Analysis of the Hadoop Architecture

Chongshu Qian
chongshq@uci.edu
65486575

Muhammad Talha Khan
muhammtk@uci.edu
46281174

Wei Pan
panw4@uci.edu
72723828

***Abstract*—The advent of big data has forced us to switch to distributed computing systems to solve problems. Apache Hadoop is an open source software for reliable, scalable and distributed computing. It allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to be scalable and also fault tolerant.**

INTRODUCTION

One of the main challenges in Distributed Systems is the management of resources. Apache Hadoop [1] is a collection of open-source software utilities that facilitate the using of a network of many computers (A Distributed System) to solve problems involving massive amounts of data and computation. It provides a software framework for distributed storage and the processing of big data. In this survey project our main focus will be to compare the the two different versions of Hadoop: Hadoop version 1 (V1) and Hadoop version2 (V2). We will discuss in detail the architecture of both of these versions and how V2 was able to improve on the shortcomings of V1. We will also talk about the existing applications of Hadoop and the open problems for the future.

PROBLEM DESCRIPTION & PROPOSED SOLUTION

With the emergence of massive amounts of big data, performing data analytics now requires petabytes of storage and an abundance of processing power. It is not easy or convenient to store massive amounts of data in traditional database management systems. The introduction of Hadoop infrastructure mainly focuses on these three scenarios that could meet the need of big data processing:

1) **Storing enormous dataset:** If the dataset is as big as terabytes or petabytes and cannot be held using techniques such as NoSQL (Not Only SQL) and RDBMS (Relational database management system), Hadoop is able to manage that. It is also feasible for the case that the dataset is expanding due to various factors.
2) **Storing a diverse set of data:** Hadoop is capable of storing and processing all kinds of file data, large or small, plain text or images, even multiple different version of some particular data format.
3) **Parallel data processing:** The design of MapReduce algorithm makes it possible to process the dataset concurrently on different machines, which in turn reeduces the time complexity drastically.

Hadoop mainly allows us to handle very big amounts of data (structured or unstructured) in an effective manner. The advantages of using Hadoop over traditional Database management software are listed below:

1) **Flexiblity:** Ability to store and process huge amounts data without pre-processing it first.
2) **Compute Power:** Hadoop's distributed computing model processes data fast by dividing the tasks and executing them in parallel.
3) **Fault Tolerance:** If a node goes down, jobs are automatically redirected to other nodes in the system. Also, data stored on the nodes is replicated in case a node fails, data is not lost.
4) **Scalability:** Cluster of nodes can be grown easily to handle more data by simply adding more nodes.
5) **Low cost:** The open-source framework is free and uses commodity hardware to store large quantities of data.

MAIN COMPONENTS

Hadoop can be divided into three major state of the art components listed below:

1) HDFS (Hadoop Distributed File System)
2) MapReduce Programming Model
3) YARN (Yet Another Resource Negotiator)

YARN was not included in Hadoop V1 rather it was introduced in Hadoop V2. We talk in the coming sections in detail about the architecture of both of these state of the art software utilities. We also discuss their applications and explain what shortcomings Hadoop V2 was able to address that were present in V1.

HDFS

Hadoop provides a distributed file system (HDFS) [2] and a framework for the analysis and transformation of very large data sets using the MapReduce [3] paradigm. For this section we will only focus in HDFS. An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and executing application computations in parallel close to their data. HDFS stores file system metadata and application data separately. HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP based protocols. Also unlike other file sytems which use data protection mechanisims such as RAID (Redundant Array of Independent Disks) [4] for data durability, HDFS replicates data on multiple DataNodes. This creates more opportunities for locating computation near the needed data. Figure 1 below gives a high level view of the HDFS file sytem.

NameNode: Files and directories are represented on

the NameNode by inodes, which record file attributes like permissions, modifications, space quotas and etc. Each file's content is broken down into chunks of size 64-128MB, can be user defined too, and is stored on multiple DataNodes, usually three but again this can be user defined too. The NameNode maintains the mapping of the file blocks in the DataNodes. When an HDFS client needs to read/write files from/to the DataNode it requests the NameNode to do so. The current design has a single NameNode per cluster. Where each cluster can have a multitude of DataNodes and clients. In addition to this, the NameNode is a multi-threaded system and is able to process requests simultaneously from multiple clients.

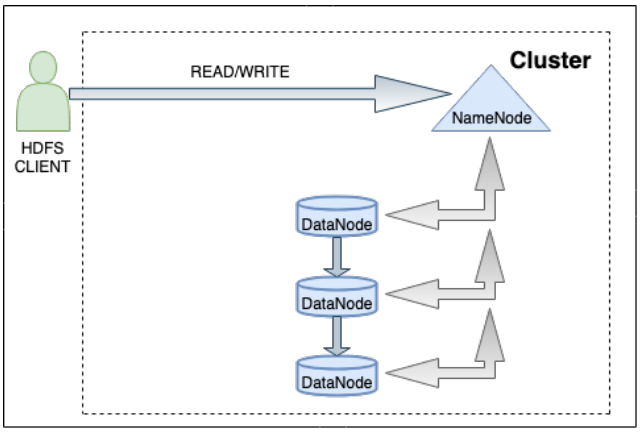

Fig. 1. High Level View of HDFS

DataNode: A block on the DataNode is represented by two files. The first file contains the data itself and the second file contains the block metadata, e.g. the checksum of the block. Also, if the size of the data file does not equal to the actual length of the block it does not have to be rounded up to the nominal block size as it happens in traditional file systems. During startup each DataNode handshakes with the NameNode and is assigned an ID. If the ID of a DataNode does not match with the ID of a NameSpace Node the node is not allowed to join the cluster, preserving the integrity of the file system. The DataNode during its operation consistently sends heartbeats, every three seconds, to the NameNode. A heartbeat informs the NameNode that the DataNode is in service and also carries some statistics like free space, storage capacity and number of transfers in progress. The statistics help the NameNode in space allocation and load balancing. If the NameNode does not receive a heartbeat in ten minutes it considers the DataNode to be out of service. The NameNode does not directly communicate with the DataNodes rather it replies to the heartbeat with a set of instructions which include commands like:

- Replicate blocks to other nodes
- Shutdown/Re-register the node
- Send a block report

The commands are vital for system integrity. This is why the frequency of heartbeats is kept high enough. A NameNode is able to process thousands of heartbeats per second without affecting any other NameNodes in the system.

HDFS Client: User applications access the file system using the HDFS client, a code library that exports the HDFS file system interface. The client is able to perform operations such as read, write and delete, without having the knowledge that how the files are stored in the cluster. This is the layer of abstraction that HDFS provides for its clients. When an application reads a file the HDFS client first asks the NameNode for the list of DataNodes that host the replicas of the blocks of the file. It then contacts the DataNode directly and requires the transfer of the desired block. If an application needs to write, the HDFS client first contacts the NameNode which assigns it multiple DataNodes, based on their capacity and load, where the client can write. The client then organizes a pipeline from one DataNode to another and sends the data that needs to be written. HDFS also provides an API that exposes the location of file blocks. This allows applications like the MapReduce framework to schedule a task near to where the data is located, thus improving the read performance. It also allows an application to set the replication factor of a file (three by default). For critical files or files which are accessed very often, having a higher replication factor improves their tolerance against faults.

MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real world tasks. Users specify the computation in terms of a *map* and a *reduce* function and the underlying system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. More than ten thousand MapReduce programs have been implemented internally at Google and hundred thousand MapReduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day.

Programming Model: The user of the MapReduce library expresses the computation as two functions: map and reduce. Map, written by the user takes in as input a pair and produces a set of intermediate key/value pairs. All the intermediate values associated with the same key are passed to a specific reduce function. The Reduce function, also written by the user accepts and intermediate key and a set of values, merges these values and outputs a result. The set of values are provided via an iterator which allows us to handle lists that are too large to fit in the memory. The example below help clarifies how this model works. Consider the problem of counting the occurrence of each word in a large collection of documents. In this case the *map* function would emit as intermediate pairs each word and a count associated with it (initially 1) e.g. (*Hello*, 1). The reducer function sums together all counts emitted for a particular word e.g. all the intermediate pairs which had key *Hello* would go to a particular reducer that would sum the values associated with *Hello* and output a result, which would be the count of *"Hello"* in this case.

Execution Overview The steps below explain how MapReduce works when a user application calls the MapReduce function.

1) The MapReduce Library first splits the input files into M pieces. (The size of the split can be defined by the user).
2) One of the nodes is designated as the master and all other are worker/slave nodes. The master picks idle workers and assigns each one a map or a reduce task.
3) A worker who is assigned a map task reads the corresponding data input split, passes the data to the user defined map function and writes to a buffer memory the intermediate key/value pairs (The output of the map function).
4) The location of the key/value pairs is sent back to the master node which partitions them into R groups and forwards these pairs to idle reducer workers.
5) The reducer worker sorts the intermediate keys after reading them remotely from the buffer and then sends them to the user defined reduce function. The output of the reducer function is appended to a final output file for this reduce partition.
6) When all the map and reduce tasks are completed the master node wakes up the user program.

Figure 2 below shows how the execution of the MapReduce function progresses.

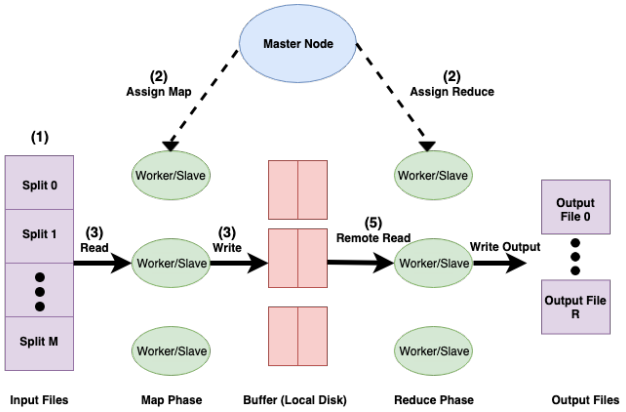

Fig. 2. The Execution Overview of MapReduce

Fault Tolerance: Since the MapReduce library is designed to process very large amounts of data using hundreds and thousands of machines it does tolerate machine failures. The master node pings the worker nodes periodically and if no response is received the master marks the worker as failed. In case of a map failure the map task is assigned to a new worker node and re-executed. The reducer nodes are also notified of the failure so they can start reading from the local buffer and continue the execution of the program.

YARN

Two problems arise from the broad adoption and ubiquitous usage of MapReduce and HDFS: firstly the programming model (MapReduce) and the resource management infrastructure (HDFS) are tightly coupled, forcing developers to abuse the usage of MapReduce; secondly, the centralized handling of the flow control of jobs, i.e. the master node controls all the jobs running on a current cluster, which results in endless scalability concerns for the scheduler.

Yet Another Resource Negotiator (YARN) [5] is designed to address these problems, while at the same time improves performance in multi-tenancy, cluster utilization, scalability and compatibility. By separating resource management functions from the programming model, YARN delegates scheduling functions to components, specialized for different jobs. In this new context, MapReduce is just one of the applications running on top of YARN. The four major components which make up the YARN architecture are liste below:

1) **Resource Manager (RM):** The RM runs on a master daemon as a background process. It acts as the central authority, arbitrating resources among various competing applications in the cluster.
2) **Application Master (AM):** There is one AM for each application. It is the "head" of a job, managing all lifecycle aspects including dynamically adjusting resources consumption, managing the flow of execution, handling faults and computational skews and performing other local optimizations.
3) **Node Manager (NM):** NM runs on slave daemons, monitoring containers resource usage and reporting faults to the RM. It also manages the user process on that machine and tracks the health of the node on which it is running.
4) **Container:** It is a package of resources including computation, memory and network bandwidth on a single node.

Figure 3 below shows the architecture of YARN.

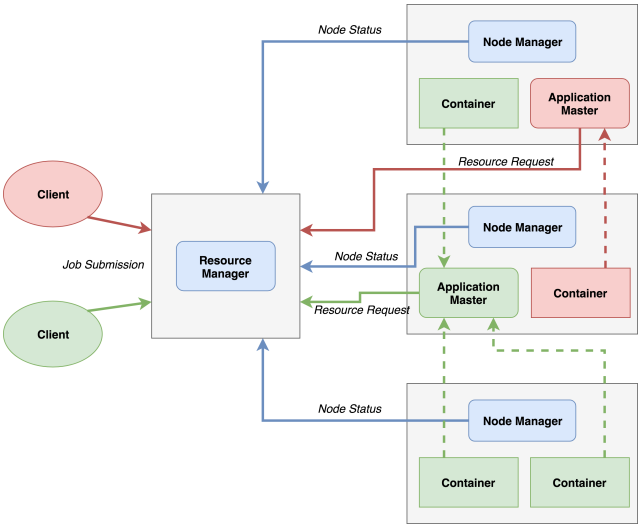

Fig. 3. The Architecture of YARN

Client jobs are submitted to the RM via a public submission protocol. They are checked for their security credentials in the admission control phase before being passed to the scheduler to be run. To obtain the computation and memory resources required for running a job, the AM for that job sends resource requests to the RM, informing the latxter of the specification of locality preferences and properties of the containers. The RM dynamically allocates containers to applications based on the application demand, scheduling priorities and resource

availability. It also generates a lease for those containers. When the resources are enough for the job and available for use, the AM encodes a launch request with the lease. Then, the job is switched to running state. In addition, a record of accepted jobs are recorded to persistent storage and recovered in case of the RM failure.

The communications among the four major components are versatile: The RM interacts with NM using heartbeat-based communications in order to track the node status. An AM may need to harness the containers available on multiple nodes to complete a job. Containers may communicate directly with the AM to report status and receive commands.

In conclusion, by delegating all the job-specific functions to AMs, the YARN architecture is a robust infrastructure with fault tolerance, scalability, programming model flexibility and improved upgrading/testing.

Analysis of YARN & MapReduce

The table below summarizes the differences between YARN and MapReduce.

	YARN	MapReduce
Version	Introduced in Hadoop 2.0	Introduced in Hadoop 1.0
Size	Default Data Node size is 128MB	Default Data Node size is 64MB.
Responsibility	Responsible for Resource Management only	Responsible for Resource Management as well as data processing
Execution Model	Execution Model is more generic	Less generic execution model
Application	Able to execute applications which don't follow the MapReduce programming model	Can only execute applications which confine to the MapReduce programming model
Architecture	Works on top the HDFS. Has new components such as the Resource Manager, Node Manager and Application Master.	Works on top of HDFS directly and has components like Job Tracker and Task Tracker.
Fault Tolerance	No concept of a single point of failure in YARN because it has multiple masters, if one fails another master resumes the execution	Single point of failure, only one Master Node per cluster
Scalability	Cluster can consist of ten thousand plus nodes	Cluster size is limited to 4000 nodes.

In the coming sections we discuss how different verison of Hadoop incorporate components like HDFS, MapReduce and YARN.

Hadoop V1 Architecture

Hadoop V1 can be divided into two major components: the **HDFS** and the **MapReduce** Programming Model. Both of these components work together, with MapReduce working on top of HDFS. Figure 4 shows the architecture of Hadoop V1.

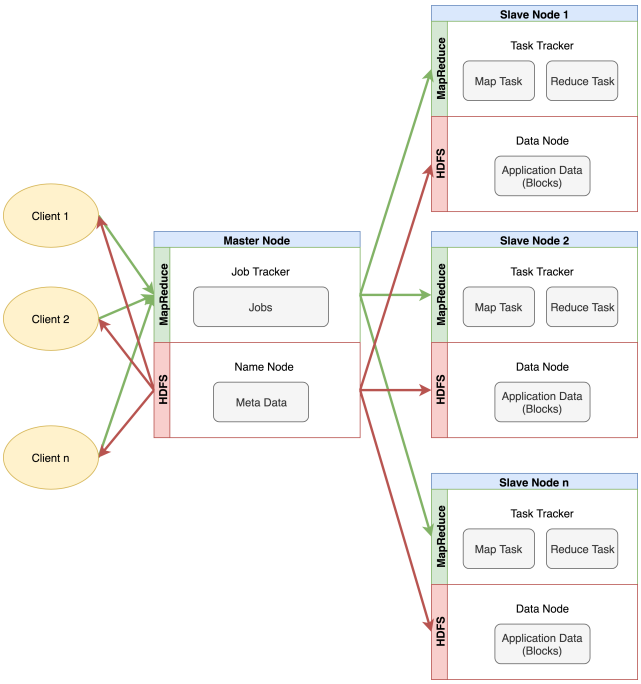

Fig. 4. The Architecture of Hadoop V1

The execution of a user program is carried out in the steps listed below:

1) Clients submit their job to Hadoop System.
2) Client requests are first received by a Master Node.
3) Master Node's MapReduce component Job Tracker is responsible for receiving client jobs. It divides them into manageable independent tasks and assigns them to Task Trackers.
4) Slave Node's MapReduce component Task Tracker receives those tasks from the Job Tracker and perform those tasks by using pre-defined (defined by client) MapReduce components, the map and reduce functions).
5) Once all Task Trackers finish their job, the Job Tracker takes the computed results and combines them to the final result.
6) Finally the Hadoop System returns the results to the client.

Hadoop V2 Architecture

Hadoop V1 Architecture has lot of limitations and drawbacks. So that Hadoop Community evaluated and redesigned this Architecture into Hadoop V2 Architecture. Some of the limitations of Hadoop V1 are listed below.

1) Data has to be loaded into the Hadoop System before any processing can be made. So it is not suitable for real time data processing.
2) It supports up to 4000 Nodes per Cluster. This means it has scalability problems.
3) It has a single component (Job Tracker) per cluster to perform many activities like resource management, job scheduling, job monitoring, re-scheduling Jobs and etc. So this means it has a single point of failure.
4) It can only run Map/Reduce jobs.
5) Supports only one namespace per cluster.

Hadoop V2 aims to solve these problems. The major components of Hadoop V2 are HDFS and YARN. The

purpose of addition of YARN in this architecture was to overcome the shortcomings of Hadoop V1. Figure 5 below shows the architecture of Hadoop V2.

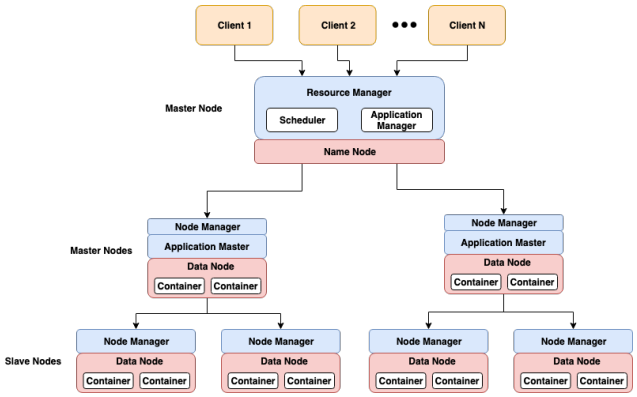

Fig. 5. The Architecture of Hadoop V2

The execution of a user program is carried out in the steps listed below:

1) Clients submit their job (application) to the YARN Resource Manager.
2) The Application Manager which is in the Resource Manager, bootstraps the Application Master instance for the application.
3) The Application Master registers with the Resource Manager and requests containers.
4) The Application Master communicates with Node Managers to launch the containers it has been granted.
5) Then the Application Master manages application execution. During execution, the application provides progress and status information to the Application master. The client can monitor the application's status by querying the Resource Manager or by communicating directly with the Application Master.
6) The Application Master reports completion of the application to the Resource Manager.
7) The Application Master un-registers with the Resource Manager, which cleans up the Application Master container for future use.

Open Problems & Suggested Solution

The major limitations of Hadoop:

1) **Replication Factor Problem:** The replication factor of HDFS decides the number of replicated copies of a block to keep in the file system. The default value of the replication factor is three but can be defined by the user too. Research [7] has shown that while HDFS deals well with increasing the replication factor, it experiences problems with decreasing it. This leads to unbalanced data and performance degradation.

   Possible Solution: In order to solve this problem the Hadoop replica placement algorithm, which decides the DataNodes that will contain the replicas, can be improved such that it leads to more balanced data and better performance.

2) **Not Suitable for Small Datasets:** As mentioned above in the HDFS section, file contents are broken down into chunks of 64-128MB as default. When the file is much smaller than 64MB but in large quantity (contains an abundance of data, e.g. text file), the name node will get overloaded due to the huge namespace. Moreover, HDFS lacks the ability to efficiently support the random reading of small files because of its high capacity design.

   Possible solution: HBase [6] is a distributed column-oriented data store built on top of HDFS. It is ideally suited for random write and read of data that is stored in HDFS and allows for dynamic changes in storage. All the data is being stored in the form of tables, rows and columns rather than in a form of a typical size.

3) **Security Concern:** Hadoop has no encryption at the storage and network level, making it vulnerable to attacks.

   Possible solution: to this problem would be to perform computations on encrypted data. The survey research paper [8] talks about some potential ways using which computations can be performed on encrypted data. Also, an extra layer of encryption/decryption can be added but that would increase the overall complexity defeating the purpose of Hadoop.

4) **No Real-time Data Processing:** Apache Hadoop is for batch processing, which means it takes a huge amount of data in input, processes it and produces the result. An output can be delayed significantly, which is bad for Real-time Data processing.

   Possible Solution: Apache Spark supports stream processing, which involves continuous input and output of data. It emphasizes on the velocity of the data and data processes within a small period of time.

Conclusion

Hadoop is becoming one of the most popular and powerful distributed architecture that is suitable for dealing with data analysis on huge amounts of data. The core components, HDFS, YARN and MapReduce are responsible for distributed data storage and parallel data processing. Moreover, Hadoop being improved from version 1 to version 2, based on MapReduce and YARN respectively, increased its ability of resource management. The future direction of Hadoop architecture can possibly be on the area of security and real-time processing based on current limitations.

References

[1] Apache Hadoop. http://hadoop.apache.org/
[2] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The hadoop distributed file system", 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, pp. 1-10 (2010)

[3] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", Communications of the ACM, vol. 51, pp. 107-113 (2008)
[4] RAID. https://en.wikipedia.org/wiki/RAID
[5] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed and E. Baldeschwieler, "Apache hadoop YARN: yet another resource negotiator", Proceedings of the 4th annual Symposium on Cloud Computing (2013)
[6] M. N. Vora, "Hadoop-HBase for large-scale data", Proceedings of International Conference on Computer Science and Network Technology (2011)
[7] H. E. Ciritoglu, T. Saber, T.S. Buda and J. Murphy, "Towards a Better Replica Management for Hadoop Distributed File System" Big Data Congress, (2018)
[8] E. Saleh, A. Alsa'deh, A. Kayed and C. Meinel, "Processing Over Encrypted Data: Between Theory and Practice", SIGMOD Rec. 45, (2016)