Module -2: INTRODUCTION TO HADOOP

2.1 Hadoop

1. Hadoop is an open source framework that supports the processing of large data sets in a distributed computing environment.
2. Hadoop consists of MapReduce, the Hadoop distributed file system (HDFS) and a number of related projects such as Apache Hive, HBase and Zookeeper. MapReduce and Hadoop distributed file system (HDFS) are the main component of Hadoop.
3. Apache Hadoop is an open-source, free and Java based software framework offers a powerful distributed platform to store and manage Big Data.
4. It is licensed under an Apache V2 license.
5. It runs applications on large clusters of commodity hardware and it processes thousands of terabytes of data on thousands of the nodes. Hadoop is inspired from Google's MapReduce and Google File System (GFS) papers.
6. The major advantage of Hadoop framework is that it provides reliability and high availability.

2.2 Use of Hadoop

There are many advantages of using Hadoop:

1. Robust and Scalable – We can add new nodes as needed as well modify them.
2. Affordable and Cost Effective – We do not need any special hardware for running Hadoop. We can just use commodity server.
3. Adaptive and Flexible – Hadoop is built keeping in mind that it will handle structured and unstructured data.
4. Highly Available and Fault Tolerant – When a node fails, the Hadoop framework automatically fails over to another node.

2.3 Core Hadoop Components

There are two major components of the Hadoop framework and both of them does two of the important task for it.

1. Hadoop MapReduce is the method to split a larger data problem into smaller chunk and distribute it to many different commodity servers. Each server have their own set of resources and they have processed them locally. Once the commodity server has processed the data they send it back collectively to main server. This is effectively a process where we process large data effectively and efficiently
2. Hadoop Distributed File System (HDFS) is a virtual file system. There is a big difference between any other file system and Hadoop. When we move a file on HDFS, it is automatically split into many small pieces. These small chunks of the file are replicated and stored on other servers (usually 3) for the fault tolerance or high availability.
3. Namenode: Namenode is the heart of the Hadoop system. The NameNode manages the file system namespace. It stores the metadata information of the data blocks. This metadata is

stored permanently on to local disk in the form of namespace image and edit log file. The NameNode also knows the location of the data blocks on the data node. However the NameNode does not store this information persistently. The NameNode creates the block to DataNode mapping when it is restarted. If the NameNode crashes, then the entire Hadoop system goes down. Read more about Namenode

4. Secondary Namenode: The responsibility of secondary name node is to periodically copy and merge the namespace image and edit log. In case if the name node crashes, then the namespace image stored in secondary NameNode can be used to restart the NameNode.
5. DataNode: It stores the blocks of data and retrieves them. The DataNodes also reports the blocks information to the NameNode periodically.
6. Job Tracker: Job Tracker responsibility is to schedule the client's jobs. Job tracker creates map and reduce tasks and schedules them to run on the DataNodes (task trackers). Job Tracker also checks for any failed tasks and reschedules the failed tasks on another DataNode. Job tracker can be run on the NameNode or a separate node.
7. Task Tracker: Task tracker runs on the DataNodes. Task trackers responsibility is to run the map or reduce tasks assigned by the NameNode and to report the status of the tasks to the NameNode.

Besides above two core components Hadoop project also contains following modules as well.

1. Hadoop Common: Common utilities for the other Hadoop modules
2. Hadoop Yarn: A framework for job scheduling and cluster resource management

2.4 RDBMS

Why can't we use databases with lots of disks to do large-scale batch analysis? Why is MapReduce needed?

The answer to these questions comes from another trend in disk drives: seek time is improving more slowly than transfer rate. Seeking is the process of moving the disk's head to a particular place on the disk to read or write data. It characterizes the latency of a disk operation, whereas the transfer rate corresponds to a disk's bandwidth.

If the data access pattern is dominated by seeks, it will take longer to read or write large portions of the dataset than streaming through it, which operates at the transfer rate. On the other hand, for updating a small proportion of records in a database, a tradi- tional B-Tree (the data structure used in relational databases, which is limited by the rate it can perform seeks) works well. For updating the majority of a database, a B-Tree is less efficient than MapReduce, which uses Sort/Merge to rebuild the database.

In many ways, MapReduce can be seen as a complement to an RDBMS. (The differences between the two systems. MapReduce is a good fit for problems that need to analyze the whole dataset, in a batch fashion, particularly for ad hoc analysis. An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data. MapReduce suits applications where the data is written once, and read many times, whereas a relational database is good for datasets that are continually updated.

Another difference between MapReduce and an RDBMS is the amount of structure in the datasets that they operate on. Structured data is data that is organized into entities that have a defined format, such as XML documents or database tables that conform to a particular predefined schema. This is the realm of the RDBMS. Semi-structured data, on the other hand, is looser, and though there may be a schema, it is often ignored, so it may be used only as a guide to the structure of the data: for example, a spreadsheet, in which the structure is the grid of cells, although the cells themselves may hold any form of data. Unstructured data does not have any particular internal structure: for example, plain text or image data. MapReduce works well on unstructured or semi- structured data, since it is designed to interpret the data at processing time. In other words, the input keys and values for MapReduce are not an intrinsic property of the data, but they are chosen by the person analyzing the data.

Relational data is often normalized to retain its integrity and remove redundancy. Normalization poses problems for MapReduce, since it makes reading a record a non- local operation, and one of the central assumptions that MapReduce makes is that it is possible to perform (high-speed) streaming reads and writes.

A web server log is a good example of a set of records that is not normalized (for ex- ample, the client hostnames are specified in full each time, even though the same client may appear many times), and this is one reason that logfiles of all kinds are particularly well-suited to analysis with MapReduce.

MapReduce is a linearly scalable programming model. The programmer writes two functions—a map function and a reduce function—each of which defines a mapping from one set of key-value pairs to another. These functions are oblivious to the size of the data or the cluster that they are operating on, so they can be used unchanged for a small dataset and for a massive one. More important, if you double the size of the input data, a job will run twice as slow. But if you also double the size of the cluster, a job will run as fast as the original one. This is not generally true of SQL queries.

Over time, however, the differences between relational databases and MapReduce sys- tems are likely to blur—both as relational databases start incorporating some of the ideas from MapReduce (such as Aster Data’s and Greenplum’s databases) and, from the other direction, as higher-level query languages built on MapReduce (such as Pig and Hive) make MapReduce systems more approachable to traditional database programmers.

2.5 A BRIEF HISTORY OF HADOOP

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Hadoop has its origins in Apache Nutch, an open source web search engine, itself a part of the Lucene project.

Building a web search engine from scratch was an ambitious goal, for not only is the software required to crawl and index websites complex to write, but it is also a challenge to run without a dedicated operations team, since there are so many moving parts. It’s expensive, too: Mike Cafarella and Doug Cutting estimated a system supporting a 1-billion-page index would cost around half a million dollars in hardware, with a monthly running cost of $30,000.Nevertheless, they believed it was a worthy goal, as it would open up and ultimately democratize search engine algorithms.

Nutch was started in 2002, and a working crawler and search system quickly emerged. However, they realized that their architecture wouldn’t scale to the billions of pages on the Web. Help was at hand with the publication of a paper in 2003 that described the architecture of Google’s distributed filesystem, called GFS, which was being used in production at Google.[11] GFS, or something like it, would solve their storage needs for the very large files generated as a part of the web crawl and indexing process. In par- ticular, GFS would free up time being spent on administrative tasks such as managing storage nodes. In 2004, they set about writing an open source implementation, the Nutch Distributed Filesystem (NDFS).

In 2004, Google published the paper that introduced MapReduce to the world.[12] Early in 2005, the Nutch developers had a working MapReduce implementation in Nutch, and by the middle of that year all the major Nutch algorithms had been ported to run using MapReduce and NDFS.

NDFS and the MapReduce implementation in Nutch were applicable beyond the realm of search, and in February 2006 they moved out of Nutch to form an independent subproject of Lucene called Hadoop. At around the same time, Doug Cutting joined Yahoo!, which provided a dedicated team and the resources to turn Hadoop into a system that ran at web scale (see sidebar). This was demonstrated in February 2008 when Yahoo! announced that its production search index was being generated by a 10,000-core Hadoop cluster.[13]

In January 2008, Hadoop was made its own top-level project at Apache, confirming its success and its diverse, active community. By this time, Hadoop was being used by many other companies besides Yahoo!, such as Last.fm, Facebook, and the New York Times.

In one well-publicized feat, the New York Times used Amazon’s EC2 compute cloud to crunch through four terabytes of scanned archives from the paper converting them to PDFs for the Web.[14] The processing took less than 24 hours to run using 100 ma- chines, and the project probably wouldn’t have been embarked on without the com- bination of Amazon’s pay-by-the-hour model (which allowed the NYT to access a large number of machines for a short period) and Hadoop’s easy-to-use parallel programming model.

In April 2008, Hadoop broke a world record to become the fastest system to sort a terabyte of data. Running on a 910-node cluster, Hadoop sorted one terabyte in 209 seconds (just under 3½ minutes), beating the previous year’s winner of 297 seconds. In November of the same year, Google reported that its MapReduce implementation sorted one ter- abyte in 68 seconds.[15] As the first edition of this book was going to press (May 2009), it was announced that a team at Yahoo! used Hadoop to sort one terabyte in 62 seconds.

2.6 ANALYZING THE DATA WITH HADOOP

To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job. After some local, small-scale testing, we will be able to run it on a cluster of machines.

MAP AND 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 programmer also specifies two functions: the map function and the reduce function.
The input to our map phase is the raw NCDC data. We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file, but as we have no need for this, we ignore it.

Our map function is simple. We pull out the year and the air temperature, since these are the only fields we are interested in. In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year. The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.

To visualize the way the map works, consider the following sample lines of input data (some unused columns have been dropped to fit the page, indicated by ellipses):

```
0067011990999991950051507004...9999999N9+00001+99999999999...

0043011990999991950051512004...9999999N9+00221+99999999999...

0043011990999991950051518004...9999999N9-00111+99999999999...

0043012650999991949032412004...0500001N9+01111+99999999999...

0043012650999991949032418004...0500001N9+00781+99999999999...
```

These lines are presented to the map function as the key-value pairs:

```
(0,   0067011990999991950051507004...9999999N9+00001+99999999999...)

(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)

(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)

(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)

(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

The keys are the line offsets within the file, which we ignore in our map function. The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted as integers):

(1950, 0)

(1950, 22)

(1950, −11)

(1949, 111)

(1949, 78)

The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key. So, continuing the example, our reduce function sees the following input:

(1949, [111, 78])

(1950, [0, 22, −11])

Each year appears with a list of all itsair temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

(1949, 111)

(1950, 22)

This is the final output: the maximum global temperature recorded in each year.

The whole data flow is illustrated in 2.2. At the bottom of the diagram is a Unix pipeline, which mimics the whole MapReduce flow, and which we will see again later in the chapter when we look at Hadoop Streaming.

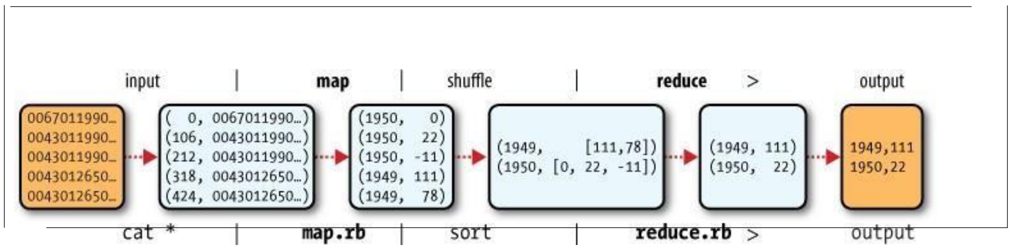

Figure 2-1. MapReduce logical data flow

JAVA MAPREDUCE

Having run through how the MapReduce program works, the next step is to express it in code.

We need three things: a map function, a reduce function, and some code to run the job. The map function is represented by the Mapper class, which declares an abstract map() method.

The Mapper class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function. For the present example, the input key is a long integer offset, the input value is a line of text, the output key is a year, and the output value is an air temperature (an integer). Rather than use built-in Java types, Hadoop provides its own set of basic types that are opti- mized for network serialization. These are found in the org.apache.hadoop.io package. Here we use LongWritable, which corresponds to a Java Long, Text (like Java String), and IntWritable (like Java Integer).

The map() method is passed a key and a value. We convert the Text value containing the line of input into a Java String, then use its substring() method to extract the columns we are interested in.

The map() method also provides an instance of Context to write the output to. In this case, we write the year as a Text object (since we are just using it as a key), and the temperature is wrapped in an IntWritable. We write an output record only if the tem- perature is present and the quality code indicates the temperature reading is OK.

Again, four formal type parameters are used to specify the input and output types, this time for the reduce function. The input types of the reduce function must match the output types of the map function: Text and IntWritable. And in this case, the output types of the reduce function are Text and IntWritable, for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

A Job object forms the specification of the job. It gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specify the name of the JAR file, we can pass a class in the Job's setJarByClass() method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.

Having constructed a Job object, we specify the input and output paths. An input path is specified by calling the static addInputPath() method on FileInputFormat, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern. As the name suggests, addInputPath() can be called more than once to use input from multiple paths.

The output path (of which there is only one) is specified by the static setOutput Path() method on FileOutputFormat. It specifies a directory where the output files from the reducer functions are written. The directory shouldn't exist before running the job, as Hadoop will complain and not run the job. This precaution is to prevent data loss(it can be very annoying to accidentally overwrite the output of a long job with another).

Next, we specify the map and reduce types to use via the setMapperClass() and

setReducerClass() methods.

The setOutputKeyClass() and setOutputValueClass() methods control the output types for the map and the reduce functions, which are often the same, as they are in our case. If they are different, then the map output types can be set using the methods setMapOutputKeyClass() and setMapOutputValueClass().

The input types are controlled via the input format, which we have not explicitly set since we are using the default TextInputFormat.

After setting the classes that define the map and reduce functions, we are ready to run the job. The waitForCompletion() method on Job submits the job and waits for it to finish. The method's boolean argument is a verbose flag, so in this case the job writes information about its progress to the console.

The return value of the waitForCompletion() method is a boolean indicating success (true) or failure (false), which we translate into the program's exit code of 0 or 1.

A TEST RUN

After writing a MapReduce job, it's normal to try it out on a small dataset to flush out any immediate problems with the code. First install Hadoop in standalone mode— there are instructions for how to do this in Appendix A. This is the mode in which Hadoop runs using the local filesystem with a local job runner. Then install and compile the examples using the instructions on the book's website.

When the hadoop command is invoked with a classname as the first argument, it launches a JVM to run the class. It is more convenient to use hadoop than straight java since the former adds the Hadoop libraries (and their dependencies) to the class- path and picks up the Hadoop configuration, too. To add the application classes to the classpath, we've defined an environment variable called HADOOP_CLASSPATH, which the hadoop script picks up.

The last section of the output, titled "Counters," shows the statistics that Hadoop generates for each job it runs. These are very useful for checking whether the amount of data processed is what you expected. For example, we can follow the number of records that went through the system: five map inputs produced five map outputs, then five reduce inputs in two groups produced two reduce outputs.

The output was written to the output directory, which contains one output file per reducer. The job had a single reducer, so we find a single file, named part-r-00000:

```
% cat output/part-r-00000

1949 111

1950 22
```

This result is the same as when we went through it by hand earlier. We interpret this as saying that the maximum temperature recorded in 1949 was 11.1°C, and in 1950 it was 2.2°C.

THE OLD AND THE NEW JAVA MAPREDUCE APIS

The Java MapReduce API used in the previous section was first released in Hadoop

0.20.0. This new API, sometimes referred to as "Context Objects," was designed to

make the API easier to evolve in the future. It is type-incompatible with the old, how- ever, so applications need to be rewritten to take advantage of it.

The new API is not complete in the 1.x (formerly 0.20) release series, so the old API is recommended for these releases, despite having been marked as deprecated in the early

0.20 releases. (Understandably, this recommendation caused a lot of confusion so the deprecation warning was removed from later releases in that series.)

Previous editions of this book were based on 0.20 releases, and used the old API throughout (although the new API was covered, the code invariably used the old API). In this edition the new API is used as the primary API, except where mentioned. How- ever, should you wish to use the old API, you can, since the code for all the examples in this book is available for the old API on the book's website.[1]

There are several notable differences between the two APIs:

- The new API favors abstract classes over interfaces, since these are easier to evolve. For example, you can add a method (with a default implementation) to an abstract class without breaking old implementations of the class[2]. For example, the Mapper and Reducer interfaces in the old API are abstract classes in the new API.
- The new API is in the org.apache.hadoop.mapreduce package (and subpackages). The old API can still be found in org.apache.hadoop.mapred.
- The new API makes extensive use of context objects that allow the user code to communicate with the MapReduce system. The new Context, for example, essen- tially unifies the role of the JobConf, the OutputCollector, and the Reporter from the old API.
- In both APIs, key-value record pairs are pushed to the mapper and reducer, but in addition, the new API allows both mappers and reducers to control the execution flow by overriding the run() method. For example, records can be processed in batches, or the execution can be terminated before all the records have been pro- cessed. In the old API this is possible for mappers by writing a MapRunnable, but no equivalent exists for reducers.
- Configuration has been unified. The old API has a special JobConf object for job configuration, which is an extension of Hadoop's vanilla Configuration object (used for configuring daemons. In the new API, this distinction is dropped, so job configuration is done through a Configuration.
- Job control is performed through the Job class in the new API, rather than the old

JobClient, which no longer exists in the new API.

- Output files are named slightly differently: in the old API both map and reduce outputs are named part-nnnnn, while in the new API map outputs are named part- m-nnnnn, and reduce outputs are named part-r-nnnnn (where nnnnn is an integer designating the part number, starting from zero).
- User-overridable methods in the new API are declared to throw java.lang.Inter ruptedException. What this means is that you can write your code to be reponsive to interupts so that the framework can gracefully cancel long-running operations if it needs to[3].
- In the new API the reduce() method passes values as a java.lang.Iterable, rather than a java.lang.Iterator (as the old API does). This change makes it easier to iterate over the values using Java's for-each loop construct: for (VALUEIN value : values) { ... }

2.7 Hadoop Ecosystem

Although Hadoop is best known for MapReduce and its distributed filesystem (HDFS, renamed from NDFS), the term is also used for a family of related projects that fall under the umbrella of infrastructure for distributed computing and large-scale data processing.

All of the core projects covered in this book are hosted by the Apache Software Foundation, which provides support for a community of open source software projects, including the original HTTP Server from which it gets its name. As the Hadoop eco- system grows, more projects are appearing, not necessarily hosted at Apache, which provide complementary services to Hadoop, or build on the core to add higher-level abstractions.

The Hadoop projects that are covered in this book are described briefly here:

Common

A set of components and interfaces for distributed filesystems and general I/O (serialization, Java RPC, persistent data structures).

Avro

A serialization system for efficient, cross-language RPC, and persistent data storage.

MapReduce

A distributed data processing model and execution environment that runs on large clusters of commodity machines.

HDFS

A distributed filesystem that runs on large clusters of commodity machines.

Pig

A data flow language and execution environment for exploring very large datasets. Pig runs on HDFS and MapReduce clusters.

Hive

A distributed data warehouse. Hive manages data stored in HDFS and provides a query language based on SQL (and which is translated by the runtime engine to MapReduce jobs) for querying the data.

HBase

A distributed, column-oriented database. HBase uses HDFS for its underlying storage, and supports both batch-style computations using MapReduce and point queries (random reads).

ZooKeeper

A distributed, highly available coordination service. ZooKeeper provides primitives such as distributed locks that can be used for building distributed applications.

Sqoop

A tool for efficiently moving data between relational databases and HDFS.

2.8 PHYSICAL ARCHITECTURE

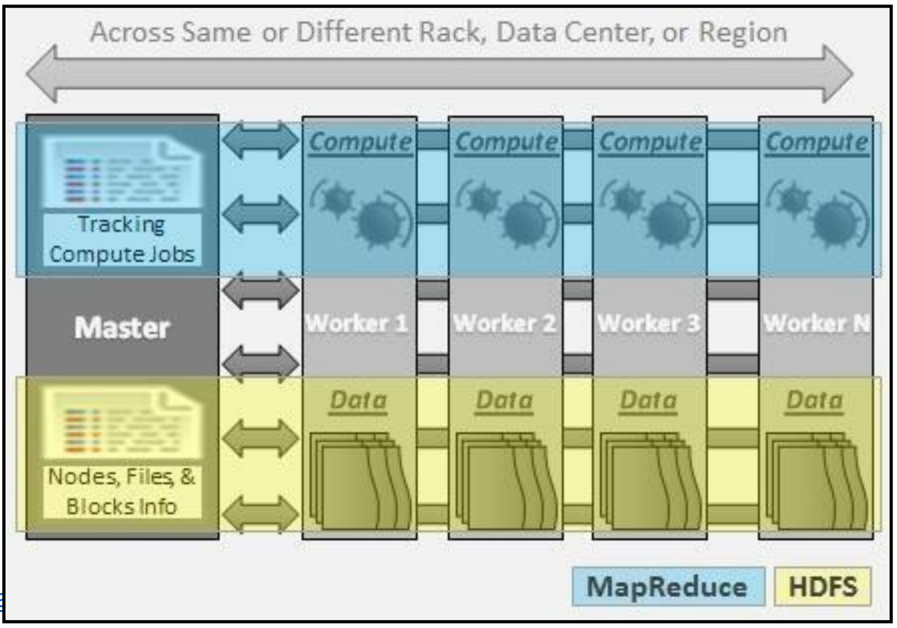

Figure 2.2: Physical Architecture

Hadoop Cluster - Architecture, Core Components and Work-flow

1. The architecture of Hadoop Cluster
2. Core Components of Hadoop Cluster
3. Work-flow of How File is Stored in Hadoop

A. Hadoop Cluster

i. Hadoop cluster is a special type of computational cluster designed for storing and analyzing vast amount of unstructured data in a distributed computing environment

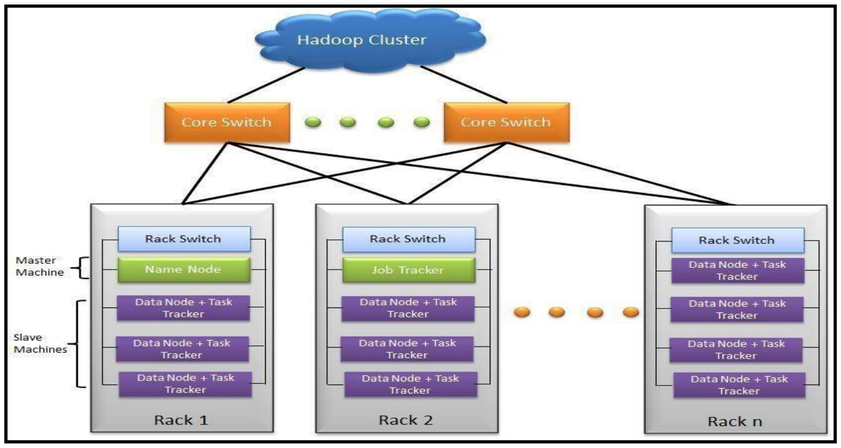

Figure 2.3: Hadoop Cluster

ii. These clusters run on low cost commodity computers.

iii. Hadoop clusters are often referred to as "shared nothing" systems because the only thing that is shared between nodes is the network that connects them.

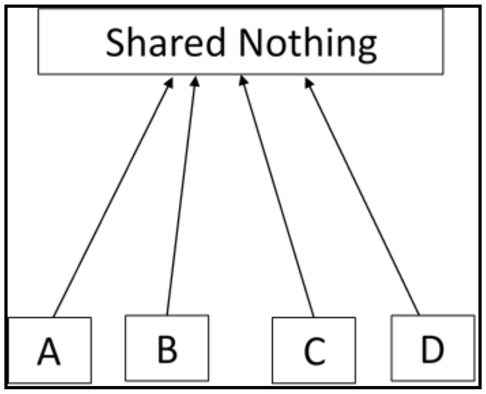

Figure 2.4: Shared Nothing

iv. Large Hadoop Clusters are arranged in several racks. Network traffic between different nodes in the same rack is much more desirable than network traffic across the racks.

A Real Time Example: Yahoo's Hadoop cluster. They have more than 10,000 machines running Hadoop and nearly 1 petabyte of user data.

Figure 2.4: Yahoo Hadoop Cluster

v. A small Hadoop cluster includes a single master node and multiple worker or slave node. As discussed earlier, the entire cluster contains two layers.

vi. One of the layer of MapReduce Layer and another is of HDFS Layer.

vii. Each of these layer have its own relevant component.

viii. The master node consists of a JobTracker, TaskTracker, NameNode and DataNode.

ix. A slave or worker node consists of a DataNode and TaskTracker.

It is also possible that slave node or worker node is only data or compute node. The matter of the fact that is the key feature of the Hadoop.

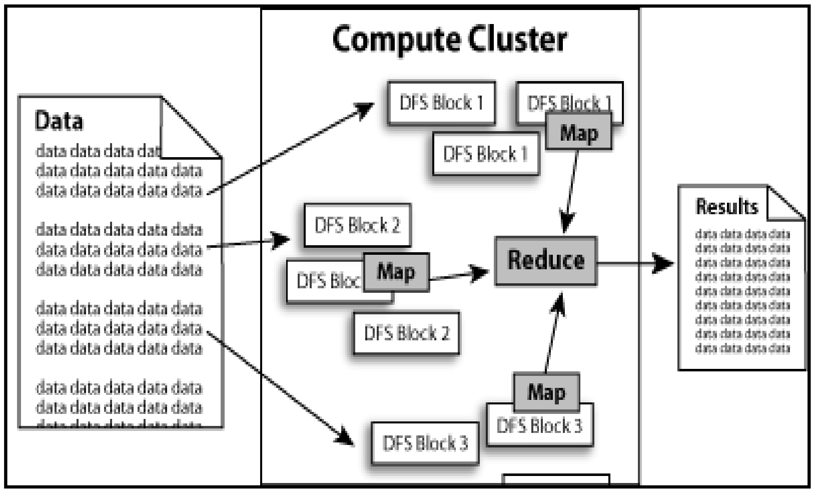

Figure 2.4: NameNode Cluster

B. HADOOP CLUSTER ARCHITECTURE:

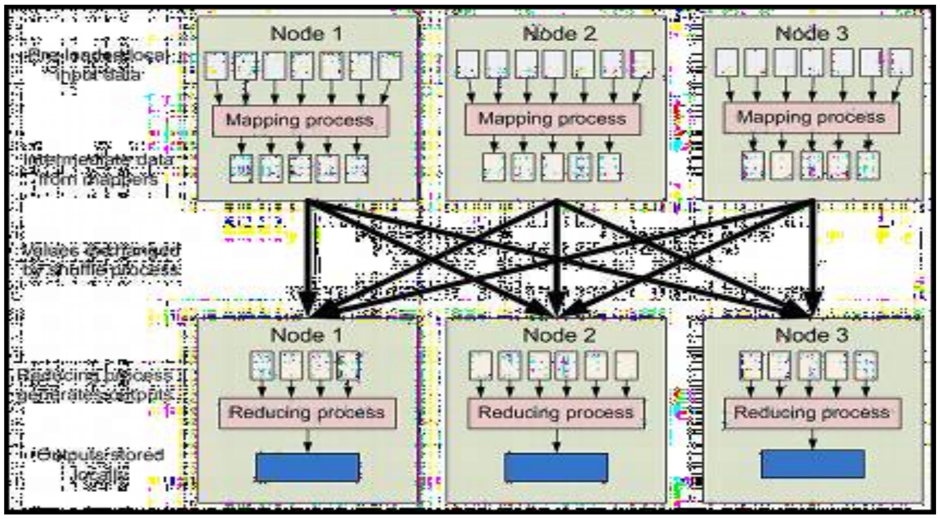

Figure 2.5: Hadoop Cluster Architecture Hadoop Cluster would consists of

- 110 different racks
- Each rack would have around 40 slave machine
- At the top of each rack there is a rack switch
- Each slave machine(rack server in a rack) has cables coming out it from both the ends
- Cables are connected to rack switch at the top which means that top rack switch will have around 80 ports
- There are global 8 core switches
- The rack switch has uplinks connected to core switches and henceconnecting all other racks with uniform bandwidth, forming the Cluster
- In the cluster, you have few machines to act as Name node and as JobTracker. They are referred as Masters. These masters have different configuration favoring more DRAM and CPU and less local storage.

Hadoop cluster has 3 components:

1. Client
2. Master
3. Slave

The role of each components are shown in the below image.

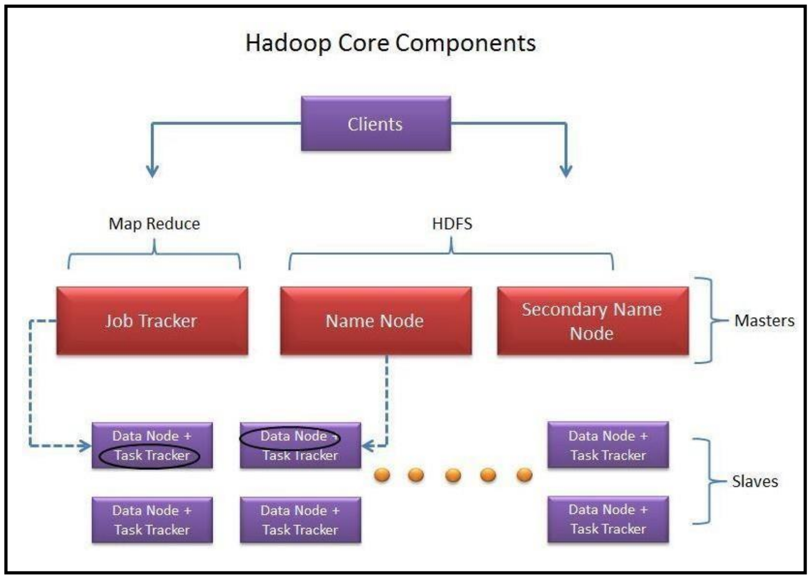

Figure 2.6: Hadoop Core Component

1. Client:

i. It is neither master nor slave, rather play a role of loading the data into cluster, submit MapReduce jobs describing how the data should be processed and then retrieve the data to see the response after job completion.

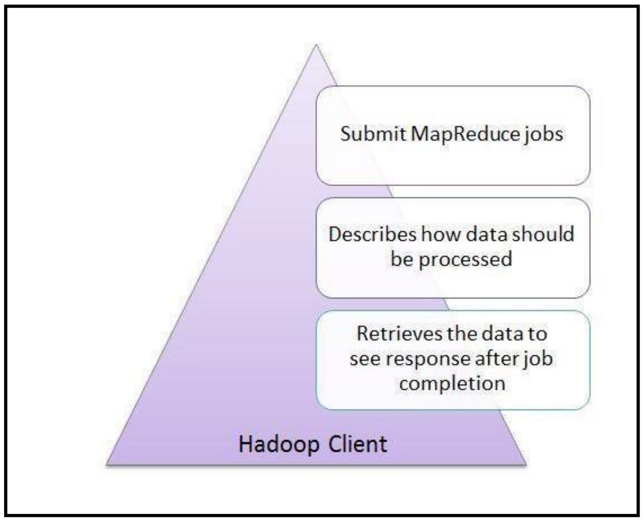
Submit MapReduce jobs
Describes how data should be processed
Retrieves the data to see response after job completion
Hadoop Client

Figure 2.6: Hadoop Client

2. Masters:

The Masters consists of 3 components NameNode, Secondary Node name and JobTracker.

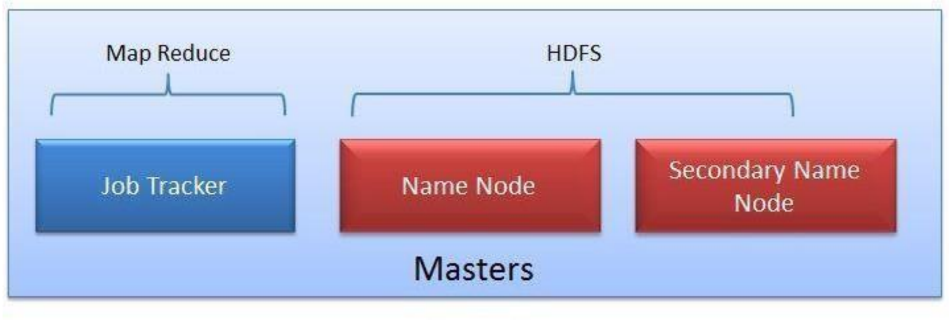

Figure 2.7: MapReduce - HDFS

i. NameNode:

➢ NameNode does NOT store the files but only the file's metadata. In later section we will see it is actually the DataNode which stores the files.

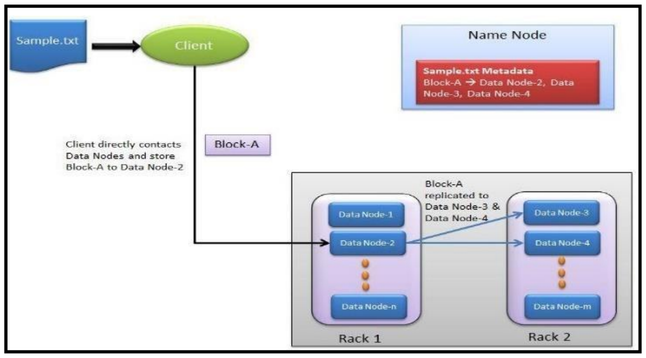
Sample.txt
Client
Name Node
Sample.txt Metadata
Block-A → Data Node-2, Data Node-3, Data Node-4
Client directly contacts Data Nodes and store Block-A to Data Node-2
Block-A
Block-A replicated to Data Node-3 & Data Node-4
Data Node-1
Data Node-2
Data Node-n
Data Node-3
Data Node-4
Data Node-m
Rack 1
Rack 2

Figure 2.8: NameNode

➢ NameNode oversees the health of DataNode and coordinates access to the data stored in DataNode.
➢ Name node keeps track of all the file system related information such as to
  - ✓ Which section of file is saved in which part of the cluster
  - ✓ Last access time for the files
  - ✓ User permissions like which user have access to the file

ii. JobTracker:

JobTracker coordinates the parallel processing of data using MapReduce.

To know more about JobTracker, please read the article All You Want to Know about MapReduce (The Heart of Hadoop)

iii. Secondary Name Node:

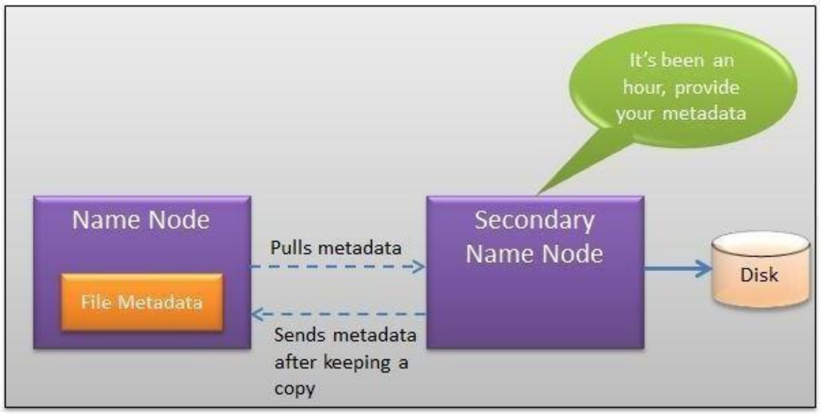
It's been an hour, provide your metadata
Name Node
File Metadata
Pulls metadata
Sends metadata after keeping a copy
Secondary Name Node
Disk

Figure 2.9: Secondary NameNode

- The job of Secondary Node is to contact NameNode in a periodic manner after certain time interval (by default 1 hour).
- NameNode which keeps all filesystem metadata in RAM has no capability to process that metadata on to disk.
- If NameNode crashes, you lose everything in RAM itself and you don't have any backup of filesystem.
- What secondary node does is it contacts NameNode in an hour and pulls copy of metadata information out of NameNode.
- It shuffle and merge this information into clean file folder and sent to back again to NameNode, while keeping a copy for itself.
- Hence Secondary Node is not the backup rather it does job of housekeeping.
- In case of NameNode failure, saved metadata can rebuild it easily.

3. Slaves:

i. Slave nodes are the majority of machines in Hadoop Cluster and are responsible to

- Store the data
- Process the computation

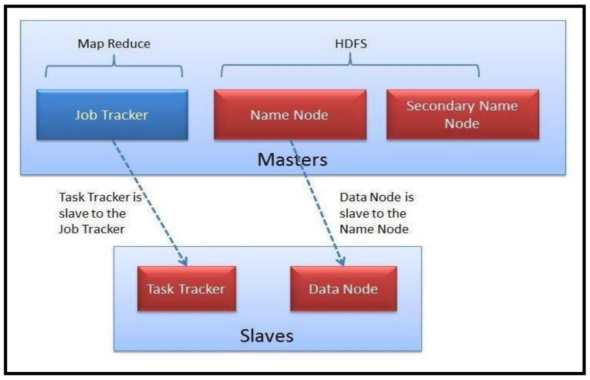

Figure 2.10: Slaves

ii. Each slave runs both a DataNode and Task Tracker daemon which communicates to their masters.

iii. The Task Tracker daemon is a slave to the Job Tracker and the DataNode daemon a slave to the NameNode

II. Hadoop- Typical Workflow in HDFS:

Take the example of input file as Sample.txt.

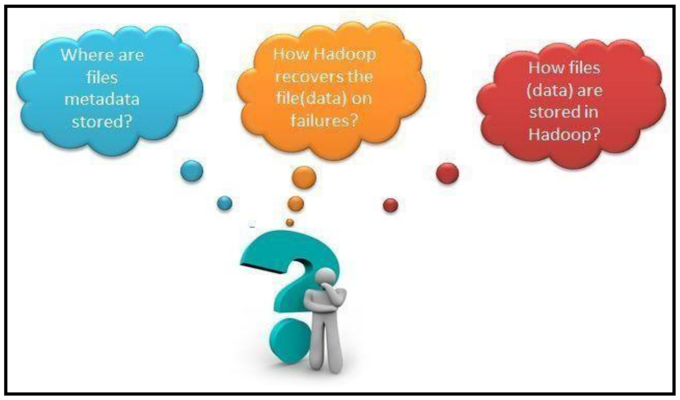
Where are files metadata stored?
How Hadoop recovers the file(data) on failures?
How files (data) are stored in Hadoop?

Figure 2.11: HDFS Workflow

1. How TestingHadoop.txt gets loaded into the Hadoop Cluster?

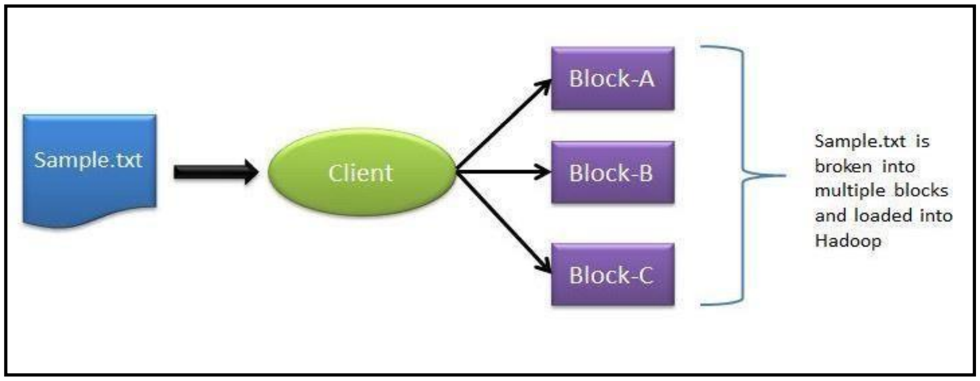
Sample.txt
Client
Block-A
Block-B
Block-C
Sample.txt is broken into multiple blocks and loaded into Hadoop

Figure 2.12: Loading file in Hadoop Cluster

- ➢ Client machine does this step and loads the Sample.txt into cluster.
- ➢ It breaks the sample.txt into smaller chunks which are known as "Blocks" in Hadoop context.
- ➢ Client put these blocks on different machines (data nodes) throughout the cluster.

2. Next, how does the Client knows that to which data nodes load the blocks?

- ➢ Now NameNode comes into picture.
- ➢ The NameNode used its Rack Awareness intelligence to decide on which DataNode to provide.

- For each of the data block (in this case Block-A, Block-B and Block-C), Client contacts NameNode and in response NameNode sends an ordered list of 3 DataNodes.

3. How does the Client knows that to which data nodes load the blocks?

- For example in response to Block-A request, Node Name may send DataNode-2, DataNode-3 and DataNode-4.
- ✓ Block-B DataNodes list DataNode-1, DataNode-3, DataNode-4 and for Block C data node list DataNode-1, DataNode-2, DataNode-3. Hence
- ❖ Block A gets stored in DataNode-2, DataNode-3, DataNode-4
- ❖ Block B gets stored in DataNode-1, DataNode-3, DataNode-4
- ❖ Block C gets stored in DataNode-1, DataNode-2, DataNode-3
- ✓ Every block is replicated to more than 1 data nodes to ensure the data recovery on the time of machine failures. That's why NameNode send 3 DataNodes list for each individual block

4. Who does the block replication?

- Client write the data block directly to one DataNode. DataNodes then replicate the block to other Data nodes.
- When one block gets written in all 3 DataNode then only cycle repeats for next block.

5. Who does the block replication?

- In Hadoop Gen 1 there is only one NameNode wherein Gen2 there is active passive model in NameNode where one more node "Passive Node" comes in picture.
- The default setting for Hadoop is to have 3 copies of each block in the cluster. This setting can be configured with "dfs.replication" parameter of hdfs-site.xml file.
- Keep note that Client directly writes the block to the DataNode without any intervention of NameNode in this process.

2.9 Hadoop limitations

i. Network File system is the oldest and the most commonly used distributed file system and was designed for the general class of applications, Hadoop only specific kind of applications can make use of it.

ii. It is known that Hadoop has been created to address the limitations of the distributed file system, where it can store the large amount of data, offers failure protection and provides fast access, but it should be known that the benefits that come with Hadoop come at some cost.

iii. Hadoop is designed for applications that require random reads; so if a file has four parts the file would like to read all the parts one-by-one going from 1 to 4 till the end. Random seek is where you want to go to a specific location in the file; this issomething that isn’t possible with Hadoop. Hence, Hadoop is designed for non- real-time batch processing of data.

iv. Hadoop is designed for streaming reads caching of data isn’t provided. Caching of data is provided which means that when you want to read data another time, it can be read very fast from the cache. This caching isn’t possible because you get faster access to the data directly by doing the sequential read; hence caching isn’t available through Hadoop.

v. It will write the data and then it will read the data several times. It will not be updating the data that it has written; hence updating data written to closed files is not available. However, you have to know that in update 0.19 appending will be supported for those files that aren’t closed.

But for those files that have been closed, updating isn't possible.

vi. In case of Hadoop we aren't talking about one computer; in this scenario we usually have a large number of computers and hardware failures are unavoidable; sometime one computer will fail and sometimes the entire rack can fail too. Hadoop gives excellent protection against hardware failure; however the performance will go down proportionate to the number of computers that are down. In the big picture, it doesn't really matter and it is not generally noticeable since if you have 100 computers and in them if 3 fail then 97 are still working. So the proportionate loss of performance isn't that noticeable. However, the way Hadoop works there is the loss in performance. Now this loss of performance through hardware failures is something that is managed through replication strategy.