MODULE 2

5.1 INTRODUCING HADOOP

Today, Big Data seems to be the buzz word! Enterprises, the world over, are beginning to realize that there is a huge volume of untapped information before them in the form of structured, semi-structured, and unstructured data. This varied variety of data is spread across the networks.

Let us look at few statistics to get an idea of the amount of data which gets generated every day, every minute, and every second.

1. **Every day:**
   (a) NYSE (New York Stock Exchange) generates 1.5 billion shares and trade data.
   (b) Facebook stores 2.7 billion comments and Likes.
   (c) Google processes about 24 petabytes of data.
2. **Every minute:**
   (a) Facebook users share nearly 2.5 million pieces of content.
   (b) Twitter users tweet nearly 300,000 times.
   (c) Instagram users post nearly 220,000 new photos.
   (d) YouTube users upload 72 hours of new video content.
   (e) Apple users download nearly 50,000 apps.
   (f) Email users send over 200 million messages.
   (g) Amazon generates over $80,000 in online sales.
   (h) Google receives over 4 million search queries.
3. **Every second:**
   (a) Banking applications process more than 10,000 credit card transactions.

5.1.1 Data: The Treasure Trove

1. Provides business advantages such as generating product recommendations, inventing new products, analyzing the market, and many, many more, ….
2. Provides few early key indicators that can turn the fortune of business.
3. Provides room for precise analysis. If we have more data for analysis, then we have greater precision of analysis.

To process, analyze, and make sense of these different kinds of data, we need a system that scales and addresses the challenges shown in Figure 5.1.

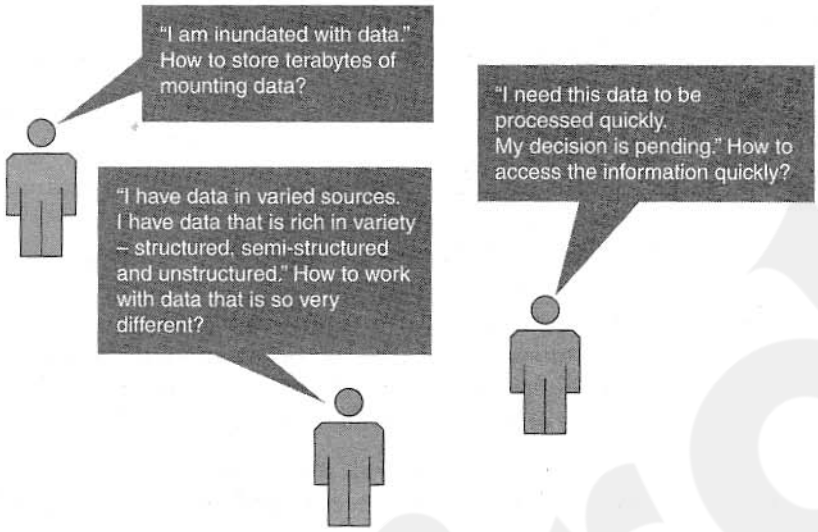


Figure 5.1 Challenges with big volume, variety, and velocity of data.

5.2 WHY HADOOP?

Ever wondered why Hadoop has been and is one of the most wanted technologies!!

The key consideration (the rationale behind its huge popularity) is:

Its capability to handle massive amounts of data, different categories of data – fairly quickly.

The other considerations are (Figure 5.2):

1. **Low cost:** Hadoop is an open-source framework and uses commodity hardware (commodity hardware is relatively inexpensive and easy to obtain hardware) to store enormous quantities of data.

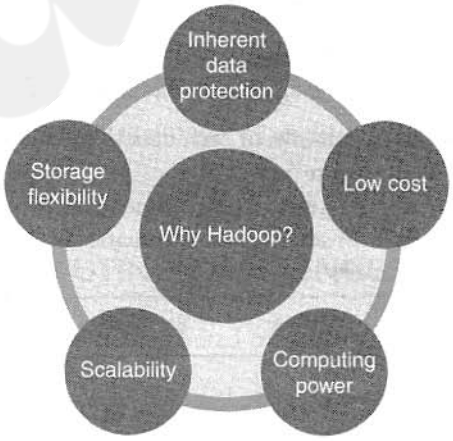


Figure 5.2 Key considerations of Hadoop.

2. **Computing power:** Hadoop is based on distributed computing model which processes very large volumes of data fairly quickly. The more the number of computing nodes, the more the processing power at hand.
3. **Scalability:** This boils down to simply adding nodes as the system grows and requires much less administration.
4. **Storage flexibility:** Unlike the traditional relational databases, in Hadoop data need not be pre-processed before storing it. Hadoop provides the convenience of storing as much data as one needs and also the added flexibility of deciding later as to how to use the stored data. In Hadoop, one can store unstructured data like images, videos, and free-form text.
5. **Inherent data protection:** Hadoop protects data and executing applications against hardware failure. If a node fails, it automatically redirects the jobs that had been assigned to this node to the other functional and available nodes and ensures that distributed computing does not fail. It goes a step further to store multiple copies (replicas) of the data on various nodes across the cluster.

Hadoop makes use of commodity hardware, distributed file system, and distributed computing as shown in Figure 5.3. In this new design, groups of machine are gathered together; it is known as a **Cluster**.

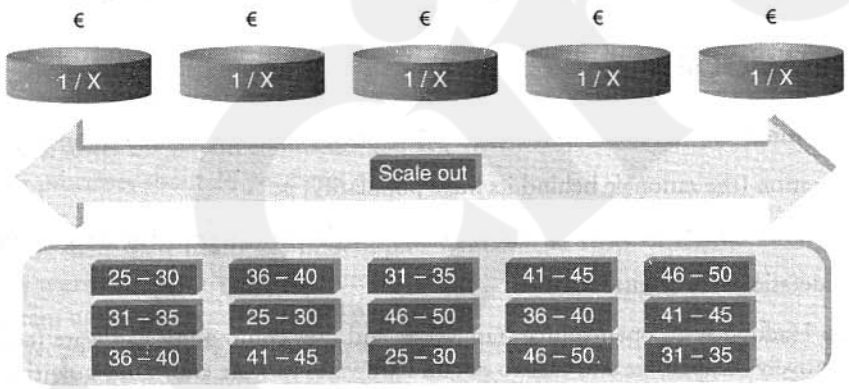


Figure 5.3 Hadoop framework (distributed file system, commodity hardware).

With this new paradigm, the data can be managed with **Hadoop** as follows:

1. Distributes the data and duplicates chunks of each data file across several nodes, for example, 25–30 is one chunk of data as shown in Figure 5.3.
2. Locally available compute resource is used to process each chunk of data in parallel.
3. Hadoop Framework handles failover smartly and automatically.

5.3 WHY NOT RDBMS?

RDBMS is not suitable for storing and processing large files, images, and videos. RDBMS is not a good choice when it comes to advanced analytics involving machine learning. Figure 5.4 describes the RDBMS system with respect to cost and storage. It calls for huge investment as the volume of data shows an upward trend.

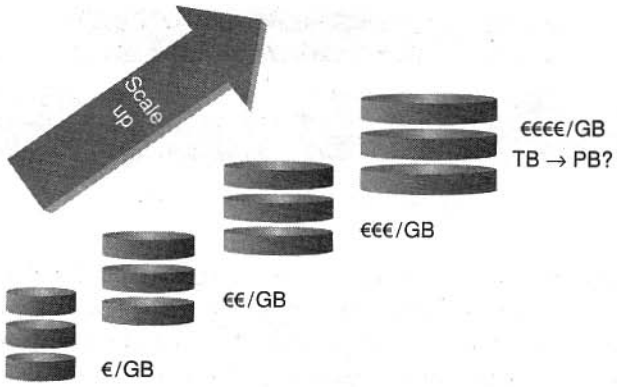


Figure 5.4 RDBMS with respect to cost/GB of storage.

5.4 RDBMS versus HADOOP

Table 5.1 describes the difference between RDBMS and Hadoop.

Table 5.1 RDBMS versus Hadoop

PARAMETERS	RDBMS	HADOOP
System	Relational Database Management System.	Node Based Flat Structure.
Data	Suitable for structured data.	Suitable for structured, unstructured data. Supports variety of data formats in real time such as XML, JSON, text based flat file formats, etc.
Processing	OLTP	Analytical, Big Data Processing
Choice	When the data needs consistent relationship.	Big Data processing, which does not require any consistent relationships between data.
Processor	Needs expensive hardware or high-end processors to store huge volumes of data.	In a Hadoop Cluster, a node requires only a processor, a network card, and few hard drives.
Cost	Cost around $10,000 to $14,000 per terabytes of storage.	Cost around $4,000 per terabytes of storage.

5.5 DISTRIBUTED COMPUTING CHALLENGES

Although there are several challenges with distributed computing, we will focus on two major challenges.

5.5.1 Hardware Failure

In a distributed system, several servers are networked together. This implies that more often than not, there may be a possibility of hardware failure. And when such a failure does happen, how does one retrieve the

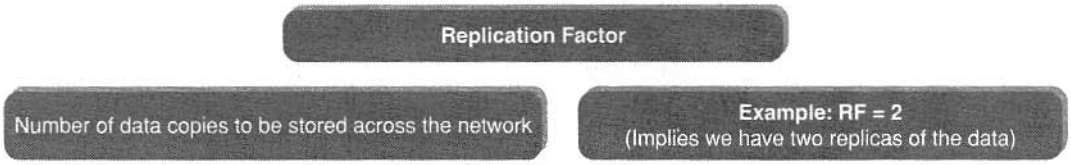


Figure 5.5 Replication factor.

data that was stored in the system? Just to explain further – a regular hard disk may fail once in 3 years. And when you have 1000 such hard disks, there is a possibility of at least a few being down every day.

Hadoop has an answer to this problem in **Replication Factor (RF)**. **Replication Factor** connotes the number of data copies of a given data item/data block stored across the network. Refer Figure 5.5.

JUST TO UNDERSTAND REPLICATION FURTHER, PICTURE THIS...

You work in a project team. There are six other members in the team. Each time there is an update related to the project work or an input received from the client, the project manager, Alex, ensures that he keeps at least three team members aware of the developments. You have been wondering at this style of working of your project manager. One day during the coffee break, when the project manager joins for coffee, you hesitantly ask him the question. Alex, "I had this question for you. Why is that each time we have an input from the client or any important piece of information, you leave it with at least three of our team members?" Alex smiled as he answered, "The reason is very simple. Assume that the client called and suggested some modification to the project. I shared it with just one person, let us say, person X. Tomorrow, when the suggested changes have to be incorporated, person X calls in sick. He is indisposed and not in office. Will that lead to our project coming to a standstill? Yes, isn't it? Therefore I share it with at least three team members, so that even if one is on leave or out of office for some reason, our work will not be stalled."

5.5.2 How to Process This Gigantic Store of Data?

In a distributed system, the data is spread across the network on several machines. A key challenge here is to integrate the data available on several machines prior to processing it.

Hadoop solves this problem by using **MapReduce** Programming. It is a programming model to process the data (MapReduce programming will be discussed a little later).

5.6 HISTORY OF HADOOP

Hadoop was created by Doug Cutting, the creator of Apache Lucene (a commonly used text search library). Hadoop is a part of the Apache Nutch (Yahoo) project (an open-source web search engine) and also a part of the Lucene project. Refer Figure 5.6 for more details.

5.6.1 The Name "Hadoop"

The name Hadoop is not an acronym; it's a made-up name. The project creator, Doug Cutting, explains how the name came about: "*The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such. Googol is a kid's term*".

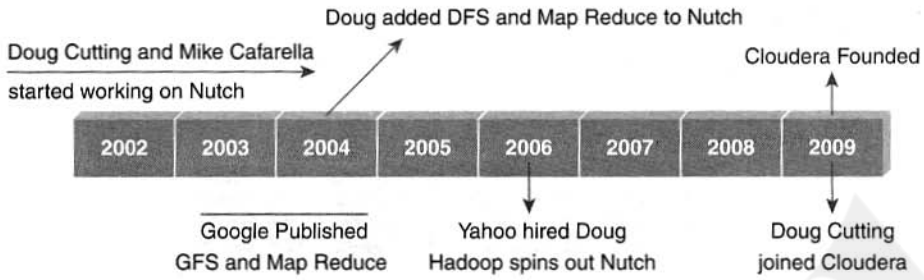


Figure 5.6 Hadoop history.

Subprojects and "contrib" modules in Hadoop also tend to have names that are unrelated to their function, often with an elephant or other animal theme ("Pig", for example).

Reference: Hadoop, The Definitive Guide, 3rd Edition, O'Reilly Publication Page. No. 9.

5.7 HADOOP OVERVIEW

Open-source software framework to store and process massive amounts of data in a distributed fashion on large clusters of commodity hardware. Basically, Hadoop accomplishes two tasks:

1. Massive data storage.
2. Faster data processing.

5.7.1 Key Aspects of Hadoop

Figure 5.7 describes the key aspects of Hadoop.

Figure 5.7 Key aspects of Hadoop.

5.7.2 Hadoop Components

Figure 5.8 depicts the Hadoop components.

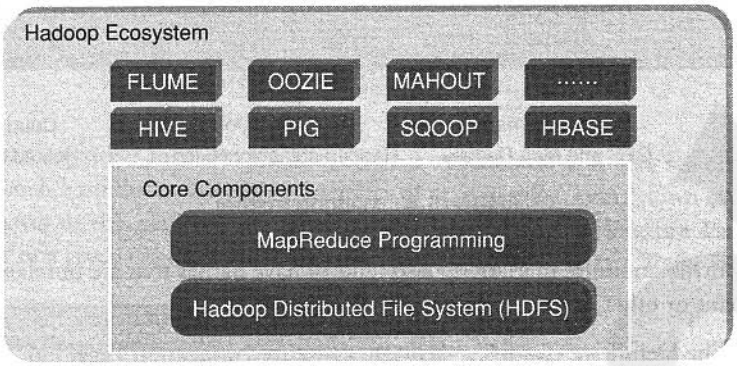


Figure 5.8 Hadoop components.

Hadoop Core Components

1. **HDFS:**
   (a) Storage component.
   (b) Distributes data across several nodes.
   (c) Natively redundant.
2. **MapReduce:**
   (a) Computational framework.
   (b) Splits a task across multiple nodes.
   (c) Processes data in parallel.

Hadoop Ecosystem: Hadoop Ecosystem are support projects to enhance the functionality of Hadoop Core Components. The Eco Projects are as follows:

1. HIVE
2. PIG
3. SQOOP
4. HBASE
5. FLUME
6. OOZIE
7. MAHOUT

5.7.3 Hadoop Conceptual Layer

It is conceptually divided into **Data Storage Layer** which stores huge volumes of data and **Data Processing Layer** which processes data in parallel to extract richer and meaningful insights from data (Figure 5.9).

5.7.4 High-Level Architecture of Hadoop

Hadoop is a distributed **Master-Slave** Architecture. Master node is known as **NameNode** and slave nodes are known as **DataNodes.** Figure 5.10 depicts the Master–Slave Architecture of Hadoop Framework.

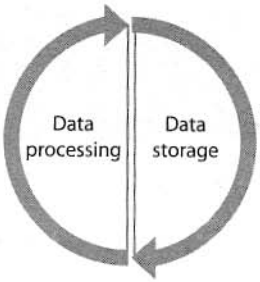


Figure 5.9 Hadoop conceptual layer.

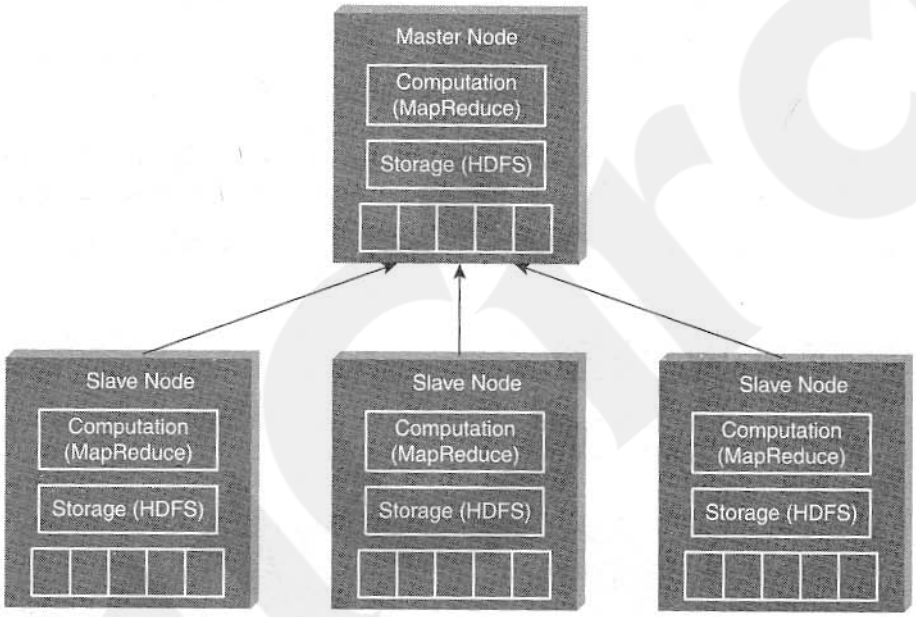


Figure 5.10 Hadoop high-level architecture.
Reference: Hadoop in Practice, Alex Holmes.

Let us look at the key components of the Master Node.

1. **Master HDFS:** Its main responsibility is partitioning the data storage across the slave nodes. It also keeps track of locations of data on DataNodes.
2. **Master MapReduce:** It decides and schedules computation task on slave nodes.

5.8 USE CASE OF HADOOP

5.8.1 ClickStream Data

ClickStream data (mouse clicks) helps you to understand the purchasing behavior of customers. ClickStream analysis helps online marketers to optimize their product web pages, promotional content, etc. to improve their business.

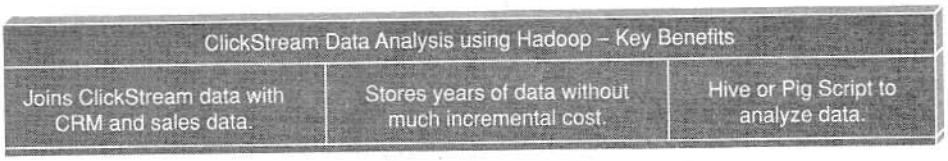


Figure 5.11 ClickStream data analysis.

The ClickStream analysis (Figure 5.11) using Hadoop provides **three key benefits**:

1. Hadoop helps to join ClickStream data with other data sources such as Customer Relationship Management Data (Customer Demographics Data, Sales Data, and Information on Advertising Campaigns). This additional data often provides the much needed information to understand customer behavior.
2. Hadoop's scalability property helps you to store years of data without ample incremental cost. This helps you to perform temporal or year over year analysis on ClickStream data which your competitors may miss.
3. Business analysts can use **Apache Pig** or **Apache Hive** for website analysis. With these tools, you can organize ClickStream data by user session, refine it, and feed it to visualization or analytics tools.

Reference: http://hortonworks.com/wp-content/uploads/2014/05/Hortonworks.BusinessValueofHadoop.v1.0.pdf

5.9 HADOOP DISTRIBUTORS

The companies shown in Figure 5.12 provide products that include Apache Hadoop, commercial support, and/or tools and utilities related to Hadoop.

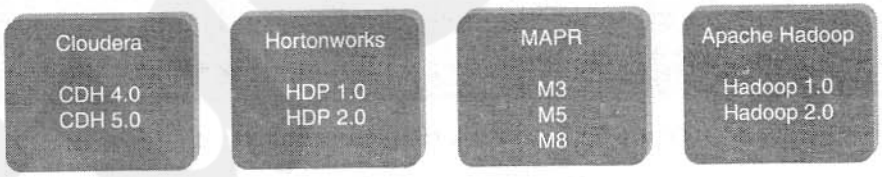


Figure 5.12 Common Hadoop distributors.

5.10 HDFS (HADOOP DISTRIBUTED FILE SYSTEM)

Some key Points of Hadoop Distributed File System are as follows:

1. Storage component of Hadoop.
2. Distributed File System.
3. Modeled after Google File System.
4. Optimized for high throughput (HDFS leverages large block size and moves computation where data is stored).
5. You can replicate a file for a configured number of times, which is tolerant in terms of both software and hardware.

6. Re-replicates data blocks automatically on nodes that have failed.
7. You can realize the power of HDFS when you perform read or write on large files (gigabytes and larger).
8. Sits on top of native file system such as ext3 and ext4, which is described in Figure 5.13.

Figure 5.14 describes important key points of HDFS. Figure 5.15 describes Hadoop Distributed File System Architecture. Client Application interacts with NameNode for metadata related activities and communicates with DataNodes to read and write files. DataNodes converse with each other for pipeline reads and writes.

Let us assume that the file "Sample.txt" is of size **192 MB**. As per the default data block size (64 MB), it will be split into three blocks and replicated across the nodes on the cluster based on the default replication factor.

5.10.1 HDFS Daemons

5.10.1.1 *NameNode*

HDFS breaks a large file into smaller pieces called **blocks**. NameNode uses a **rack ID** to identify DataNodes in the rack. A rack is a collection of DataNodes within the cluster. NameNode keeps tracks of blocks of a file as it is placed on various DataNodes. NameNode manages file-related operations such as read, write, create, and delete. Its main job is managing the **File System Namespace**. A file system namespace is collection of files in the cluster. NameNode stores HDFS namespace. File system namespace includes mapping of blocks to file, file properties and is stored in a file called **FsImage**. NameNode uses an EditLog (transaction log) to record every transaction that happens to the file system metadata. Refer Figure 5.16. When NameNode starts up, it reads FsImage and EditLog from disk and applies all transactions from the EditLog to in-memory representation of the FsImage. Then it flushes out new version of FsImage on disk and truncates the old EditLog because the changes are updated in the FsImage. There is a single NameNode per cluster.

Reference: http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html

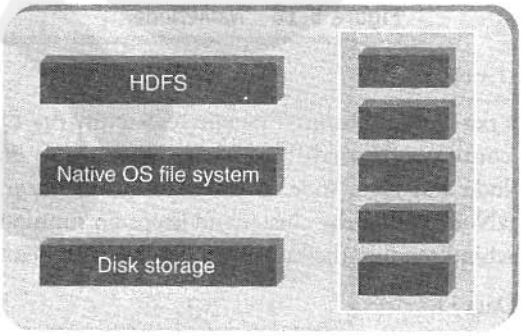


Figure 5.13 Hadoop Distributed File System.

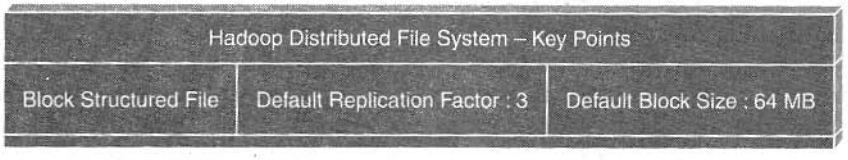


Figure 5.14 Hadoop Distributed File System – key points.

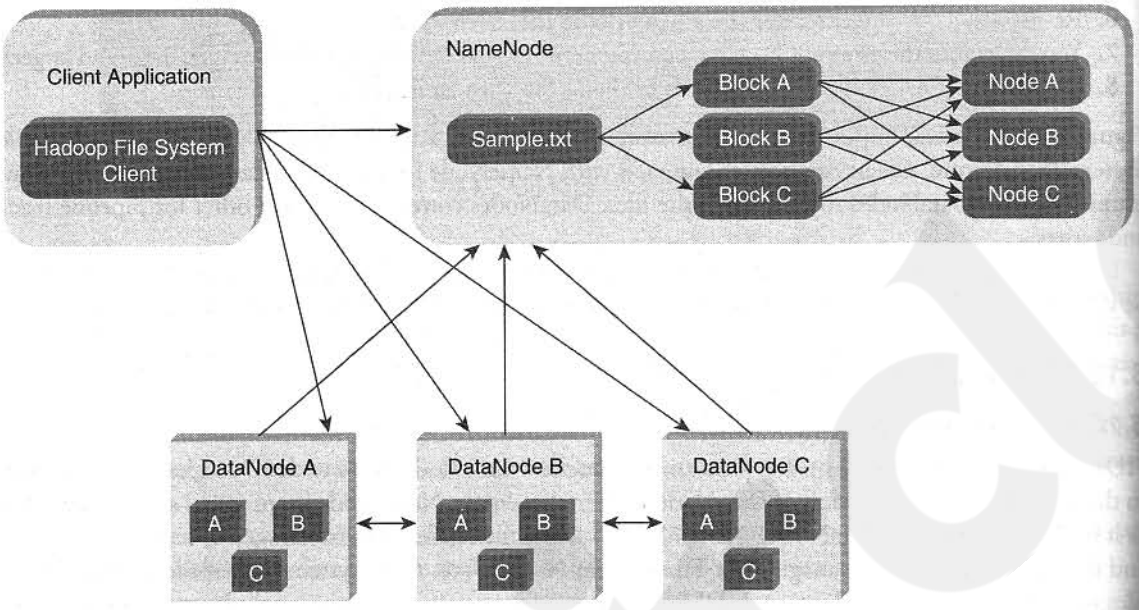


Figure 5.15 Hadoop Distributed File System Architecture.
Reference: Hadoop in Practice, Alex Holmes.

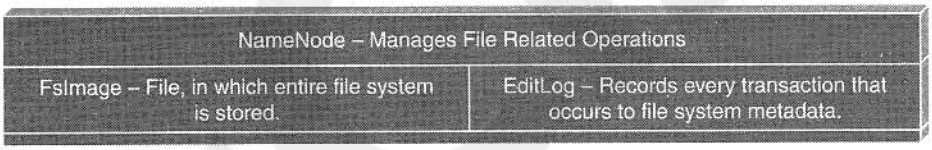


Figure 5.16 NameNode.

5.10.1.2 DataNode

There are multiple DataNodes per cluster. During Pipeline read and write DataNodes communicate with each other. A DataNode also continuously sends **"heartbeat"** message to NameNode to ensure the connectivity between the NameNode and DataNode. In case there is no heartbeat from a DataNode, the NameNode replicates that DataNode within the cluster and keeps on running as if nothing had happened.

Let us explain the concept behind sending the heartbeat report by the DataNodes to the NameNode.

Reference: Wrox Certified Big Data Developer.

PICTURE THIS...

You work for a renowned IT organization. Every day when you come to office, you are required to swipe in to record your attendance. This record of attendance is then shared with your manager to keep him posted on who all from his team have reported for work. Your manager is able to allocate tasks to the team members who are present in office. The tasks for the day cannot be allocated to team members who have not turned in. Likewise heartbeat report is a way by which DataNodes inform the NameNode that they are up and functional and can be assigned tasks. Figure 5.17 depicts the above scenario.

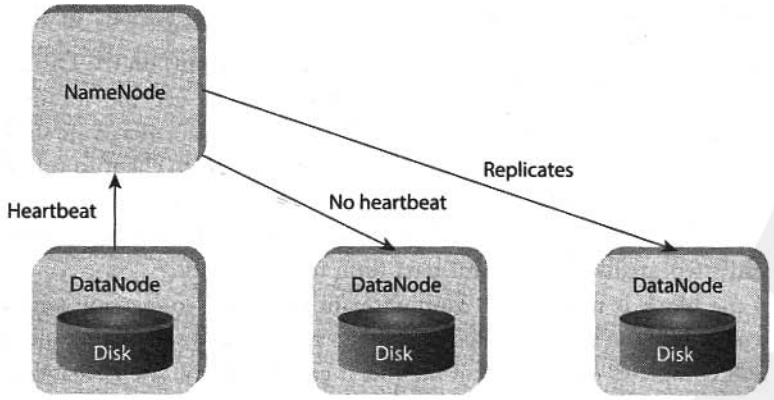


Figure 5.17 NameNode and DataNode Communication.

5.10.1.3 *Secondary NameNode*

The Secondary NameNode takes a snapshot of HDFS metadata at intervals specified in the Hadoop configuration. Since the memory requirements of Secondary NameNode are the same as NameNode, it is better to run NameNode and Secondary NameNode on different machines. In case of failure of the NameNode, the Secondary NameNode can be configured manually to bring up the cluster. However, the Secondary NameNode does not record any real-time changes that happen to the HDFS metadata.

5.10.2 Anatomy of File Read

Figure 5.18 describes the anatomy of File Read.

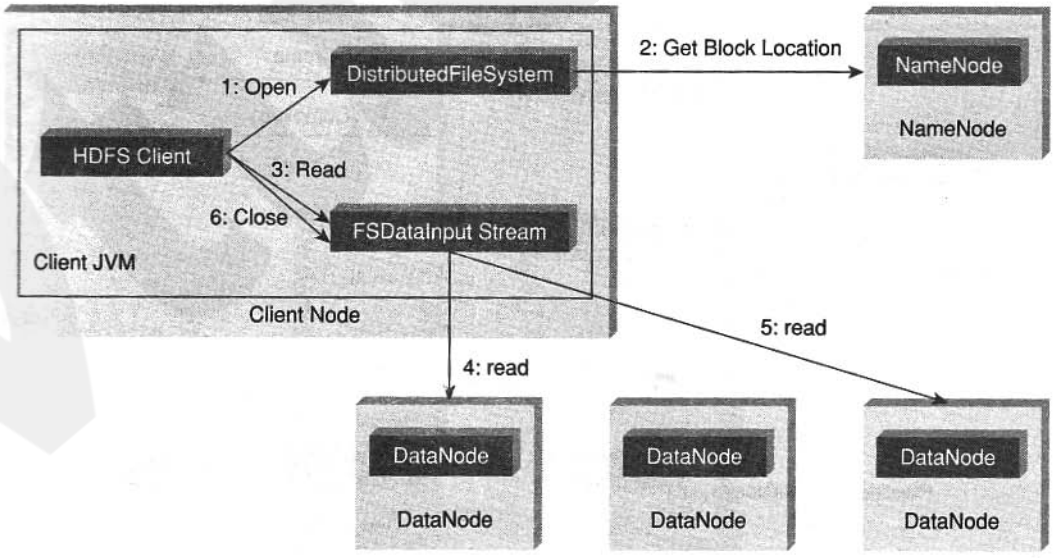


Figure 5.18 File Read.

The steps involved in the File Read are as follows:

1. The client opens the file that it wishes to read from by calling open() on the DistributedFileSystem.
2. DistributedFileSystem communicates with the NameNode to get the location of data blocks. NameNode returns with the addresses of the DataNodes that the data blocks are stored on. Subsequent to this, the DistributedFileSystem returns an FSDataInputStream to client to read from the file.
3. Client then calls read() on the stream DFSInputStream, which has addresses of the DataNodes for the first few blocks of the file, connects to the closest DataNode for the first block in the file.
4. Client calls read() repeatedly to stream the data from the DataNode.
5. When end of the block is reached, DFSInputStream closes the connection with the DataNode. It repeats the steps to find the best DataNode for the next block and subsequent blocks.
6. When the client completes the reading of the file, it calls close() on the FSDataInputStream to close the connection.

Reference: Hadoop, The Definitive Guide, 3rd Edition, O'Reilly Publication.

5.10.3 Anatomy of File Write

Figure 5.19 describes the anatomy of File Write. The steps involved in anatomy of File Write are as follows:

1. The client calls create() on DistributedFileSystem to create a file.
2. An RPC call to the NameNode happens through the DistributedFileSystem to create a new file. The NameNode performs various checks to create a new file (checks whether such a file exists or not). Initially, the NameNode creates a file without associating any data blocks to the file. The DistributedFileSystem returns an FSDataOutputStream to the client to perform write.
3. As the client writes data, data is split into packets by DFSOutputStream, which is then written to an internal queue, called *data queue*. DataStreamer consumes the data queue. The DataStreamer requests

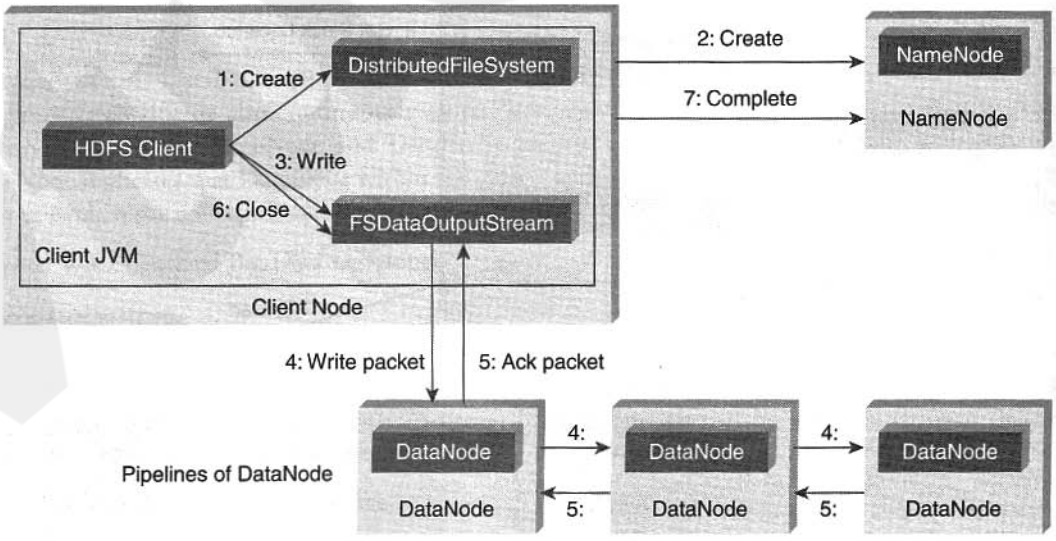


Figure 5.19 File Write.

the NameNode to allocate new blocks by selecting a list of suitable DataNodes to store replicas. This list of DataNodes makes a pipeline. Here, we will go with the default replication factor of three, so there will be three nodes in the pipeline for the first block.

4. DataStreamer streams the packets to the first DataNode in the pipeline. It stores packet and forwards it to the second DataNode in the pipeline. In the same way, the second DataNode stores the packet and forwards it to the third DataNode in the pipeline.
5. In addition to the internal queue, DFSOutputStream also manages an "Ack queue" of packets that are waiting for the acknowledgement by DataNodes. A packet is removed from the "Ack queue" only if it is acknowledged by all the DataNodes in the pipeline.
6. When the client finishes writing the file, it calls close() on the stream.
7. This flushes all the remaining packets to the DataNode pipeline and waits for relevant acknowledgments before communicating with the NameNode to inform the client that the creation of the file is complete.

Reference: Hadoop, The Definitive Guide, 3rd Edition, O'Reilly Publication.

5.10.4 Replica Placement Strategy

5.10.4.1 *Hadoop Default Replica Placement Strategy*

As per the Hadoop Replica Placement Strategy, first replica is placed on the same node as the client. Then it places second replica on a node that is present on different rack. It places the third replica on the same rack as second, but on a different node in the rack. Once replica locations have been set, a pipeline is built. This strategy provides good reliability. Figure 5.20 describes the typical replica pipeline.

Reference: Hadoop, the Definite Guide, 3rd Edition, O'Reilly Publication.

5.10.5 Working with HDFS Commands

Objective: To get the list of directories and files at the root of HDFS.
Act:

hadoop fs -ls /

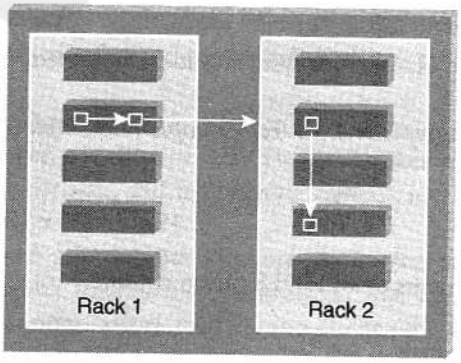


Figure 5.20 Replica Placement Strategy.

Objective: To get the list of complete directories and files of HDFS.
Act:
hadoop fs -ls -R /

Objective: To create a directory (say, sample) in HDFS.
Act:
hadoop fs -mkdir /sample

Objective: To copy a file from local file system to HDFS.
Act:
hadoop fs -put /root/sample/test.txt /sample/test.txt

Objective: To copy a file from HDFS to local file system.
Act:
hadoop fs -get /sample/test.txt /root/sample/testsample.txt

Objective: To copy a file from local file system to HDFS via copyFromLocal command.
Act:
hadoop fs -copyFromLocal /root/sample/test.txt /sample/testsample.txt

Objective: To copy a file from Hadoop file system to local file system via copyToLocal command.
Act:
hadoop fs -copyToLocal /sample/test.txt /root/sample/testsample1.txt

Objective: To display the contents of an HDFS file on console.
Act:
hadoop fs -cat /sample/test.txt

Objective: To copy a file from one directory to another on HDFS.
Act:
hadoop fs -cp /sample/test.txt /sample1

Objective: To remove a directory from HDFS.
Act:
hadoop fs-rm-r /sample1

5.10.6 Special Features of HDFS

1. **Data Replication:** There is absolutely no need for a client application to track all blocks. It directs the client to the nearest replica to ensure high performance.
2. **Data Pipeline:** A client application writes a block to the first DataNode in the pipeline. Then this DataNode takes over and forwards the data to the next node in the pipeline. This process continues for all the data blocks, and subsequently all the replicas are written to the disk.

Reference: Wrox Certified Big Data Developer.

5.11 PROCESSING DATA WITH HADOOP

MapReduce Programming is a software framework. MapReduce Programming helps you to process massive amounts of data in parallel.

In MapReduce Programming, the input dataset is split into independent chunks. **Map tasks** process these independent chunks completely in a parallel manner. The output produced by the map tasks serves as intermediate data and is stored on the local disk of that server. The output of the mappers are automatically shuffled and sorted by the framework. MapReduce Framework sorts the output based on **keys**. This sorted output becomes the input to the **reduce tasks**. Reduce task provides reduced output by combining the output of the various mappers. Job inputs and outputs are stored in a file system. MapReduce framework also takes care of the other tasks such as scheduling, monitoring, re-executing failed tasks, etc.

Hadoop Distributed File System and MapReduce Framework run on the same set of nodes. This configuration allows effective scheduling of tasks on the nodes where data is present **(Data Locality)**. This in turn results in very high throughput.

There are two daemons associated with MapReduce Programming. A single master **JobTracker** per cluster and one slave **TaskTracker** per cluster-node. The JobTracker is responsible for scheduling tasks to the TaskTrackers, monitoring the task, and re-executing the task just in case the TaskTracker fails. The TaskTracker executes the task. Refer Figure 5.21.

The MapReduce functions and input/output locations are implemented via the MapReduce applications. These applications use suitable interfaces to construct the job. The application and the job parameters together are known as **job configuration**. Hadoop **job client** submits job (jar/executable, etc.) to the JobTracker. Then it is the responsibility of JobTracker to schedule tasks to the slaves. In addition to scheduling, it also monitors the task and provides status information to the job-client.

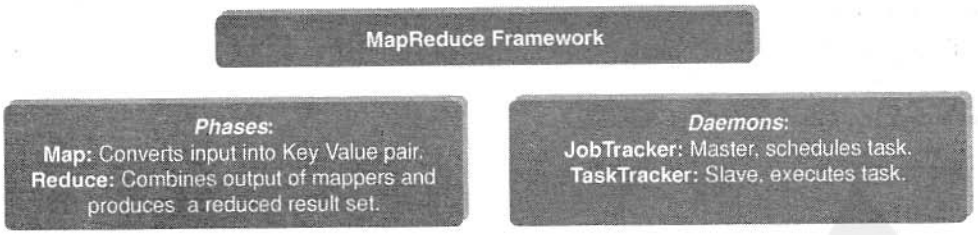


Figure 5.21 MapReduce Programming phases and daemons.

Reference: http://hadoop.apache.org/docs/r1.0.4/mapred_tutorial.html

5.11.1 MapReduce Daemons

1. **JobTracker:** It provides connectivity between Hadoop and your application. When you submit code to cluster, JobTracker creates the execution plan by deciding which task to assign to which node. It also monitors all the running tasks. When a task fails, it automatically re-schedules the task to a different node after a predefined number of retries. JobTracker is a master daemon responsible for executing overall MapReduce job. There is a single JobTracker per Hadoop cluster.
2. **TaskTracker:** This daemon is responsible for executing individual tasks that is assigned by the JobTracker. There is a single TaskTracker per slave and spawns multiple Java Virtual Machines (JVMs) to handle multiple map or reduce tasks in parallel. TaskTracker continuously sends heartbeat message to JobTracker. When the JobTracker fails to receive a heartbeat from a TaskTracker, the JobTracker assumes that the TaskTracker has failed and resubmits the task to another available node in the cluster. Once the client submits a job to the JobTracker, it partitions and assigns diverse MapReduce tasks for each TaskTracker in the cluster. Figure 5.22 depicts JobTracker and TaskTracker interaction.

Reference: Hadoop in Action, Chuck Lam.

5.11.2 How Does MapReduce Work?

MapReduce divides a data analysis task into two parts – **map** and **reduce**. Figure 5.23 depicts how the MapReduce Programming works. In this example, there are two mappers and one reducer. Each mapper

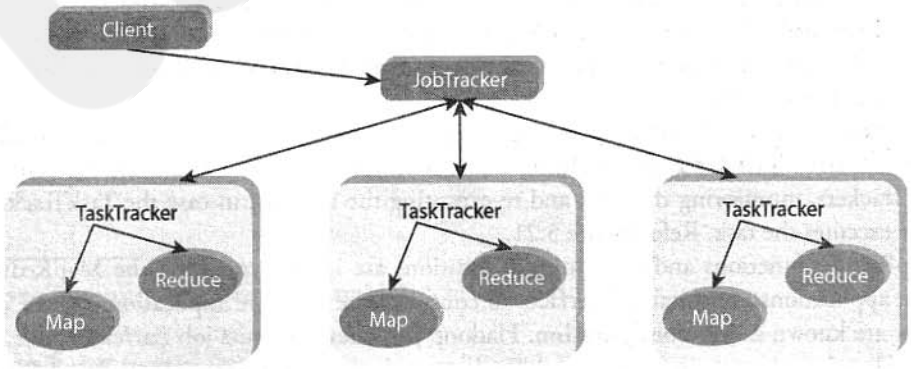


Figure 5.22 JobTracker and TaskTracker interaction.

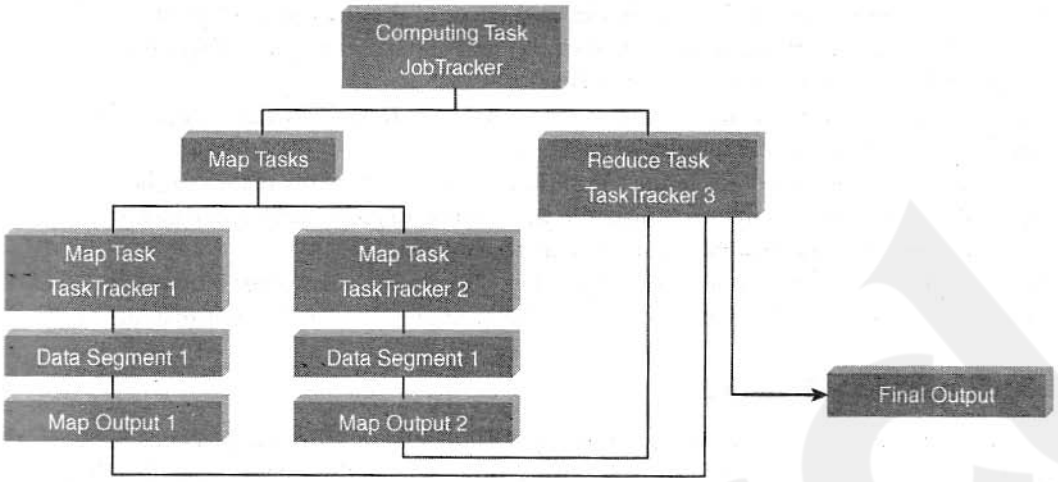


Figure 5.23 MapReduce programming workflow.

works on the partial dataset that is stored on that node and the reducer combines the output from the mappers to produce the reduced result set.

Reference: Wrox Big Data Certification Materials.

Figure 5.24 describes the working model of MapReduce Programming. The following steps describe how MapReduce performs its task.

1. First, the input dataset is split into multiple pieces of data (several small subsets).
2. Next, the framework creates a master and several workers processes and executes the worker processes remotely.

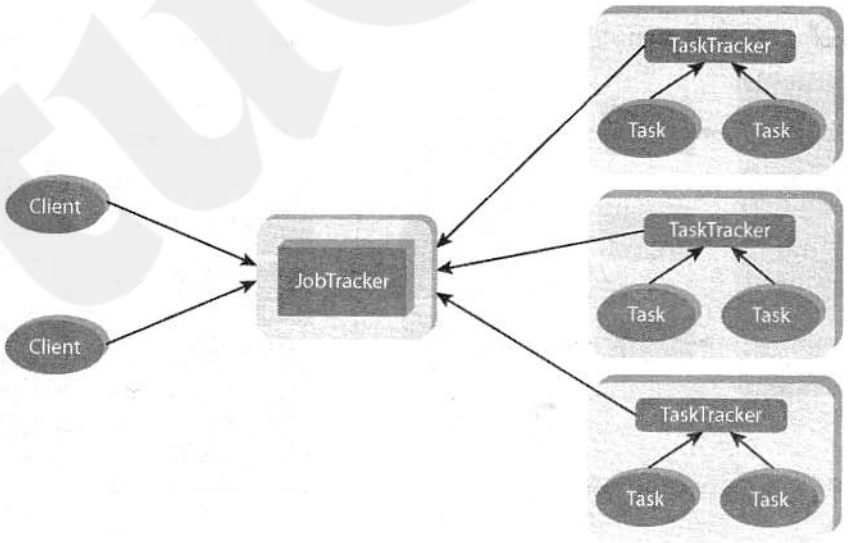


Figure 5.24 MapReduce programming architecture.

3. Several map tasks work simultaneously and read pieces of data that were assigned to each map task. The map worker uses the map function to extract only those data that are present on their server and generates key/value pair for the extracted data.
4. Map worker uses partitioner function to divide the data into regions. Partitioner decides which reducer should get the output of the specified mapper.
5. When the map workers complete their work, the master instructs the reduce workers to begin their work. The reduce workers in turn contact the map workers to get the key/value data for their partition. The data thus received is shuffled and sorted as per keys.
6. Then it calls reduce function for every unique key. This function writes the output to the file.
7. When all the reduce workers complete their work, the master transfers the control to the user program.

5.11.3 MapReduce Example

The famous example for MapReduce Programming is **Word Count.** For example, consider you need to count the occurrences of similar words across 50 files. You can achieve this using MapReduce Programming. Refer Figure 5.25.

Word Count MapReduce Programming using Java

The MapReduce Programming requires three things.

1. **Driver Class:** This class specifies **Job Configuration** details.
2. **Mapper Class:** This class overrides the **Map Function** based on the problem statement.
3. **Reducer Class:** This class overrides the **Reduce Function** based on the problem statement.

Wordcounter.java: Driver Program

```
package com.app;
import java.io.IOException;
```

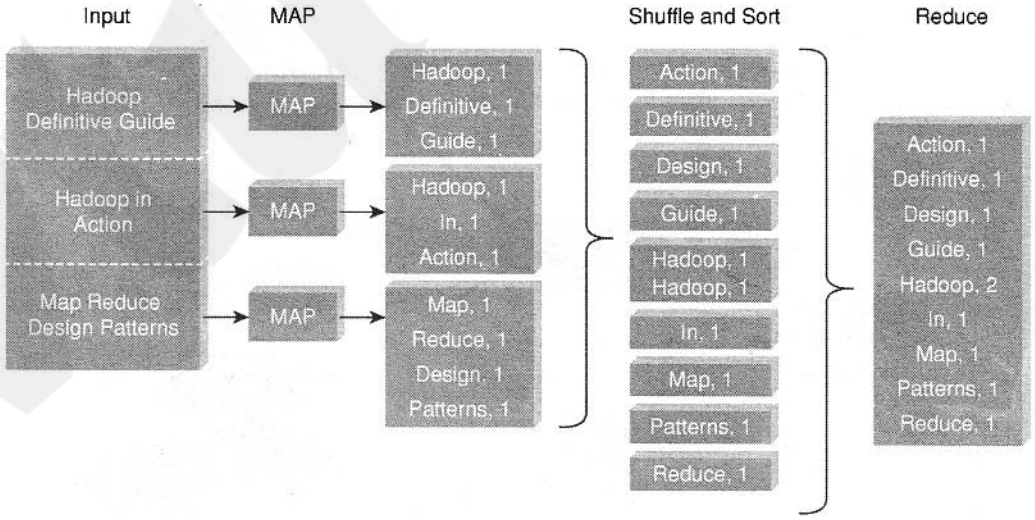


Figure 5.25 Wordcount example.

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCounter {

public static void main (String [] args) throws IOException,
InterruptedException, ClassNotFoundException {
          Job job = new Job ();
          job.setJobName ("wordcounter");
          job.setJarByClass (WordCounter.class);
          job.setMapperClass (WordCounterMap.class);
          job.setReducerClass (WordCounterRed.class);
          job.setOutputKeyClass (Text.class);
          job.setOutputValueClass (IntWritable.class);

          FileInputFormat.addInputPath (job, new Path ("/sample/word.
txt"));
          FileOutputFormat.setOutputPath (job, new Path ("/sample/
wordcount"));
          System.exit (job.waitForCompletion (true)? 0: 1);

     }
}
```

WordCounterMap.java: Map Class

```
package com.app;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordCounterMap extends Mapper<LongWritable, Text, Text,
IntWritable> {
     @Override
```

```
    protected void map (LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {

        String [] words=value.toString ().split (",");
        for (String word: words) {
            context.write (new Text (word), new IntWritable (1));
        }
    }
}
```

WordCountReduce.java: Reduce Class

```
package com.infosys;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordCounterRed extends Reducer<Text, IntWritable, Text,
IntWritable>{

    @Override
    protected void reduce(Text word, Iterable<IntWritable> values,
Context context)
            throws IOException, InterruptedException {
        Integer count = 0;
        for(IntWritable val: values){
            count += val.get();
        }
        context.write(word, new IntWritable(count));
    }
}
```

Table 5.2 describes differences between SQL and MapReduce.

Table 5.2 SQL versus MapReduce

	SQL	MapReduce
Access	Interactive and Batch	Batch
Structure	Static	Dynamic
Updates	Read and write many times	Write once, read many times
Integrity	High	Low
Scalability	Nonlinear	Linear

5.12 MANAGING RESOURCES AND APPLICATIONS WITH HADOOP YARN (YET ANOTHER RESOURCE NEGOTIATOR)

Apache Hadoop YARN is a sub-project of Hadoop 2.x. Hadoop 2.x is YARN-based architecture. It is a general processing platform. YARN is not constrained to MapReduce only. You can run multiple applications in Hadoop 2.x in which all applications share a common resource management. Now Hadoop can be used for various types of processing such as Batch, Interactive, Online, Streaming, Graph, and others.

5.12.1 Limitations of Hadoop 1.0 Architecture

In Hadoop 1.0, HDFS and MapReduce are Core Components, while other components are built around the core.

1. Single NameNode is responsible for managing entire namespace for Hadoop Cluster.
2. It has a restricted processing model which is suitable for batch-oriented MapReduce jobs.
3. Hadoop MapReduce is not suitable for interactive analysis.
4. Hadoop 1.0 is not suitable for machine learning algorithms, graphs, and other memory intensive algorithms.
5. **MapReduce** is responsible for **cluster resource management and data processing**.

In this Architecture, **map slots might be "full", while the reduce slots are empty and vice versa**. This causes **resource utilization issues**. This needs to be improved for proper resource utilization.

5.12.2 HDFS Limitation

NameNode saves all its file metadata in main memory. Although the main memory today is not as small and as expensive as it used to be two decades ago, still there is a limit on the number of objects that one can have in the memory on a single NameNode. The NameNode can quickly become overwhelmed with load on the system increasing.

In Hadoop 2.x, this is resolved with the help of **HDFS Federation**.

5.12.3 Hadoop 2: HDFS

HDFS 2 consists of two major components: **(a) namespace, (b) blocks storage service**. Namespace service takes care of file-related operations, such as creating files, modifying files, and directories. The block storage service handles data node cluster management, replication.

HDFS 2 Features

1. Horizontal scalability.
2. High availability.

HDFS Federation uses multiple independent NameNodes for horizontal scalability. NameNodes are independent of each other. It means, NameNodes does not need any coordination with each other. The DataNodes are common storage for blocks and shared by all NameNodes. All DataNodes in the cluster registers with each NameNode in the cluster.

High availability of NameNode is obtained with the help of **Passive Standby NameNode**. In Hadoop 2.x, Active–Passive NameNode handles failover automatically. All namespace edits are recorded to a shared NFS storage and there is a single writer at any point of time. Passive NameNode reads edits from shared storage

and keeps updated metadata information. In case of Active NameNode failure, Passive NameNode becomes an Active NameNode automatically. Then it starts writing to the shared storage. Figure 5.26 describes the Active–Passive NameNode interaction.

Reference: http://www.edureka.co/blog/introduction-to-hadoop-2-0-and-advantages-of-hadoop-2-0/

Figure 5.27 depicts Hadoop 1.0 and Hadoop 2.0 architecture.

5.12.4 Hadoop 2 YARN: Taking Hadoop beyond Batch

YARN helps us to store all data in one place. We can interact in multiple ways to get predictable performance and quality of services. This was originally architected by **Yahoo**. Refer Figure 5.28.

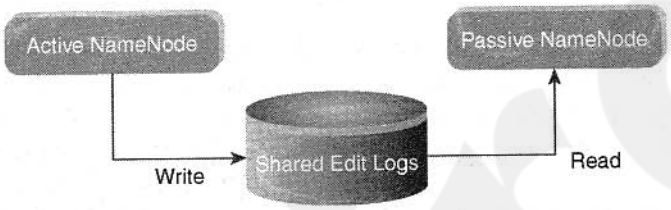


Figure 5.26 Active and Passive NameNode interaction.

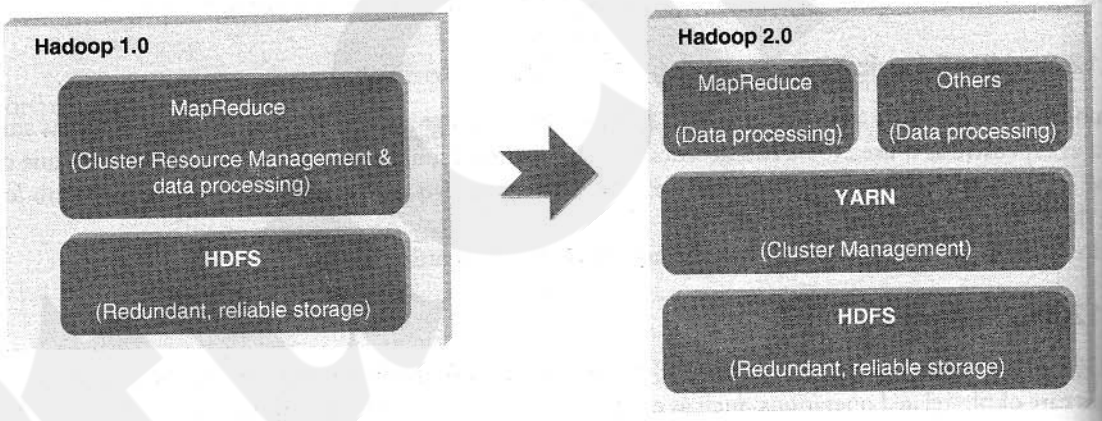


Figure 5.27 Hadoop 1.x versus Hadoop 2.x.

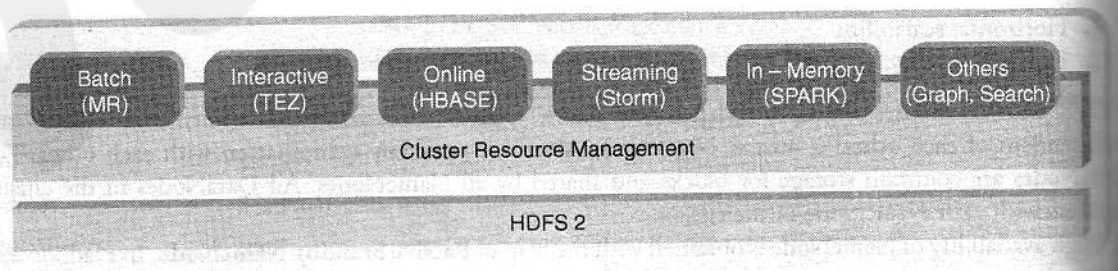


Figure 5.28 Hadoop YARN.

5.12.4.1 Fundamental Idea

The fundamental idea behind this architecture is splitting the JobTracker responsibility of resource management and Job Scheduling/Monitoring into separate daemons. Daemons that are part of YARN Architecture are described below.

1. **A Global ResourceManager:** Its main responsibility is to distribute resources among various applications in the system. It has two main components:
   (a) ***Scheduler:*** The pluggable scheduler of ResourceManager decides allocation of resources to various running applications. The scheduler is just that, a pure scheduler, meaning it does NOT monitor or track the status of the application.
   (b) ***ApplicationManager:*** ApplicationManager does the following:
   - Accepting job submissions.
   - Negotiating resources (container) for executing the application specific ApplicationMaster.
   - Restarting the ApplicationMaster in case of failure.
2. **NodeManager:** This is a per-machine slave daemon. NodeManager responsibility is launching the application containers for application execution. NodeManager monitors the resource usage such as memory, CPU, disk, network, etc. It then reports the usage of resources to the global ResourceManager.
3. **Per-application ApplicationMaster:** This is an application-specific entity. Its responsibility is to negotiate required resources for execution from the ResourceManager. It works along with the NodeManager for executing and monitoring component tasks.

5.12.4.2 Basic Concepts

Application:

1. Application is a job submitted to the framework.
2. Example – **MapReduce Job**.

Container:

1. Basic unit of allocation.
2. Fine-grained resource allocation across multiple resource types (Memory, CPU, disk, network, etc.)
   (a) container_0 = 2GB, 1CPU
   (b) container_1 = 1GB, 6 CPU
3. Replaces the fixed map/reduce slots.

YARN Architecture:

Figure 5.29 depicts YARN architecture. The steps involved in YARN architecture are as follows:

1. A client program submits the application which includes the necessary specifications to launch the application-specific **ApplicationMaster** itself.
2. The ResourceManager launches the ApplicationMaster by assigning some container.
3. The ApplicationMaster, on boot-up, registers with the ResourceManager. This helps the client program to query the ResourceManager directly for the details.
4. During the normal course, ApplicationMaster negotiates appropriate resource containers via the resource-request protocol.

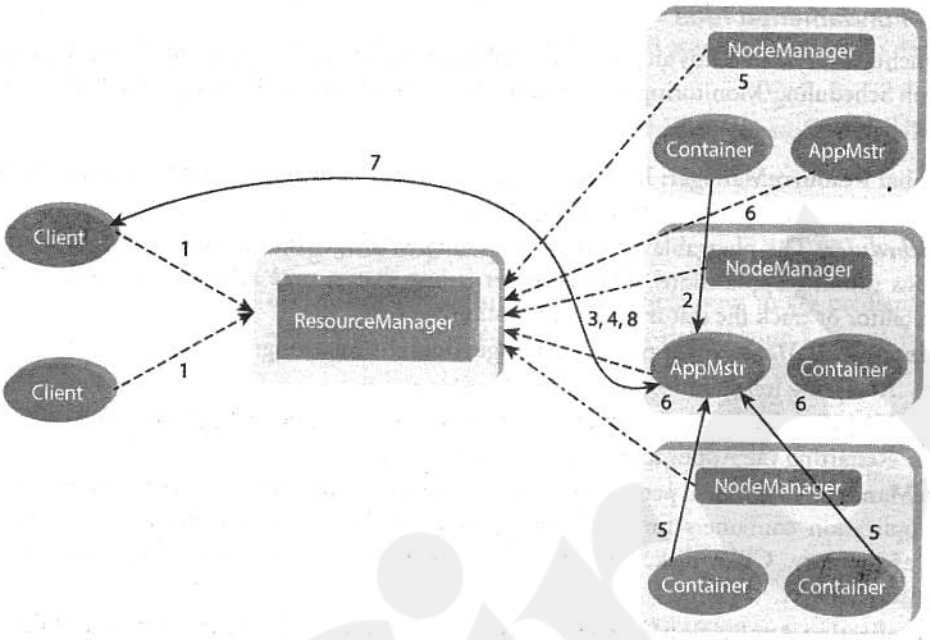


Figure 5.29 YARN architecture.

5. On successful container allocations, the ApplicationMaster launches the container by providing the container launch specification to the NodeManager.
6. The NodeManager executes the application code and provides necessary information such as progress, status, etc. to it's ApplicationMaster via an application-specific protocol.
7. During the application execution, the client that submitted the job directly communicates with the ApplicationMaster to get status, progress updates, etc. via an application-specific protocol.
8. Once the application has been processed completely, ApplicationMaster deregisters with the ResourceManager and shuts down, allowing its own container to be repurposed.

Reference: http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/

8.1 INTRODUCTION

In MapReduce Programming, Jobs (Applications) are split into a set of map tasks and reduce tasks. Then these tasks are executed in a distributed fashion on Hadoop cluster. Each task processes small subset of data that has been assigned to it. This way, Hadoop distributes the load across the cluster. MapReduce job takes a set of files that is stored in HDFS (Hadoop Distributed File System) as input.

Map task takes care of loading, parsing, transforming, and filtering. The responsibility of reduce task is grouping and aggregating data that is produced by map tasks to generate final output. Each map task is broken into the following phases:

1. RecordReader.
2. Mapper.
3. Combiner.
4. Partitioner.

The output produced by map task is known as intermediate keys and values. These intermediate keys and values are sent to reducer. The reduce tasks are broken into the following phases:

1. Shuffle.
2. Sort.
3. Reducer.
4. Output Format.

Hadoop assigns map tasks to the DataNode where the actual data to be processed resides. This way, Hadoop ensures data locality. Data locality means that data is not moved over network; only computational code is moved to process data which saves network bandwidth.

8.2 MAPPER

A mapper maps the input key–value pairs into a set of intermediate key–value pairs. Maps are individual tasks that have the responsibility of transforming input records into intermediate key–value pairs.

1. **RecordReader:** RecordReader converts a byte-oriented view of the input (as generated by the InputSplit) into a record-oriented view and presents it to the Mapper tasks. It presents the tasks with keys and values. Generally the key is the positional information and value is a chunk of data that constitutes the record.
2. **Map:** Map function works on the key–value pair produced by RecordReader and generates zero or more intermediate key–value pairs. The MapReduce decides the key–value pair based on the context.
3. **Combiner:** It is an optional function but provides high performance in terms of network bandwidth and disk space. It takes intermediate key–value pair provided by mapper and applies user-specific aggregate function to only that mapper. It is also known as local reducer.
4. **Partitioner:** The partitioner takes the intermediate key–value pairs produced by the mapper, splits them into shard, and sends the shard to the particular reducer as per the user-specific code. Usually, the key with same values goes to the same reducer. The partitioned data of each map task is written to the local disk of that machine and pulled by the respective reducer.

8.3 REDUCER

The primary chore of the Reducer is to reduce a set of intermediate values (the ones that share a common key) to a smaller set of values. The Reducer has three primary phases: Shuffle and Sort, Reduce, and Output Format.

1. **Shuffle and Sort:** This phase takes the output of all the partitioners and downloads them into the local machine where the reducer is running. Then these individual data pipes are sorted by keys which produce larger data list. The main purpose of this sort is grouping similar words so that their values can be easily iterated over by the reduce task.
2. **Reduce:** The reducer takes the grouped data produced by the shuffle and sort phase, applies reduce function, and processes one group at a time. The reduce function iterates all the values associated with that key. Reducer function provides various operations such as aggregation, filtering, and combining data. Once it is done, the output (zero or more key–value pairs) of reducer is sent to the output format.
3. **Output Format:** The output format separates key–value pair with tab (default) and writes it out to a file using record writer.

Figure 8.1 describes the chores of Mapper, Combiner, Partitioner, and Reducer for the word count problem. The Word Count problem has been discussed under "Combiner" and "Partitioner".

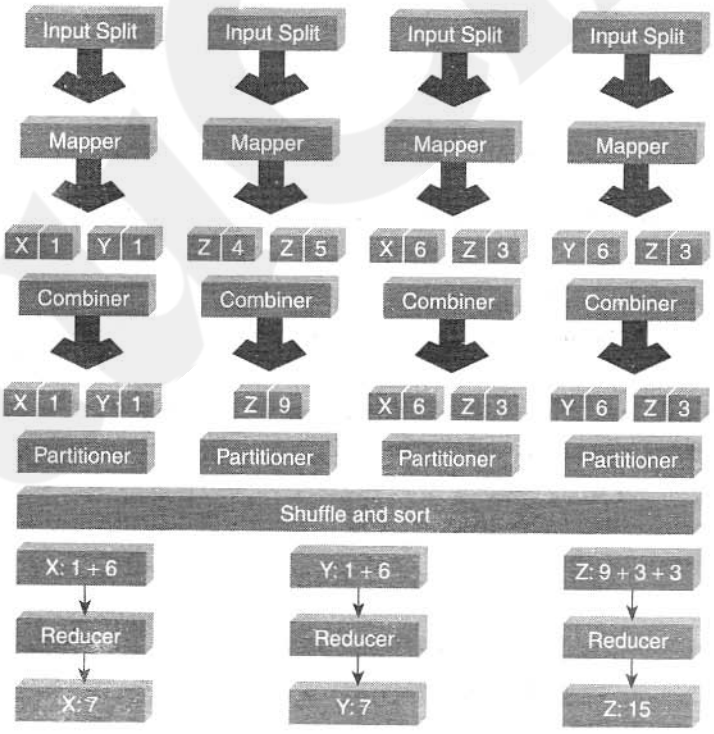


Figure 8.1 The chores of Mapper, Combiner, Partitioner, and Reducer.

8.4 COMBINER

It is an optimization technique for MapReduce Job. Generally, the reducer class is set to be the combiner class. The difference between combiner class and reducer class is as follows:

1. Output generated by combiner is intermediate data and it is passed to the reducer.
2. Output of the reducer is passed to the output file on disk.

The sections have been designed as follows:

Objective: What is it that we are trying to achieve here?
Input Data: What is the input that has been given to us to act upon?
Act: The actual statement/command to accomplish the task at hand.
Output: The result/output as a consequence of executing the statement.

Objective: Write a MapReduce program to count the occurrence of similar words in a file. Use combiner for optimization.

Note: Refer Chapter 5 – Hadoop for Mapper Class and Reduce Class and Driver Program.

Input Data:

```
Welcome to Hadoop Session
Introduction to Hadoop
Introducing Hive
Hive Session
Pig Session
```

Act: In the driver program, set the combiner class as shown below.

```
job.setCombinerClass(WordCounterRed.class);

// Input and Output Path
FileInputFormat.addInputPath(job, new Path("/mapreducedemos/lines.txt"));
FileOutputFormat.setOutputPath(job, new Path("/mapreducedemos/output/wordcount/"));
```

hadoop jar <<jar name>> <<driver class>> <<input path>> <<output path>>

Here driver class name, input path, and output path are optional arguments.

Output:

```
[root@volgalnx010 mapreducedemos]# hadoop jar wordcount.jar
```

Contents of directory /mapreducedemos

Goto : /mapreducedemos go.

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
lines.txt	file	91 B	3	128 MB	2015-03-01 21:05	rw-r--r--	root	supergroup
output	dir				2015-03-01 23:21	rwxr-xr-x	root	supergroup

Go back to DFS home

Local logs

Contents of directory /mapreducedemos/output

Goto : /mapreducedemos/output go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
wordcount	dir				2015-03-01 23:21	rwxr-xr-x	root	supergroup

Go back to DFS home

Local logs

The reducer output will be stored in part-r-00000 file by default.

Contents of directory /mapreducedemos/output/wordcount

Goto : /mapreducedemos/output/wo go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
_SUCCESS	file	0 B	3	128 MB	2015-03-01 23:21	rw-r--r--	root	supergroup
part-r-00000	file	76 B	3	128 MB	2015-03-01 23:21	rw-r--r--	root	supergroup

Go back to DFS home

Local logs

File: /mapreducedemos/output/wordcount/part-r-00000

Goto : /mapreducedemos/output/wo go

Go back to dir listing
Advanced view/download options

```
Hadoop  2
Hive    2
Introducing     1
Introduction    1
Pig     1
Session 3
Welcome 1
to      2
```

8.5 PARTITIONER

The partitioning phase happens after map phase and before reduce phase. Usually the number of partitions are equal to the number of reducers. The default partitioner is hash partitioner.

Objective: Write a MapReduce program to count the occurrence of similar words in a file. Use partitioner to partition key based on alphabets.
Note: Refer Chapter 5 – Hadoop for Mapper Class and Reduce Class and Driver Program.

Input Data:

```
Welcome to Hadoop Session
Introduction to Hadoop
Introducing Hive
Hive Session
Pig Session
```

Act:

WordCountPartitioner.java

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class WordCountPartitioner extends Partitioner<Text, IntWritable> {
        @Override
        public int getPartition(Text key, IntWritable value, int numPartitions) {
                        String word = key.toString();
                        char alphabet = word.toUpperCase().charAt(0);
                        int partitionNumber = 0;
                        switch(alphabet) {
                        case 'A': partitionNumber = 1; break;
                        case 'B': partitionNumber = 2; break;
                        case 'C': partitionNumber = 3; break;
                        case 'D': partitionNumber = 4; break;
                        case 'E': partitionNumber = 5; break;
                        case 'F': partitionNumber = 6; break;
                        case 'G': partitionNumber = 7; break;
                        case 'H': partitionNumber = 8; break;
                        case 'I': partitionNumber = 9; break;
                        case 'J': partitionNumber = 10; break;
                        case 'K': partitionNumber = 11; break;
                        case 'L': partitionNumber = 12; break;
                        case 'M': partitionNumber = 13; break;
                        case 'N': partitionNumber = 14; break;
                        case 'O': partitionNumber = 15; break;
                        case 'P': partitionNumber = 16; break;
                        case 'Q': partitionNumber = 17; break;
                        case 'R': partitionNumber = 18; break;
                        case 'S': partitionNumber = 19; break;
                        case 'T': partitionNumber = 20; break;
                        case 'U': partitionNumber = 21; break;
                        case 'V': partitionNumber = 22; break;
                        case 'W': partitionNumber = 23; break;
                        case 'X': partitionNumber = 24; break;
                        case 'Y': partitionNumber = 25; break;
                        case 'Z': partitionNumber = 26; break;
                        default: partitionNumber = 0; break;
                        }
                        return partitionNumber;
        }
}
```

In the driver program, set the partitioner class as shown below:

```
job.setNumReduceTasks(27);
job.setPartitionerClass(WordCountPartitioner.class);

// Input and Output Path
FileInputFormat.addInputPath(job, new Path("/mapreducedemos/lines.txt"));
FileOutputFormat.setOutputPath(job, new Path("/mapreducedemos/output/
wordcountpartitioner/"));
```

Output:

You can see 27 partitions in the below output.

Contents of directory /mapreducedemos/output/wordcountpartitioner

Goto : /mapreducedemos/output/wo go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
_SUCCESS	file	0 B	3	128 MB	2015-03-01 23:40	rw-r--r--	root	supergroup
part-r-00000	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00001	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00002	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00003	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00004	file	0 B	3	128 MB	2015-03-01 23:40	rw-r--r--	root	supergroup
part-r-00005	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00006	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00007	file	0 B	3	128 MB	2015-03-01 23:40	rw-r--r--	root	supergroup
part-r-00008	file	16 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00009	file	29 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00010	file	0 B	3	128 MB	2015-03-01 23:40	rw-r--r--	root	supergroup
part-r-00011	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00012	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00013	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00014	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00015	file	0 B	3	128 MB	2015-03-01 23:40	rw-r--r--	root	supergroup
part-r-00016	file	6 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00017	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00018	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00019	file	10 B	3	128 MB	2015-03-01 23:40	rw-r--r--	root	supergroup
part-r-00020	file	5 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00021	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00022	file	0 B	3	128 MB	2015-03-01 23:40	rw-r--r--	root	supergroup
part-r-00023	file	10 B	3	128 MB	2015-03-01 23:40	rw-r--r--	root	supergroup
part-r-00024	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00025	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup
part-r-00026	file	0 B	3	128 MB	2015-03-01 23:39	rw-r--r--	root	supergroup

The output file part-r-00008 is associated with alphabet **'H'**.

File: /mapreducedemos/output/wordcountpartitioner/part-r-00008

Goto : /mapreducedemos/output/wo go

Go back to dir listing

Advanced view/download options

Hadoop 2
Hive 2

8.6 SEARCHING

Objective: To write a MapReduce program to search for a specific keyword in a file.

Input Data:

```
1001,John,45
1002,Jack,39
1003,Alex,44
1004,Smith,38
1005,Bob,33
```

Act:

WordSearcher.java

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordSearcher {

        public static void main(String[] args) throws IOException,
                        InterruptedException, ClassNotFoundException {
                Configuration conf = new Configuration();
                Job job = new Job(conf);
                job.setJarByClass(WordSearcher.class);
                job.setOutputKeyClass(Text.class);
                job.setOutputValueClass(Text.class);
                job.setMapperClass(WordSearchMapper.class);
                job.setReducerClass(WordSearchReducer.class);
                job.setInputFormatClass(TextInputFormat.class);
                job.setOutputFormatClass(TextOutputFormat.class);
                job.setNumReduceTasks(1);
                job.getConfiguration().set("keyword", "Jack");
                FileInputFormat.setInputPaths(job, new Path("/mapreduce/student.csv"));
```

```
                FileOutputFormat.setOutputPath(job, new Path("/mapreduce/output/search"));
                System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
}
```

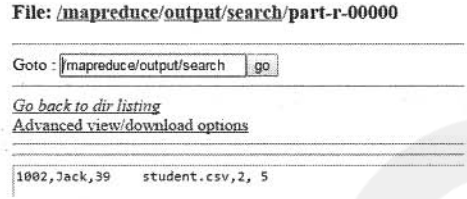
WordSearchMapper.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class WordSearchMapper extends Mapper<LongWritable, Text, Text, Text> {

        static String keyword;
        static int pos = 0;

        protected void setup(Context context) throws IOException,
                        InterruptedException {

                Configuration configuration = context.getConfiguration();
                keyword = configuration.get("keyword");
        }

        protected void map(LongWritable key, Text value, Context context)
                        throws IOException, InterruptedException {

                InputSplit i = context.getInputSplit(); // Get the input split for this map.
                FileSplit f = (FileSplit) i;
                String fileName = f.getPath().getName();
                Integer wordPos;
                pos++;
                if (value.toString().contains(keyword)) {
                        wordPos = value.find(keyword);
                        context.write(value, new Text(fileName + ","+ new IntWritable(pos).
toString() + ", "+ wordPos.toString()));
                }
        }
}
```

WordSearchReducer.java

```
import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordSearchReducer extends Reducer<Text, Text, Text, Text> {

        protected void reduce(Text key, Text value, Context context)
                        throws IOException, InterruptedException {

                context.write(key, value);
        }
}
```

Output:



8.7 SORTING

Objective: To write a MapReduce program to sort data by student name (value).

Input Data:

```
1001,John,45
1002,Jack,39
1003,Alex,44
1004,Smith,38
1005,Bob,33
```

Act:

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class SortStudNames {

    public static class SortMapper extends
            Mapper<LongWritable, Text, Text, Text> {

        protected void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {

            String[ ] token = value.toString().split(",");
            context.write(new Text(token[1]), new Text(token[0]+" - "+token[1]));
        }
    }
    // Here, value is sorted...
    public static class SortReducer extends
            Reducer<Text, Text, NullWritable, Text> {

        public void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {

            for (Text details : values) {
                context.write(NullWritable.get(), details);
            }
        }
    }

    public static void main(String[ ] args) throws IOException,
            InterruptedException, ClassNotFoundException {

        Configuration conf = new Configuration();
        Job job = new Job(conf);
        job.setJarByClass(SortEmpNames.class);
        job.setMapperClass(SortMapper.class);
        job.setReducerClass(SortReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        FileInputFormat.setInputPaths(job, new Path("/mapreduce/student.csv"));
        FileOutputFormat.setOutputPath(job, new
Path("/mapreduce/output/sorted/"));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }

}
```

Output:

File: /mapreduce/output/search/part-r-00000

Goto : /mapreduce/output/search go

Go back to dir listing
Advanced view/download options

```
1002,Jack,39    student.csv,2, 5
```

8.8 COMPRESSION

In MapReduce programming, you can compress the MapReduce output file. Compression provides two benefits as follows:

1. Reduces the space to store files.
2. Speeds up data transfer across the network.

You can specify compression format in the Driver Program as shown below:

```
conf.setBoolean("mapred.output.compress", true);
conf.setClass("mapred.output.compression.codec", GzipCodec.class,CompressionCodec.class);
```

Here, codec is the implementation of a compression and decompression algorithm. GzipCodec is the compression algorithm for gzip. This compresses the output file.