**CHAPTER 1**

INTRODUCTION

Hadoop is the Apache Software Foundation top-level project that holds the various Hadoop subprojects that graduated from the Apache Incubator. The Hadoop project provides and supports the development of open source software that supplies a framework for the development of highly scalable distributed computing applications. The Hadoop framework handles the processing details, leaving developers free to focus on application logic.

Hadoop was created by Doug Cutting who named it after his son's toy elephant. It was originally developed to support distribution for the Nutch search engine project.

**1.1 HISTORY OF HADOOP**

Hadoop started out as a subproject of Nutch, which in turn was a subproject of Apache Lucene. Doug Cutting founded all three projects, and each project was a logical progression of the previous one. Lucene is a full-featured text indexing and searching library. Given a text collection, a developer can easily add search capability to the documents using the Lucene engine. Nutch is the most ambitious extension of Lucene. It tries to build a complete web search engine using Lucene as its core component. Around 2004, Google published two papers describing the Google File System (GFS) and the MapReduce frame work. Google claimed to use these two technologies for scaling its own search system. Doug Cutting immediately saw the applicability of these technologies to Nutch, and his team implemented the new framework and ported Nutch to it. The new implementation immediately boosted Nutch’s scalability. It started to handle several hundred million web pages and could run on clusters of dozens of nodes. Doug realized that a dedicated project to flesh out the two technologies was needed to get to web scale, and Hadoop was born. Yahoo! hired Doug in January 2006 to work with a dedicated team on improving Hadoop as an open source project. Two years later, Hadoop achieved the status of an Apache Top Level Project. Later, on February 19, 2008, Yahoo! announced that Hadoop running on a 10,000+.

**CHAPTER 2**

HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

**2.1 INTRODUCTION**

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets.

**2.1.1 DISTRIBUTED FILE SYSTEM BASICS**

A distributed file system is designed to hold a large amount of data and provide access to this data to many clients distributed across a network. There are a number of distributed file systems that solve this problem in different ways.

NFS , the Network File System is the oldest distributed file system in use. NFS provides remote access to a single logical volume stored on a single machine. An NFS server makes a portion of its local file system visible to external clients. The clients can then mount this remote file system directly into their own Linux file system, and interact with it as though it were part of the local drive.

One of the primary advantages of this model is its transparency. Clients do not need to be particularly aware that they are working on files stored remotely.

But as a distributed file system, it is limited in its power. The files in an NFS volume all reside on a single machine. This means that it will only store as much information as can be stored in one machine, and does not provide any reliability guarantees if that machine goes down (e.g., by replicating the files to other servers). Finally, as all the data is stored on a single machine, all the clients must go to this machine to retrieve their data. This can overload the server if a large number of clients must be handled. Clients must also always copy the data to their local machines before they can operate on it.

**2.1.2 WHY IS HDFS?**

HDFS is chosen for a set of functionalities those are not provided by the other distributed file systems.

* HDFS is designed to store a very large amount of information (terabytes or petabytes). This requires spreading the data across a large number of machines. It also supports much larger file sizes than NFS.
* HDFS should store data reliably. If individual machines in the cluster malfunction, data should still be available.
* HDFS should provide fast, scalable access to this information. It should be possible to serve a larger number of clients by simply adding more machines to the cluster.
* HDFS should integrate well with Hadoop MapReduce(described later), allowing data to be read and computed upon locally when possible.

But HDFS is not a general purpose file system like NFS. There are some trade-offs too. In particular,

* Applications that use HDFS are assumed to perform long sequential streaming reads from files. HDFS is optimized to provide streaming read performance.
* Data will be written to the HDFS once and then read several times; updates to files after they have already been closed are not supported. (An extension to Hadoop will provide support for appending new data to the ends of files; it is scheduled to be included in Hadoop 0.19 but is not available yet.)
* Due to the large size of files, and the sequential nature of reads, the system does not provide a mechanism for local caching of data. The overhead of caching is great enough that data should simply be re-read from HDFS source.
* Individual machines are assumed to fail on a frequent basis, both permanently and intermittently. The cluster must be able to withstand the complete failure of several machines, possibly many happening at the same time (e.g., if a rack fails all together). While performance may degrade proportional to the number of machines lost, the system as a whole should not become overly slow, nor should information be lost. Data replication strategies combat this problem.

The design of HDFS is based on the design of **GFS**, the Google File System

**2.2 HDFS ARCHITECTURE**

**2.2.1**  HDFS has a master /slave architecture. An HDFS cluster consists of a Namenode and a number of Datanodes.

Namenode acts as a master server and manages the namespace of the file system and regulates access to files. Whereas the datanodes manage the storages of replicas of blocks. HDFS is a block-structured file system: individual files are broken into blocks of a fixed size. These blocks are stored across a cluster of one or more machines with data storage capacity. A file can be made of several blocks, and they are not necessarily stored on the same machine; the target machines which hold each block are chosen randomly on a block-by-block basis. Thus access to a file may require the cooperation of multiple machines, but supports file sizes far larger than a single-machine DFS; individual files can require more space than a single hard drive could hold.

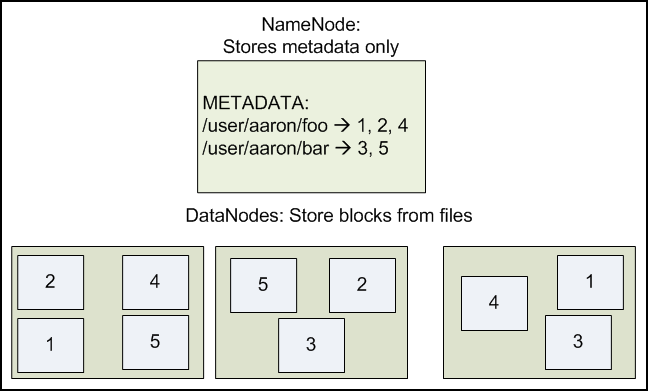


Figure 2.1 the namenode and datanode structure where the replication factor of the files are 2

The default block size in HDFS is 64MB -- orders of magnitude larger. This allows HDFS to decrease the amount of metadata storage required per file (the list of blocks per file will be smaller as the size of individual blocks increases). Furthermore, it allows for fast streaming reads of data, by keeping large amounts of data sequentially laid out on the disk.

**2.2.2FILE SYSTEM NAMESPACE**

It is important for this file system to store its metadata reliably. Furthermore, while the file data is accessed in a write once and read many model, the metadata structures (e.g., the names of files and directories) can be modified by a large number of clients concurrently. It is important that this information is never desynchronized. Therefore, it is all handled by a single machine, the **NameNode**. The NameNode stores all the metadata for the file system. Because of the relatively low amount of metadata per file (it only tracks file names, permissions, and the locations of each block of each file), all of this information can be stored in the main memory of the NameNode machine, allowing fast access to the metadata.

To open a file, a client contacts the NameNode and retrieves a list of locations for the blocks that comprise the file. These locations identify the DataNodes which hold each block. Clients then read file data directly from the DataNode servers, possibly in parallel. The NameNode is not directly involved in this bulk data transfer, keeping its overhead to a minimum.

Of course, NameNode information must be preserved even if the NameNode machine fails; there are multiple redundant systems that allow the NameNode to preserve the file system's metadata even if the NameNode itself crashes irrecoverably. NameNode failure is more severe for the cluster than DataNode failure. While individual DataNodes may crash and the entire cluster will continue to operate, the loss of the NameNode will render the cluster inaccessible until it is manually restored. Fortunately, as the NameNode's involvement is relatively minimal, the odds of it failing are considerably lower than the odds of an arbitrary DataNode failing at any given point in time.

Because HDFS stores files as a set of large blocks across several machines, these files are not part of the ordinary file system. Typing ls on a machine running a DataNode daemon will display the contents of the ordinary Linux file system being used to host the Hadoop services -- but it will not include any of the files stored inside the HDFS. This is because HDFS runs in a **separate namespace**, isolated from the contents of your local files. The files inside HDFS (or more accurately: the blocks that make them up) are stored in a particular directory managed by the DataNode service, but the files will named only with block ids. You cannot interact with HDFS-stored files using ordinary Linux file modification tools (e.g., ls, cp, mv, etc). However, HDFS does come with its own utilities for file management, which act very similar to these familiar tools.

**2.2.3 DATA REPLICATION**

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later.

**2.2.4 PERSISTENCE OF FILE SYSTEM METADATA**

The NameNode uses a transaction log called the EditLog to persistently record every change that occurs to file system metadata. For example, creating a new file in HDFS causes the NameNode to insert a record into the EditLog indicating this. Similarly, changing the replication factor of a file causes a new record to be inserted into the EditLog. The NameNode uses a file in its local host OS file system to store the EditLog. The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the FsImage. The FsImage is stored as a file in the NameNode’s local file system too.

The NameNode keeps an image of the entire file system namespace and file Blockmap in memory.

When the NameNode starts up, it reads the FsImage and EditLog from disk, applies all the transactions from the EditLog to the in-memory representation of the FsImage, and flushes out this new version into a new FsImage on disk. It can then truncate the old EditLog because its transactions have been applied to the persistent FsImage. This process is called a checkpoint. In the current implementation, a checkpoint only occurs when the NameNode starts up. Work is in progress to support periodic checkpointing in the near future.

**2.2.5 THE NEED OF SECONDARY NAMENODE**

Since NameNode merges fsimage and edits files only during start up, the edits log file could get very large over time on a busy cluster. Another side effect of a larger edits file is that next restart of NameNode takes longer.

The secondary NameNode merges the fsimage and the edits log files periodically and keeps edits log size within a limit. It is usually run on a different machine than the primary NameNode since its memory requirements are on the same order as the primary NameNode.

Secondary namenode has been deprecated. There is an upgradation called checkpoint node or back up node.

**2.2.5.1 CHECKPOINT NODE:**

The Checkpoint node periodically creates checkpoints of the namespace. It downloads fsimage and edits from the active NameNode, merges them locally, and uploads the new image back to the active NameNode. The Checkpoint node usually runs on a different machine than the NameNode since its memory requirements are on the same order as the NameNode.

**2.2.5.2 BACK UP NODE:**

The Backup node provides the same checkpointing functionality as the Checkpoint node, as well as maintaining an in-memory, up-to-date copy of the file system namespace that is always synchronized with the active NameNode state. Along with accepting a journal stream of file system edits from the NameNode and persisting this to disk, the Backup node also applies those edits into its own copy of the namespace in memory, thus creating a backup of the namespace.

The Backup node does not need to download fsimage and edits files from the active NameNode in order to create a checkpoint, as would be required with a Checkpoint node or Secondary NameNode, since it already has an up-to-date state of the namespace state in memory. The Backup node checkpoint process is more efficient as it only needs to save the namespace into the local fsimage file and reset edits.

As the Backup node maintains a copy of the namespace in memory, its RAM requirements are the same as the NameNode.

**2.3 DATA ORGANIZATION**

**2.3.1 DATA BLOCKS**

A typical block size used by HDFS is 64 MB. Thus, an HDFS file is chopped up into 64 MB chunks, and if possible, each chunk will reside on a different DataNode.

**2.3.2 STAGING**

A client request to create a file does not reach the NameNode immediately. In fact, initially the HDFS client caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates data worth over one HDFS block size, the client contacts the NameNode. The NameNode inserts the file name into the file system hierarchy and allocates a data block for it. The NameNode responds to the client request with the identity of the DataNode and the destination data block. Then the client flushes the block of data from the local temporary file to the specified DataNode. When a file is closed, the remaining un-flushed data in the temporary local file is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. If the NameNode dies before the file is closed, the file is lost.

**2.3.3 REPLICATION PIPELINING**

Suppose the HDFS file has a replication factor of three. When the local file accumulates a full block of user data, the client retrieves a list of DataNodes from the NameNode. This list contains the DataNodes that will host a replica of that block. The client then flushes the data block to the first DataNode. The first DataNode starts receiving the data in small portions (4 KB), writes each portion to its local repository and transfers that portion to the second DataNode in the list. And the process goes on with the next Datanodes in the list.

**2.4 ACCESSIBILITY**

HDFS can be accessed from applications in many different ways. Natively, HDFS provides a Java API for applications to use. A C language wrapper for this Java API is also available. In addition, an HTTP browser can also be used to browse the files of an HDFS instance.

**2.4.1 FS SHELL**

HDFS allows user data to be organized in the form of files and directories. It provides a command line interface called FS shell that lets a user interact with the data in HDFS. The syntax of this command set is similar to other shells.

|  |  |
| --- | --- |
| ACTION | COMMAND |
| Listing files and directories in DFS | $ bin/hadoop dfs -ls |

Table 2.1 Listing commands to operate on HDFS

**2.4.2 Browser Interface**

A typical HDFS install configures a web server to expose the HDFS namespace through a configurable TCP port. This allows a user to navigate the HDFS namespace and view the contents of its files using a web browser.

The one core part, HDFS, is now quite understood, we should now move to the next important and interesting part.

**CHAPTER 3**

MAPREDUCE – A NEW WAY TO SOLVE PROBLEMS

3.1 INTRODUCTION

The user configures and submits a MapReduce job (or just *job* for short) to the framework, which will decompose the job into a set of map tasks, shuffles, a sort, and a set of reduce tasks. The framework will then manage the distribution and execution of the tasks, collect the output, and report the status to the user.

1>JOB CONFIGURATION

2> INPUT FORMAT

3> INPUT LOCATION

4> MAP FUNCTION

5> NUMBER OF REDUCE TASKS

6> REDUCE FUNCTION

7> OUTPUT KEY TYPE

8> OUTPUT VALUE TYPE

8> OUTPUT FORMAT

9> OUTPUT LOCATION

INPUT SPLITTING & DISTRIBUTION

START OF MAP TASKS

SHUFFLE, PARTITION/SORT PER MAP OUTPUT

COLLECTION OF FINAL OUTPUT

START OF REDUCE TASKS

Merge Sort for

Map Outputs for Each

Reduce Task

Provided by user

Provided by hadoop framework

Fig 3.1 parts of a MapReduce job

The user is responsible for handling the job setup, specifying the input location(s), specifying

the input, and ensuring the input is in the expected format and location. The framework

is responsible for distributing the job among the TaskTracker nodes of the cluster; running the map, shuffle, sort, and reduce phases; placing the output in the output directory; and informing the user of the job-completion status.

**3.2 JOBTRACKER AND TASKTRACKER**

The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

Minimally, applications specify the input/output locations and supply *map* and *reduce* functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the *job configuration*. The Hadoop *job client* then submits the job (jar/executable etc.) and configuration to the JobTracker which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

Limitations of this approach are:

* The allocation of work to task trackers is very simple. Every task tracker has a number of available *slots*. Every active map or reduce task takes up one slot. The Job Tracker allocates work to the tracker nearest to the data with an available slot. There is no consideration of the current active load of the allocated machine, and hence its actual availability.
* If one task tracker is very slow, it can delay the entire MapReduce operation -especially towards the end of a job, where everything can end up waiting for a single slow task. With speculative-execution enabled, however, a single task can be executed on multiple slave nodes. So this technology makes the networking very efficient

**3.3 HDFS AND MAPREDUCE**

SECONDARY NAME NODE

NAMENODE

JOBTRACKER

DATANODE

TASKTRACKER

DATANODE

TASK TRACKER

DATANODE

TASKTRACKER

Fig 3.2 Master /slave architecture where Namenode and JobTracker is master and Datanode and TaskTrackers are the slaves

A Hadoop cluster will include a single master and multiple worker nodes. The master node consists of a jobtracker, namenode. A slave or *worker node* consists of a datanode and tasktracker.

In a small cluster, the master node consists of a jobtracker, tasktracker,datanode and namenode. A slave or *worker node* consists of a datanode and tasktracker.

**3.4 HOW MAPREDUCE PROGRAM WORKS**

The MapReduce framework operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

Input and Output types of a MapReduce job:

(input) <k1, v1> -> **map** -> <k2, v2> -> **combine\*** -> <k2, v2> -> **reduce** -> <k3, v3> (output)

Note that the combine phase may run zero or more times in this process.

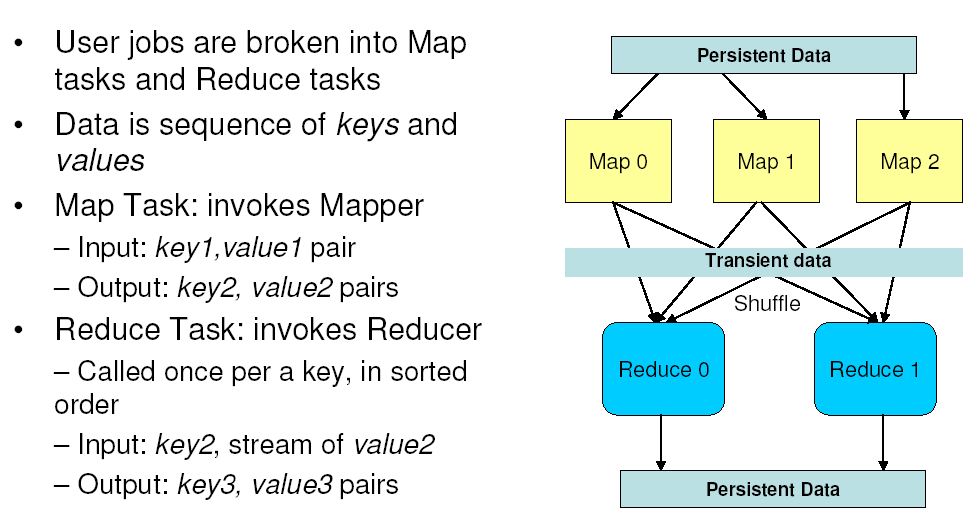


Fig 3.3 MapReduce Work flow Diagram3.5

Here, each document is split into words, and each word is counted initially with a "1" value by the *Map* function, using the word as the result key. The framework puts together all the pairs with the same key and feeds them to the same call to *Reduce*, thus this function just needs to sum all of its input values to find the total appearances of that word.

**3.4 Benefits of MapReduce**

* Large data sets can be processed in parallel.
* Useful technique for implementing distributed grep, distributed sort, matrix multiplication, graph search algorithms, machine learning algorithms and some parallel algorithms.
* MapReduce achieves reliability by parceling out a number of operations on the set of data to each node in the network. Each node is expected to report back periodically with completed work and status updates. If a node falls silent for longer than that interval, the master node (similar to the master server in the Google File System) records the node as dead and sends out the node's assigned work to other nodes.

**3.5 Limitations of MapReduce**

* For maximum parallelism, you need the Maps and Reduces to be stateless, to not depend on any data generated in the same MapReduce job. You cannot control the order in which the maps run, or the reductions.
* It is very inefficient if you are repeating similar searches again and again. A database with an index will always be faster than running an MR job over unindexed data. However, if that index needs to be regenerated whenever data is added, and data is being added continually, MR jobs may have an edge. That inefficiency can be measured in both CPU time and power consumed.
* In the Hadoop implementation Reduce operations do not take place until all the Maps are complete (or have failed and been skipped). As a result, you do not get any data back until the entire mapping has finished.
* There is a general assumption that the output of the reduce is smaller than the input to the Map. That is, you are taking a large datasource and generating smaller final values.
* MapReduce works on independent data sets, so recursion can not be done using it.,as it is not working with data dependent algorithm.