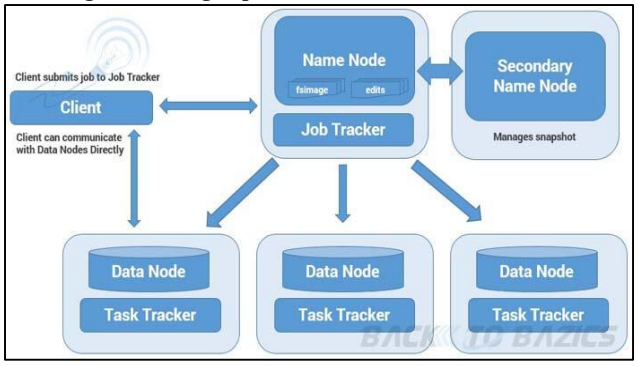

Fig. 1: HDFS Architecture

When transferring large data, this initialization overhead becomes relatively small and can be negligible

compared with the data transmission itself. However, when transferring small size data, this overhead becomes significant. In addition to the initialization overhead, files with high I/O data access frequencies can also quickly overburden The regulating component in the HDFS, i.e., the single name-node that supervises and manages every access to data-nodes. If the number of data-nodes is large, the single name-node can quickly become a bottleneck when the frequency of I/O To read an HDFS file, client applications simply use a standard Java file input stream, as if the file was in the native filesystem. Behind the scenes, however, this stream is manipulated to retrieve data from HDFS instead. First, the NameNode is contacted to request access permission. If granted, the NameNode will translate the HDFS filename into a list of the HDFS block IDs comprising that file and a list of DataNodes that store each block, and return the lists to the client. Next, the client opens a connection to the "closest" DataNode and requests a specific block ID. That HDFS block is returned over the same connection, and the data delivered to the application.

To write data to HDFS, client applications see the HDFS file as a standard output stream. Internally, however, stream data is first fragmented into HDFS-sized blocks (64MB) and then smaller packets (64kB) by the client thread. Each packet is enqueued into a FIFO that can hold up to 5MB of data, thus decoupling the application thread from storage system latency during normal operation. A second thread is responsible for dequeuing packets from the FIFO, coordinating with the NameNode to assign HDFS block IDs and destinations, and transmitting blocks to the DataNodes for storage. A third thread manages acknowledgements from the DataNodes that data has been committed to disk.

B. Map-Reduce

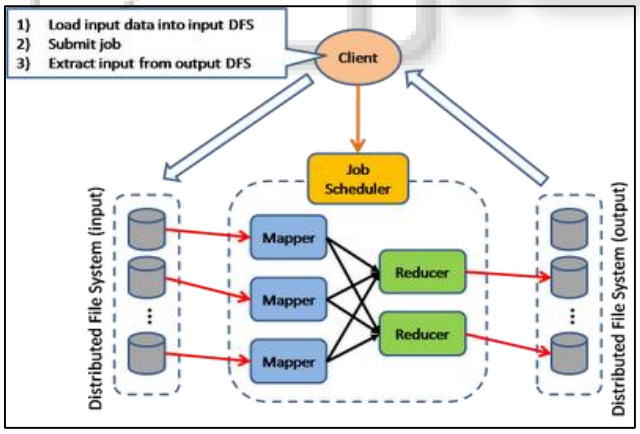

Fig. 2: MAP-REDUCE

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programme also specifies two functions: the map function and the reduce function. Working here application is divided into many small pieces, each of which can be executed or to perform any node in the cluster. Hadoop is an important feature of partition of data and calculation in many thousands of the host, and execute the application parallel computing examples close to their data through the use of graphs. In the nutshell in a typical MapReduce job, multiple map tasks on slave nodes are executed in parallel, generating results buffered on local machines. Once some or all of the map tasks have finished, the shuffle process begins, which aggregates the map task outputs by sorting and combining key-value pairs based on keys. Then, the shuffled data partitions are copied to reducer machine.In Hadoop, a centralized JobTracker service is responsible for splitting the input data into pieces for processing by independent map and reduce tasks, scheduling each task on a cluster node for execution, and recovering from failures by re-running tasks. On each node, a TaskTracker service runs MapReduce tasks and periodically contacts the JobTracker to report task completions and request new tasks. By default, when a newtask is received, a new JVM instance will be spawned to execute it. Hadoop divides the input to a MapReduce job into fixed-size pieces called inputsplits, or just splits. Hadoop creates one map task for each split, which runs the userdefined map function for each recording the split. Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the processing is better load-balanced when the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained. On the other hand, if splits are too small, the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of an HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files) or specified when each file is created. Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization because it doesn't use valuable cluster bandwidth.

Sometimes, however, all three nodes hosting the HDFS block replicas for a map task's input split are running other map tasks, so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer. It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient than running the whole map task using local data.

Map tasks write their output to the local disk, not to HDFS. This is because the Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away. So storing it in HDFS with replication would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output. Reduce tasks don't have the advantage of data locality; the input to a single reduce task is normally the output from all mappers. In the present

example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in HDFS for reliability. Each HDFS block of the reduce output; the first replica is stored on the local node, with other replicas being stored on off-rack nodes.

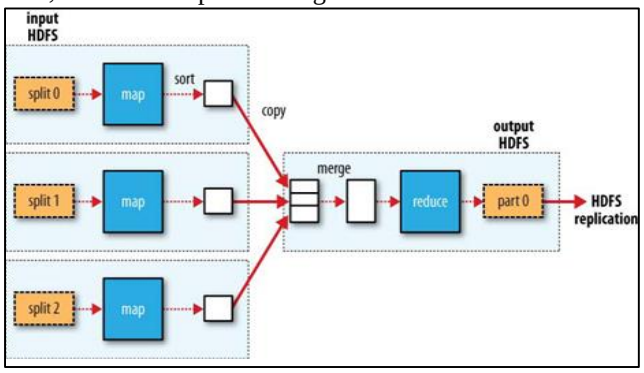

Fig. 3: Working of reduce task

C. Drawbacks of the Current Systems

A NameNode can eat up memory, since a reference to every block of every file is maintained in memory. It's difficult to give a precise formula because memory usage depends on the number of blocks per file, the filename length, and the number of directories in the filesystem and it can even change from one Hadoop release to another.

The NameNode server in the Hadoop cluster keeps the track of filesystem meta-data, it keeps a track of how your files are broken down into file blocks, which nodes store those blocks, and the overall health of the distributed filesystem. From the NameNode is a single container file system metadata, it naturally became the file system constraints. In order to make the metadata operations is fast, the NameNode the namespace loaded into memory, so the size of the namespace is a limited number of available RAM the NameNode. Therefore, the available memory NameNode has no chance to limit the number of customer cluster growth. Is also a big HDFS install one operation in a large memory space of the NameNode JVM is susceptible to almost all frequent garbage collection, this may require the NameNode service for a few minutes. Thus it is clear that memory available to the NameNode machine in a Hadoop cluster dictates the size of cluster and ultimately the number of active clients using a Hadoop based application.

III. Related Work

Extended NameNode Structure is shown in Figure 5. The original HDFS structure, which consists of a single NameNode, is redesigned with a two-layer NameNode structure. The first layer contains a Master Name Node (MNN) which is actually the NameNode in the original HDFS structure. The second layer consists of a set of Secondary Name Nodes (SNNs), which are applied to every rack of datanodes.

Similarly, to write data into a datenote, the MNN first allocates one or more racks for the client application to write (Step A). Then, the SNN further allocates datanodes for the client.

A. The Model Architecture

In this architecture, along with all the basic block of Hadoop a hardware connection. This increases the hardware is quite rapid access storage device. It as a least recently used secondary storage devices to store metadata. Metadata is stored in it will refer to only when the request for access to the data is used less often.

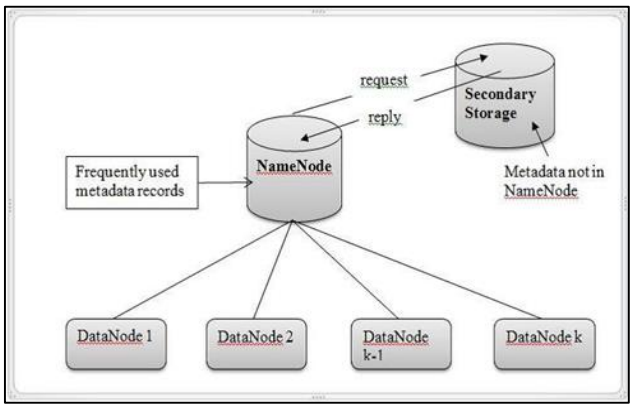

Fig. 5: Extended NameNode Structure

When the client performs a read operation the usual processing that is done by the NameNode to fetch metadata for requested file will be done and the client will be given back the file. The only difference will occur when the requested file is moved to the secondary storage as it was not recently used and the NameNode has reached threshold value. In this case the NameNode will not find the record in its memory and thus a request will be made to secondary device to fetch the metadata record and the metadata record will be removed from secondary device and will be loaded again on the RAM.

IV. Conclusion

In this paper, I have proposed a novel methodology of hierarchical storage system based on Hadoop framework. This way the Hadoop cluster will not reach the stage where the NameNode becomes irresponsive due to excessive JVM garbage collection as the HDFS will not be heavily loaded. Also as the NameNode will only store relatively more frequently used data the operations carried on the cluster will be faster and more efficient.

References

[1] Tom White. "HADOOP the Definitive Guide".
[2] Garry Turkington. "Hadoop for Beginners".
[3] Eric Sammer. "Hadoop Operations".
[4] Novel methodology for optimal reconfiguration of distribution networks with distributed energy resources.
[5] http://www.scientific.net/AMM.519-520.1325
[6] http://bigdata.devcodenote.com/
[7] Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium.
[8] HDFS (hadoop distributed file system) architecture. http ://hadoop.apache.org/common/docs/current/hdfs design.html, 2009.
[9] Hadoop And Distributed Computing at Yahoo!(http://developer.yahoo.com/hadoop)

