Department of Computer Science and Engineering

Bigdata and Analytics (BCS714D)

Module-02

Introduction to Hadoop and Map Reduce Programming

Introduction to Hadoop

- Hadoop is an open-source framework designed to store and process massive volumes of data efficiently using a distributed file system (HDFS) and a distributed computing model.
- It enables organizations to manage and analyze structured, semi-structured, and unstructured data spread across networks.
- Hadoop uses MapReduce programming to process data in parallel across clusters of computers.
- Understanding Hadoop is crucial due to the exponential growth of data generated daily by enterprises worldwide.

Big Data and Hadoop

- Big Data refers to the huge volumes of diverse data generated every second, minute, and day.
- Enterprises are increasingly recognizing Big Data as a valuable resource for insights and innovation.

Data Generation Statistics

- **Every Day:**
  - NYSE generates 1.5 billion shares and trade data.
  - Facebook stores 2.7 billion comments and likes.
  - Google processes about 24 petabytes of data.
- **Every Minute:**
  - Facebook users share 2.5 million pieces of content.
  - Twitter users tweet 300,000 times.
  - Instagram users post 220,000 new photos.

- YouTube users upload 72 hours of new videos.
- Apple users download 50,000 apps.
- Email users send 200 million messages.
- Amazon generates over $80,000 in sales.
- Google handles over 4 million search queries.

- **Every Second:**
  - Banking applications process over 10,000 credit card transactions.

The Need for Hadoop

- To handle, process, and analyze the vast and varied data efficiently, a scalable system like Hadoop is essential.
- Hadoop addresses challenges related to storing and processing colossal volumes of data by distributing the workload.

5.1.1 Data: The Treasure Trove

- Provides business advantages such as generating product recommendations, inventing new products, and market analysis.
- Offers key indicators that can influence business success.
- Allows for **more precise analysis** — the more data available, the better the accuracy of insights.

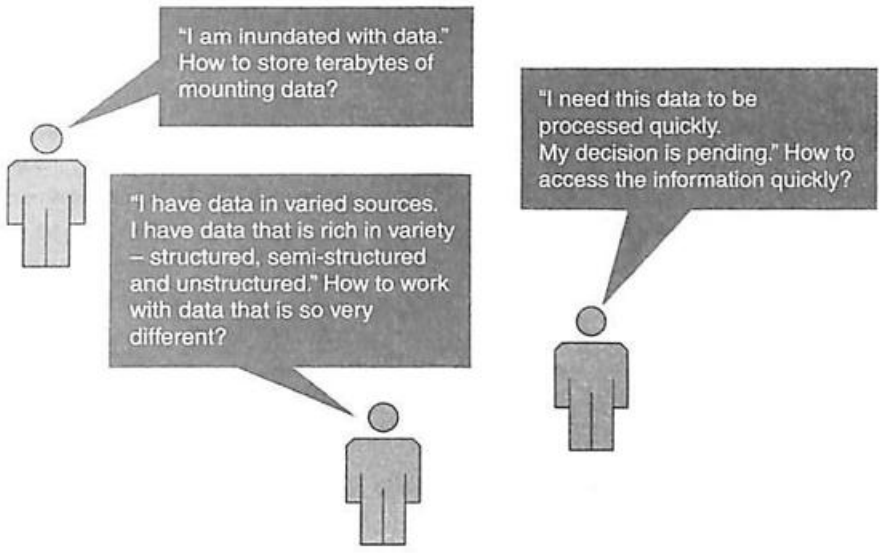


Figure 5.1 Challenges with big volume, variety, and velocity of data.

Challenges Hadoop Addresses (Figure 5.1):

- **Volume**: "How to store terabytes of mounting data?"
- **Variety**: "How to work with data from varied sources – structured, semi-structured, unstructured?"
- **Velocity**: "How to process data quickly for timely decision-making?"

5.2 Why Hadoop?

Hadoop is widely adopted because of its **ability to handle massive amounts of data**, in **varied formats**, and **process it quickly**. Here's a breakdown of the key points:

Other Considerations for Using Hadoop (Figure 5.2):

1. **Low Cost**
   - Open-source framework
   - Uses **commodity hardware**, which is inexpensive and easy to obtain.
2. **Inherent Data Protection**
   - Built-in fault tolerance and data replication across nodes.
3. **Storage Flexibility**
   - Capable of storing **structured, semi-structured, and unstructured** data.
4. **Scalability**
   - Can scale horizontally by adding more machines (nodes).
5. **Computing Power**
   - Distributed processing allows for **parallel computation**, reducing processing time.

Hadoop provides a **cost-effective**, **scalable**, and **efficient** way to manage the challenges of Big Data—making it one of the most in-demand technologies in data-driven enterprises.

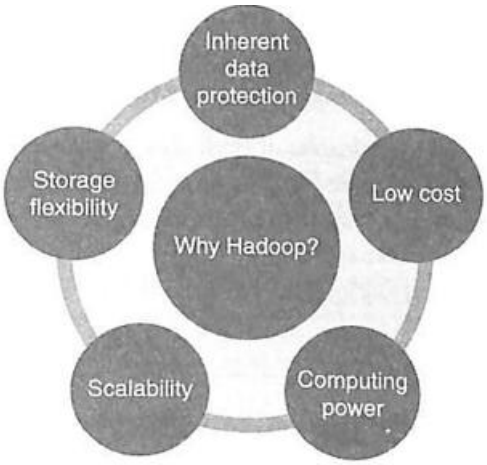


Figure 5.2 Key considerations of Hadoop.

- **Key Benefits of Hadoop (Continued from Figure 5.2):**

1. **Computing Power**
   - Based on a **distributed computing model**.
   - Can process **very large volumes of data quickly**.
   - More computing nodes = More processing power.
2. **Scalability**
   - Easily scalable by **adding more nodes**.
   - Requires minimal system administration as the system grows.
3. **Storage Flexibility**
   - Unlike traditional RDBMS, **no need to pre-process** data before storing it.
   - Can store **structured, semi-structured, and unstructured data** like images, videos, text.
   - Allows flexibility in deciding **how to use stored data later**.
4. **Inherent Data Protection**
   - Automatically redirects tasks if a node fails.
   - Ensures system reliability and **high availability**.
   - Stores **replicas** of data across different nodes in a **cluster** to prevent data loss.

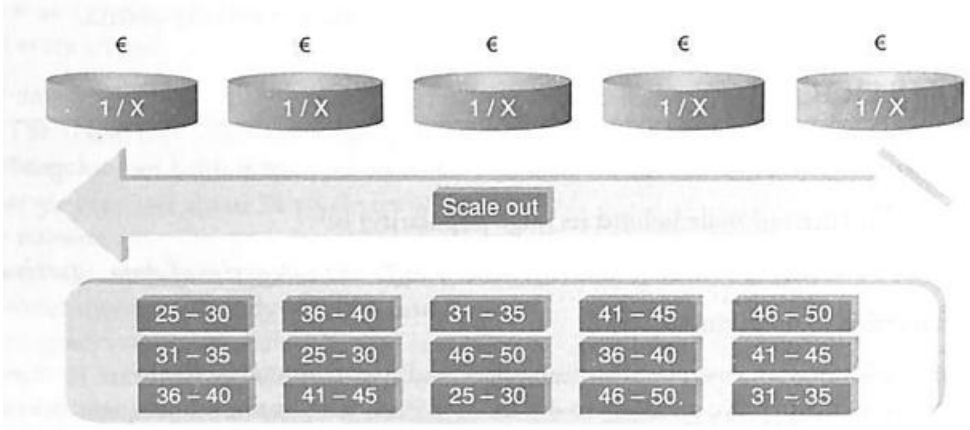


Figure 5.3 Hadoop framework (distributed file system, commodity hardware).

Hadoop Framework Overview (Figure 5.3)

- **Cluster-based design**: A group of machines (nodes) that work together.
- Data is:
  1. **Distributed and duplicated** across nodes.
  2. **Processed in parallel** using local resources.
  3. Managed with **automatic failover and fault tolerance**.

5.3 Why Not RDBMS?

- RDBMS is **not ideal** for:
  - Handling **large files, images, and videos**.
  - **Advanced analytics** and **machine learning** workloads.
- Requires:
  - **High investment** for scalability.
  - More complex architecture and **higher storage costs**.
- **Hadoop outperforms RDBMS** in terms of **cost-efficiency**, **scalability**, and **flexibility** for large, complex data sets.

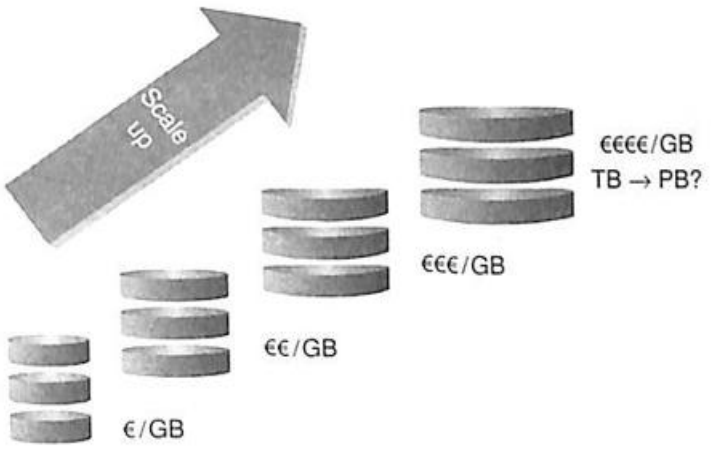


Figure 5.4 RDBMS with respect to cost/GB of storage.

Figure 5.4: RDBMS Cost Scaling

- As data size increases (e.g., TB → PB), cost per GB grows rapidly with RDBMS.
- Hadoop provides better scalability at much lower cost.

5.4 RDBMS vs Hadoop

Parameter	RDBMS	Hadoop
System	Relational Database Management System	Node-Based Flat Structure
Data	Suitable for structured data only	Supports structured, semi-structured, and unstructured data (e.g., XML, JSON, text files)
Processing	OLTP (Online Transaction Processing)	Analytical, Big Data Processing
Choice	Ideal for consistent relationships between data	Ideal for Big Data with no need for consistency
Processor	Requires expensive/high-end hardware	Uses commodity hardware (basic processor, network card, and a few hard drives)
Cost	~$10,000–$14,000 per terabyte	~$4,000 per terabyte

5.6 HISTORY OF HADOOP

- **Hadoop was created by Doug Cutting**, who also developed **Apache Lucene**, a popular text search library.
- Hadoop originated as a part of the **Apache Nutch project** (an open-source web search engine).
- It is also tied to the **Lucene project**.

◇ 5.6.1 The Name "Hadoop"

- **"Hadoop" is not an acronym**.
- It's a **made-up name** given by Doug Cutting's son to a **stuffed yellow toy elephant**.

- Doug Cutting chose the name because it was:
  - Short and easy to spell
  - Easy to pronounce
  - Meaningless (not previously used)
  - Memorable and unique

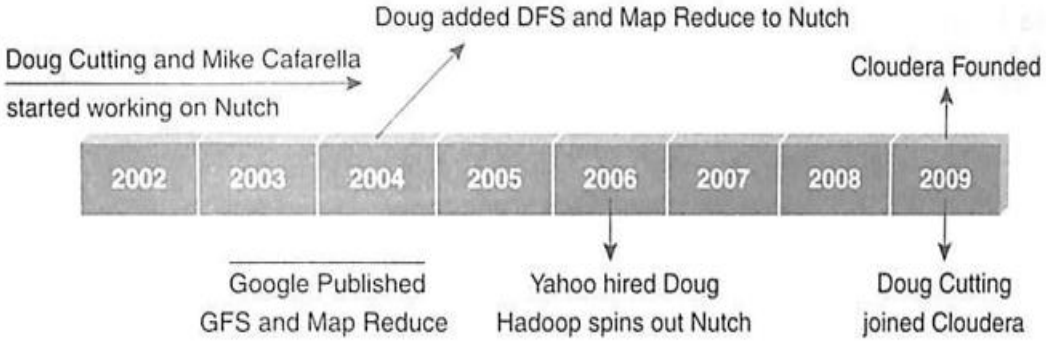


Figure 5.6 Hadoop history.

Year	Event
2002	Doug Cutting and Mike Cafarella started working on **Nutch**.
2003	Continued development of Nutch.
2004	Google published **GFS and MapReduce** papers.
2005	Doug added **DFS and MapReduce** concepts to Nutch.
2006	Yahoo hired Doug; Hadoop **spins out of Nutch** as a separate project.
2008	**Cloudera founded**.
2009	Doug Cutting joined **Cloudera**.

What is Hadoop?

- Hadoop is an **open-source software framework**.
- It is used to **store and process massive amounts of data** in a **distributed manner**.
- Operates on **large clusters of commodity hardware** (low-cost machines).

- **Core Objectives of Hadoop**

1. **Massive Data Storage**
2. **Faster Data Processing**

5.7.1 Key Aspects of Hadoop

Aspect	Description
✅ **Open Source Software**	Free to **download**, **use**, and **contribute to**.
✅ **Framework**	Includes **tools, programs, and infrastructure** needed for development and execution.
✅ **Distributed**	Data is **divided and stored across multiple machines**. Processing is **parallelized**.
✅ **Massive Storage**	Stores **huge volumes** of data using **low-cost hardware**.
✅ **Faster Processing**	Enables **quick response times** by processing data **in parallel** across nodes.

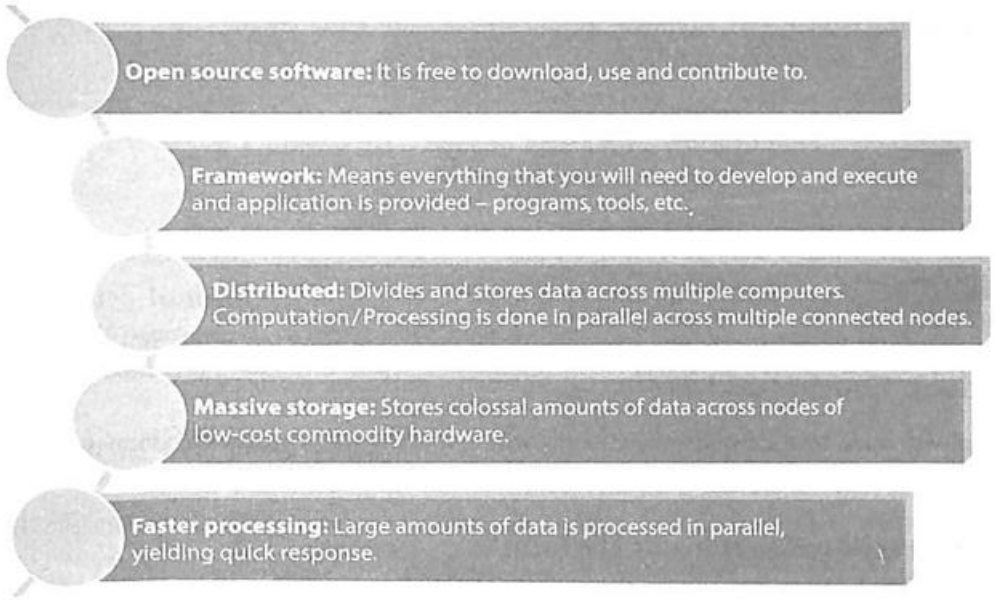


Figure 5.7 Key aspects of Hadoop.

5.7.2 Hadoop Components

Core Components:

1. **HDFS (Hadoop Distributed File System)**
   - (a) Acts as the **storage component**
   - (b) **Distributes data** across nodes
   - (c) **Redundant by design** (stores multiple copies)
2. **MapReduce**
   - (a) A **computational framework**
   - (b) **Splits tasks** across multiple nodes
   - (c) **Processes data in parallel**

🔧 **Hadoop Ecosystem Tools (Enhance Core Capabilities):**

1. **HIVE** – SQL-like querying of big data
2. **PIG** – Scripting for data transformation
3. **SQOOP** – Transfers data between RDBMS and Hadoop
4. **HBASE** – NoSQL database on Hadoop
5. **FLUME** – Collects and moves large amounts of log data
6. **OOZIE** – Workflow scheduler
7. **MAHOUT** – Machine learning algorithms

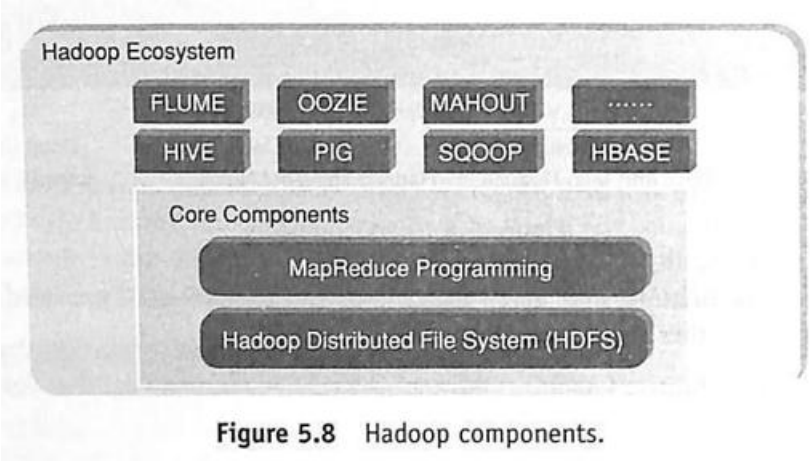


Figure 5.8 Hadoop components.

5.7.3 Hadoop Conceptual Layer

- Conceptually divided into two layers:
    1. **Data Storage Layer** – For storing massive data
    2. **Data Processing Layer** – For **parallel processing** and generating insights

5.7.4 High-Level Architecture of Hadoop

- **Distributed Master-Slave Architecture**:
    - **Master Node** = **NameNode**
    - **Slave Nodes** = **DataNodes**
- The **NameNode manages** metadata and coordination.
- **DataNodes handle** storage and actual data processing.

5.8 Use case of Hadoop

Three Key Benefits:

1. **Data Integration**:
    - Combines ClickStream data with:
        - CRM (Customer Relationship Management) data
        - Sales data
        - Advertising campaign info
    - Gives **deeper customer insights**
2. **Scalability + Cost Efficiency**:
    - Stores **years of data** at **low incremental cost**
    - Enables long-term or year-over-year trend analysis
    - Gives **competitive edge** in understanding customer behavior over time
3. **Powerful Analytics with Tools**:
    - Tools: **Apache Pig** or **Apache Hive**

- Allows:
  - Organizing data by user sessions
  - Refinement and analysis
  - Feeding data into **visualization or analytics tools.**

5.8.1 ClickStream Data

- **ClickStream data** = User activity data (e.g., mouse clicks on websites)
- Helps understand **customer purchasing behavior**
- Used by online marketers to:
  - Optimize product pages
  - Improve promotional content
  - Enhance overall business strategy

🔹 **Benefits of Using Hadoop for ClickStream Data Analysis**

(Refer to Figure 5.11)

🔑 **Three Key Benefits:**

1. **Data Integration**:
   - Combines ClickStream data with:
     - CRM (Customer Relationship Management) data
     - Sales data
     - Advertising campaign info
   - Gives **deeper customer insights**
2. **Scalability + Cost Efficiency**:
   - Stores **years of data** at **low incremental cost**
   - Enables long-term or year-over-year trend analysis
   - Gives **a competitive edge** in understanding customer behavior over time
3. **Powerful Analytics with Tools**:

- Tools: **Apache Pig** or **Apache Hive**
- Allows:
  - Organizing data by user sessions
  - Refinement and analysis
  - Feeding data into **visualization or analytics tools**

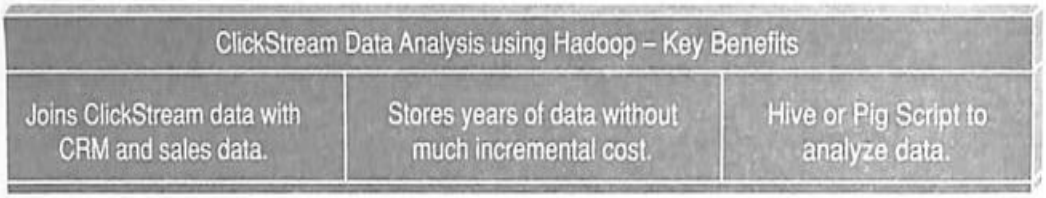


Figure 5.11 ClickStream data analysis.

5.10 HDFS (Hadoop Distributed File System)

- **Key Features of HDFS**

1. **Storage Component** of the Hadoop framework.
2. A **Distributed File System** that stores data across multiple nodes.
3. **Modeled after Google File System (GFS)**.
4. **High throughput optimization**:
   - Uses **large block sizes**
   - Moves **computation to the data**, not vice versa
5. **Replication**:
   - Files can be replicated **multiple times** (configurable), making the system **tolerant to hardware/software failures**
6. **Automatic re-replication** of data blocks if a node fails
7. **Designed for large files** (e.g., GBs to TBs) — optimized for reading/writing large datasets
8. Built **on top of native file systems** like **ext3/ext4**

- **Example:**
  - A file Sample.txt of **192 MB**
  - **Default block size** = **64 MB**
  - File will be split into **3 blocks** (192 ÷ 64)
  - Each block is **replicated across nodes** in the cluster according to the **default replication factor** (usually 3)

5.10.1 HDFS Daemons

5.10.1.1 NameNode

- The NameNode is the master daemon in the HDFS architecture.
- It manages the file system namespace, which includes:
  - Mapping of blocks to files
  - Tracking file metadata
  - Handling read, write, create, delete operations

- Key Responsibilities of NameNode:

1. File System Namespace:
   - A logical representation of files and directories in HDFS.
   - Stored in a file called FsImage.
2. Transaction Logging:
   - Every metadata change is recorded in the EditLog.
   - On startup, NameNode:
     - Reads both FsImage and EditLog
     - Applies the transactions to reconstruct the latest state
     - Writes a new FsImage and clears the old EditLog
3. Rack Awareness:
   - Uses Rack ID to identify the DataNodes in different racks (helps in fault tolerance and efficient data transfer).

4. Single NameNode per Hadoop cluster.

- **Figure 5.13**: Shows the layered structure:
  `Disk Storage → Native OS File System → HDFS`
- **Figure 5.14**: Key defaults in HDFS:
  - **Block Structure**
  - **Replication Factor = 3**
  - **Default Block Size = 64 MB**

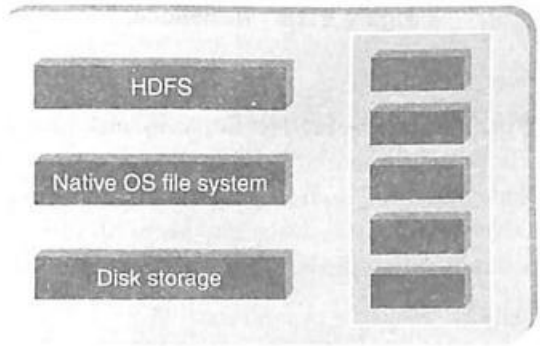


Figure 5.13 Hadoop Distributed File System.

Hadoop Distributed File System – Key Points		
Block Structured File	Default Replication Factor : 3	Default Block Size : 64 MB

Figure 5.14 Hadoop Distributed File System – key points.

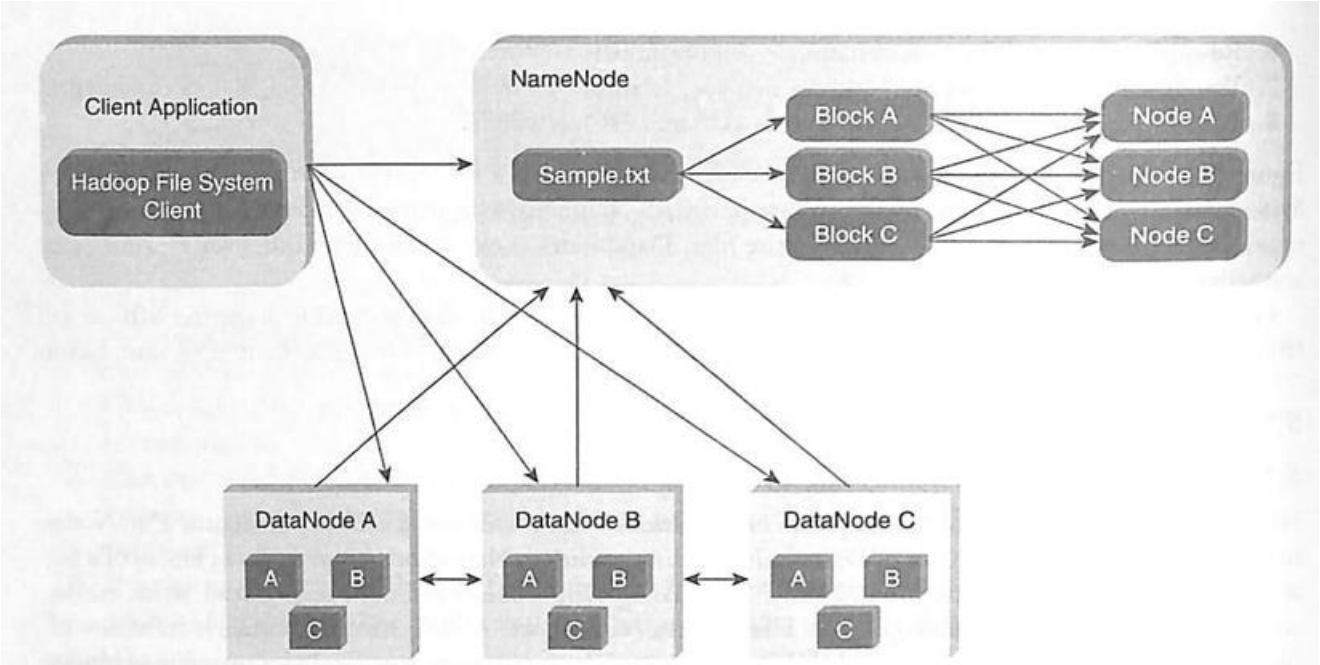


Figure 5.15 Hadoop Distributed File System Architecture.
Reference: Hadoop in Practice, Alex Holmes.

Figure 5.16 NameNode.

5.10.1.2 Data Node

1. Multiple DataNodes exist in a Hadoop cluster.
2. DataNodes communicate with each other and the NameNode during read/write operations.
3. Data Nodes send "heartbeat" messages to the NameNode regularly.
4. The heartbeat confirms that the DataNode is alive and working.
5. If a Data Node fails to send a heartbeat, the NameNode assumes it is down.
6. The NameNode then replicates the data from the failed DataNode to other active DataNodes.
7. This allows the system to keep functioning smoothly even if a DataNode goes down.

Real-Life Example (Analogy):

- Just like **employees swipe in** at the office to mark attendance,
- DataNodes send heartbeat signals to show they are present.
- The manager (like the NameNode) assigns tasks only to those who have checked in.
- If no swipe (heartbeat), the employee (DataNode) is considered absent, and tasks are reassigned.

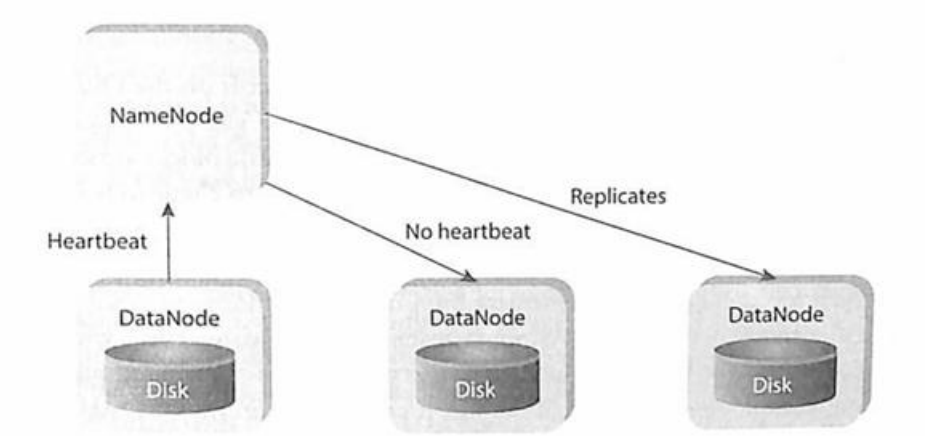


Figure 5.17 NameNode and DataNode Communication.

5.10.2 Anatomy of File Read

1. **Client Opens File**
   - The client uses open() to request a file via the **DistributedFileSystem (DFS)**.
2. **Get Block Location from NameNode**
   - DFS contacts the **NameNode** to get the locations (addresses) of the DataNodes storing the file's data blocks.
3. **Receive Input Stream**
   - DFS gives the client an **FSDataInputStream** to begin reading the file.
4. **Read from Closest DataNode**
   - The client reads the file data block-by-block, starting with the **closest DataNode**.
5. **Read Continues with Next Block**

- After reading one block, the client connects to the next best DataNode for the next block and continues reading.

6. **Close the Stream**
   - When the reading is complete, the client calls close() to close the **FSDataInputStream**.

Flow:

open() → **DFS** → **NameNode** → Block locations → **FSDataInputStream** → **Read from DataNode(s)** → close()

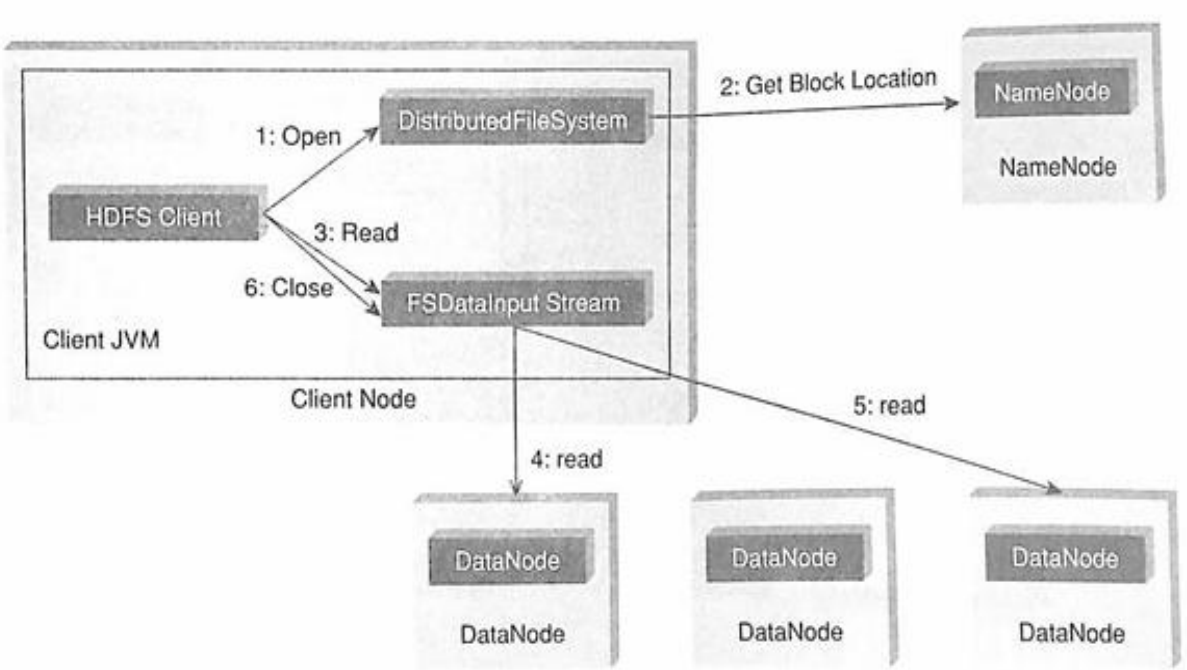


Figure 5.18 File Read.

5.10.3 Anatomy of File Write

1. **Client Requests to Create File**
   - Client calls create() using **DistributedFileSystem** to begin the file creation process.

2. **File Creation Request Sent to NameNode**

- An **RPC** (Remote Procedure Call) is made to the **NameNode** to create a new file.
- NameNode does checks (e.g., if file already exists) and creates a file **without data blocks initially**.

3. **Client Starts Writing Data**
   - The client writes data through **FSDataOutputStream**.
   - Data is split into **packets**, placed into a **data queue**, and processed by **DataStreamer**.
4. **Data Packets Streamed to DataNodes**
   - Packets are streamed to a **pipeline of 3 DataNodes** (default replication factor = 3):
     - First DataNode stores and forwards to second.
     - Second DataNode stores and forwards to third.
     - Third DataNode stores the final copy.
5. **Acknowledgment Process**
   - Each DataNode sends an **acknowledgment (ack)** back up the pipeline.
   - DFSOutputStream maintains an **Ack Queue** to track unacknowledged packets.
   - A packet is removed from this queue only when **all three DataNodes** have acknowledged it.
6. **Client Finishes Writing**
   - Once all data is written, the client calls close() on the stream.
7. **Final Acknowledgment to NameNode**
   - Remaining packets are flushed.
   - NameNode is informed that the file write is complete.

🔁 **Flow:**

create() → **DFS** → **NameNode** → **FSDataOutputStream** → **DataStreamer** → **DataNode Pipeline** → ack packets → close() → **Notify NameNode**

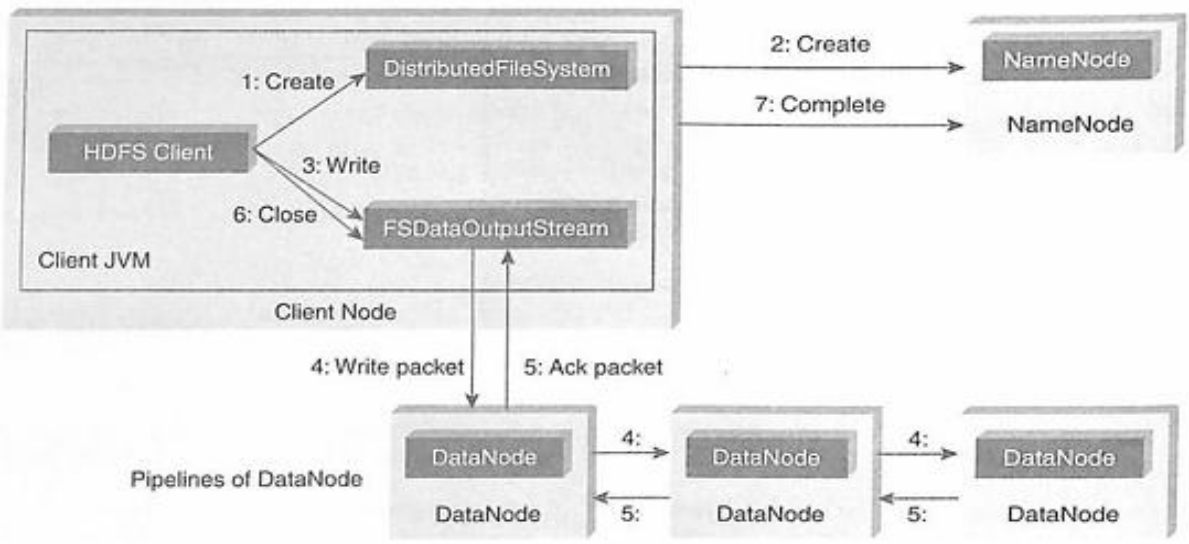


Figure 5.19 File Write.

5.10.4 Replica Placement Strategy

5.10.4.1 Hadoop Default Replica Placement Strategy:

1. First Replica
    - Placed on the same node as the client (local write).
2. Second Replica
    - Placed on a different rack, on a different node (for fault tolerance).
3. Third Replica
    - Placed on the same rack as the second replica, but on a different node.

5.10.5 Working with HDFS Commands

Objective	Command	Description
List root directories/files	`hadoop fs -ls /`	Shows top-level directories and files in HDFS.
List all files and subdirectories recursively	`hadoop fs -ls -R /`	Shows full directory tree in HDFS.
Create a directory (e.g. `/sample`)	`hadoop fs -mkdir /sample`	Creates a new directory in HDFS.

Objective	Command	Description
Upload file to HDFS	`hadoop fs -put /root/sample/test.txt /sample/test.txt`	Copies file from local filesystem to HDFS.
Download file from HDFS	`hadoop fs -get /sample/test.txt /root/sample/testsample.txt`	Copies file from HDFS to local filesystem.
Upload using `copyFromLocal`	`hadoop fs -copyFromLocal /root/sample/test.txt /sample/testsample.txt`	Similar to `put`, copies file from local to HDFS.
Download using `copyToLocal`	`hadoop fs -copyToLocal /sample/test.txt /root/sample/testsample1.txt`	Similar to `get`, copies file from HDFS to local.
Display file content	`hadoop fs -cat /sample/test.txt`	Prints content of a file in HDFS.
Copy file within HDFS	`hadoop fs -cp /sample/test.txt /sample1`	Copies file from one HDFS dir to another.
Remove directory from HDFS	`hadoop fs -rm -r /sample1`	Deletes directory and contents from HDFS.

5.10.6 Special Features of HDFS

Data Replication

- The client doesn't need to manage block locations.
- HDFS automatically directs the client to the **nearest replica** for better **performance**.

Data Pipeline

- When writing data, the client sends it to the **first DataNode**.
- That DataNode **forwards** it to the **second**, and then to the **third**, forming a **pipeline**.
- All replicas are written to disk through this flow.

5.11 – Processing Data with Hadoop

What is MapReduce?

- MapReduce is a **tool in Hadoop** that helps process **big data** in **parallel**.
- It splits the data into parts and processes each part **at the same time**.

How It Works

1. **Map Task**
    - Breaks input into **key-value pairs**.

- Each mapper works **independently**.

2. **Shuffle & Sort**
   - The outputs from all mappers are **shuffled and sorted** by keys.
3. **Reduce Task**
   - Combines mapper outputs into a **final result**.

📍 Key Features

- **Data Locality**: Tries to run the task **where the data is stored** to save time.
- **Stored on Disk**: Outputs from mappers are stored **locally on disk**.

⚙️ Important Components

Component	Role
JobTracker	Master – gives tasks to others.
TaskTracker	Worker – does the task given to it.

🔁 How a Job Runs

1. You (Job Client) send the job to **JobTracker**.
2. **JobTracker** splits it into tasks.
3. **TaskTrackers** do the work.
4. JobTracker **tracks progress** and tells you when done.

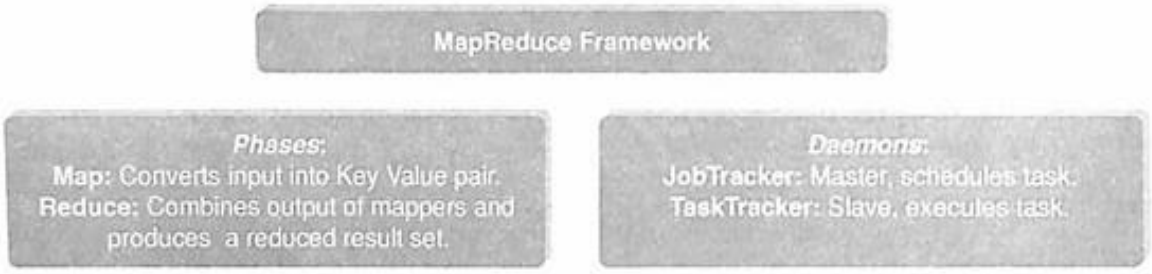


Figure 5.21 MapReduce Programming phases and daemons.

5.11.1 MapReduce Daemons

1. JobTracker (Master)

- Connects your code to the Hadoop cluster.
- Decides **which task runs on which node**.
- **Monitors** running tasks.
- If a task fails, it **retries** on another node.
- Only **one JobTracker** per cluster.

2. TaskTracker (Worker)

- Executes the actual **Map** or **Reduce** tasks given by JobTracker.
- One TaskTracker per slave node.
- Uses **multiple JVMs** to run tasks in parallel.
- Sends regular **heartbeat signals** to JobTracker.
- If JobTracker doesn't get a heartbeat, it assumes failure and reassigns the task.

5.11.2 How does Mapreduce Work

1. **Input Data is Split**
   The big data you want to analyze is split into many small parts. Each part is called a data segment.
2. **JobTracker and TaskTrackers**
   - There is one **JobTracker** (the master) that controls the whole job.
   - There are many **TaskTrackers** (workers) that do the actual work on the data.
3. **Map Tasks**
   - Each TaskTracker gets a small part of the data.
   - They perform the **map** function, which processes the data and creates key-value pairs (like labeling the data).
4. **Shuffle and Sort**
   - The output from the map tasks is shuffled and sorted by keys automatically by the framework.

5. **Reduce Tasks**
    - Reduce workers get the sorted key-value pairs.
    - They combine or summarize the data based on the keys, producing the final result.
6. **Output**
    - The final combined output is saved or sent back to the user program.

Roles

- **JobTracker:** Master controller that assigns tasks to workers.
- **TaskTracker:** Worker nodes that run map and reduce tasks.
- **Map Task:** Processes data pieces to generate key-value pairs.
- **Reduce Task:** Combines data based on keys to produce final results.

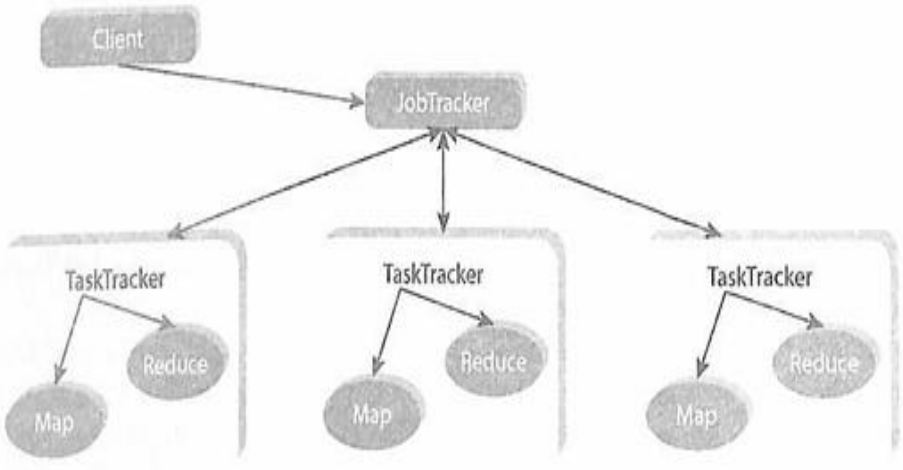


Figure 5.22 JobTracker and TaskTracker interaction.

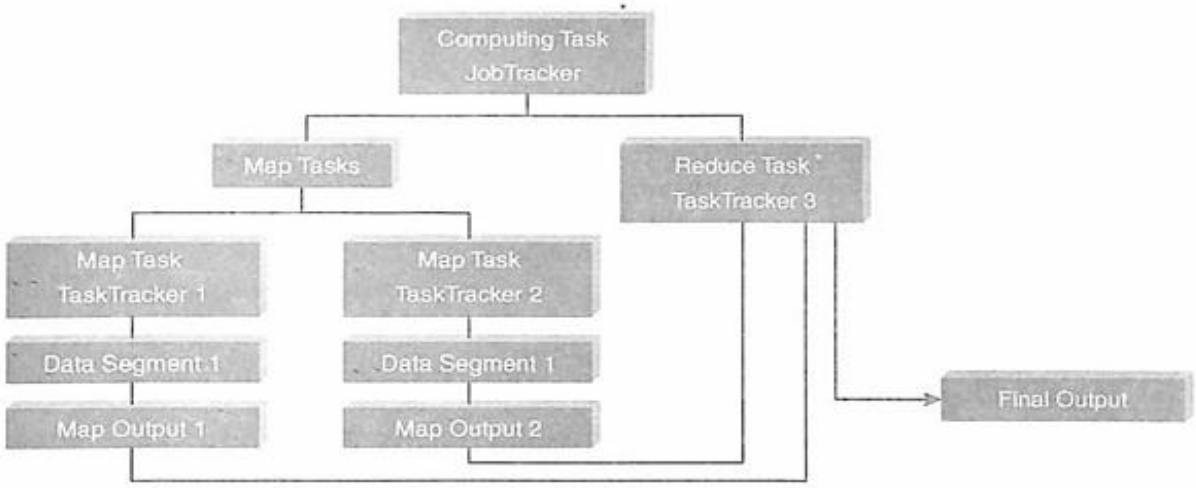


Figure 5.23 MapReduce programming workflow.

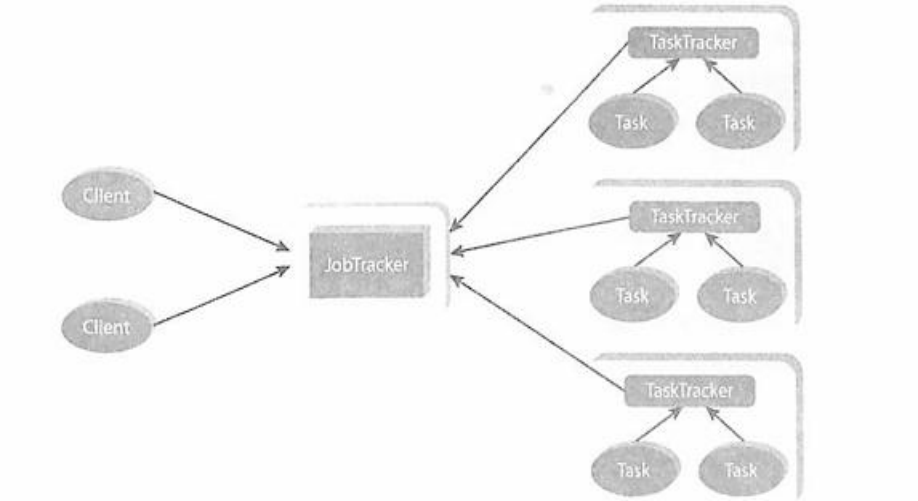


Figure 5.24 MapReduce programming architecture.

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.

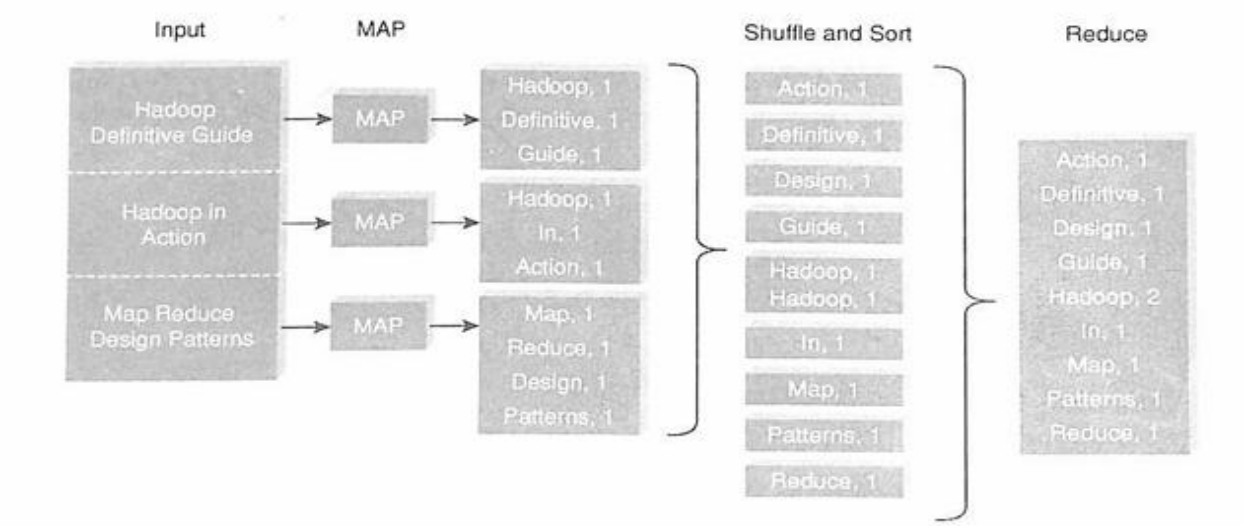


Figure 5.25 Wordcount example.

WordCounter.java: Driver Program

```
package com.app;

import java.io.IOException;
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.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));
   }
}
```

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 has two main parts:
  1. Namespace (manages files and directories)
  2. Blocks storage service (handles data storage and replication)
- Key features of HDFS 2:
  - Horizontal scalability (can easily grow by adding more nodes)
  - High availability (system stays reliable and available)
- Uses multiple independent NameNodes that don't need to coordinate with each other.
- DataNodes store blocks and are shared among all NameNodes.
- High availability is achieved using **Passive Standby NameNode**:
  - Active-Passive NameNode setup where passive node takes over automatically if active node fails.
  - Namespace edits are saved to shared storage.
  - Passive NameNode reads from shared storage and keeps metadata updated.
  - If active NameNode fails, passive becomes active and continues writing to shared storage.

5.12.4 Hadoop 2 YARN:

Figure 5.26 Active and Passive NameNode interaction.

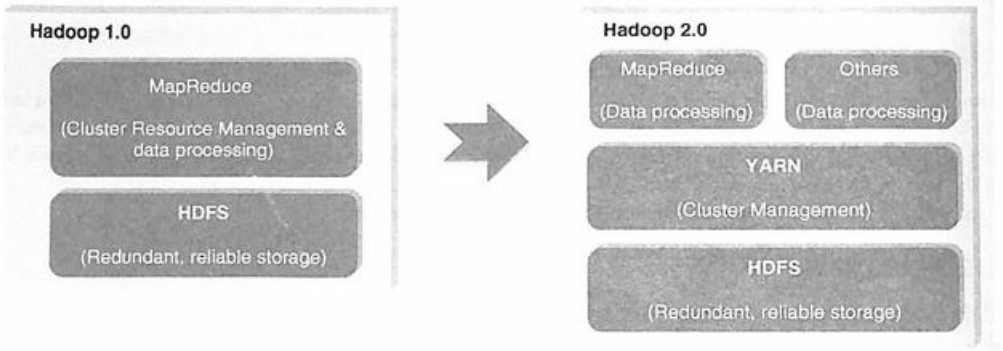


Figure 5.27 Hadoop 1.x versus Hadoop 2.x.

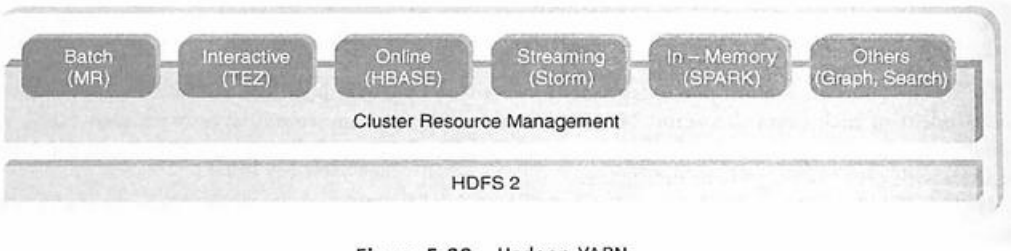


Figure 5.28 Hadoop YARN.

5.12.4.1. YARN's fundamental idea

- YARN splits the JobTracker's responsibilities into separate daemons for resource management and job scheduling/monitoring.

Key Components:

1. **Global ResourceManager:**
    - Distributes resources among running applications.
    - Has two parts:
        - **Scheduler:** Decides resource allocation; does not monitor application status.
        - **ApplicationManager:** Accepts job submissions, negotiates resources, restarts ApplicationMaster if it fails.
2. **NodeManager:**

- Runs on each slave machine.
- Launches application containers.
- Monitors resources like memory, CPU, disk, network.
- Reports usage to ResourceManager.

3. **Per-application ApplicationMaster:**
   - Manages resource negotiation with ResourceManager for its application.
   - Works with NodeManager to execute and monitor tasks.

5.1.2.4.2 Basic Concepts of YARN

Basic Concepts

Application:

1. A job submitted to the YARN framework.
2. Example: A MapReduce job.

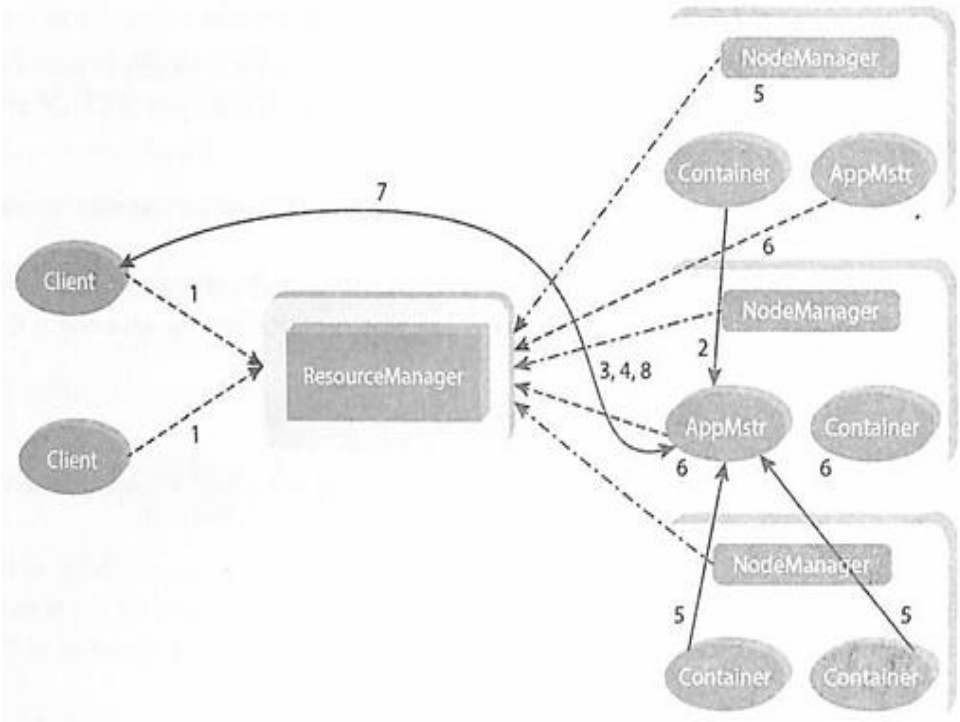
Container:

1. The basic unit of resource allocation.
2. Provides fine-grained resource allocation across multiple resource types such as memory, CPU, disk, and network.
   - Example:
     - container_0 = 2GB memory, 1 CPU
     - container_1 = 1GB memory, 6 CPUs
3. Replaces the fixed map/reduce slots in earlier Hadoop versions.

YARN Architecture (Key Steps):

1. A client submits an application specifying how to launch the application-specific **ApplicationMaster**.
2. The **ResourceManager** launches the ApplicationMaster by assigning a container.
3. The ApplicationMaster registers with the ResourceManager during startup, enabling direct queries about resource status.

4. The ApplicationMaster negotiates resource containers via a resource-request protocol during the job execution.
5. **Launching Containers**:
6. The **ApplicationMaster** launches containers by sending container launch instructions to the **NodeManager** after successful resource allocation.
7. **Execution by NodeManager**:
8. The **NodeManager** runs the application code and reports progress, status, etc., back to the **ApplicationMaster** using a specific protocol.
9. **Client Communication**:
10. The **client** that submitted the job talks directly to the **ApplicationMaster** to receive job updates (e.g., status, progress), also via a protocol.
11. **Completion & Shutdown**:
12. After job completion, the **ApplicationMaster** deregisters from the **ResourceManager** and shuts down, freeing up its container for reuse.



Introduction to MapReduce Programming

Introduction

- **MapReduce jobs** are split into **map tasks** and **reduce tasks**, executed on a Hadoop cluster.
- Each task processes a small portion of the data, allowing Hadoop to **distribute the load**.
- Input data is stored in **HDFS (Hadoop Distributed File System)**.

Map tasks handle:

1. RecordReader – loads data.
2. Mapper – processes data.
3. Combiner – optional local aggregation.
4. Partitioner – splits data for reducers.

Reduce tasks handle:

1. Shuffle – move data to reducers.
2. Sort – organize keys.
3. Reducer – combine values.
4. Output Format – write results.

- Output from map tasks = **intermediate keys and values**, sent to reducers.
- Hadoop runs map tasks **where the data is stored** (DataNode) to **reduce network usage** and **save bandwidth**.

8.2 Mapper

- **Mapper** turns input key–value pairs into intermediate key–value pairs.
- The process has four main parts:
    1. **RecordReader** – Converts raw byte data into record-oriented form, giving key–value pairs to the mapper.
    2. **Map** – Processes the input pairs to produce zero or more intermediate pairs.
    3. **Combiner** (optional) – Aggregates intermediate data locally to save bandwidth and disk space.

4. **Partitioner** – Splits intermediate pairs into shards, sending each shard to the correct reducer (same keys go to the same reducer).

- Partitioned data is stored locally and then fetched by the reducer.

8.3 Reducer

Reducer's job: Take intermediate values with the same key and reduce them into a smaller set.

- It works in **three phases**:

1. **Shuffle and Sort** – Collects outputs from all partitioners, downloads them to the reducer's local machine, and sorts them by key so similar items are grouped together.
2. **Reduce** – Processes each group of values for a key (e.g., sum, filter, aggregate, or combine data), producing zero or more output key–value pairs.
3. **Output Format** – Writes the final key–value pairs to a file (default separator is a tab).

- **Figure 8.1** shows how data flows through Mapper → Combiner → Partitioner → Shuffle & Sort → Reducer.

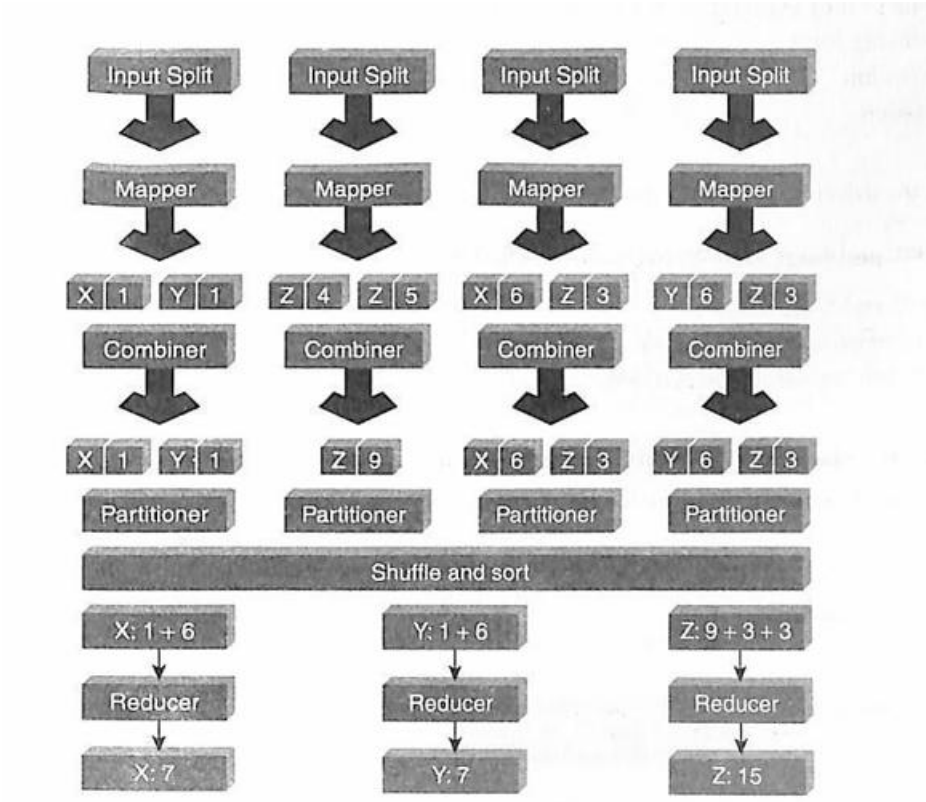


Figure 8.1 The chores of Mapper, Combiner, Partitioner, and Reducer.

8.4 Combiner :

- **Purpose**: Optimization technique in MapReduce to reduce data transfer between Mapper and Reducer.
- **Difference from Reducer**:
    1. Combiner output = **intermediate data** → sent to Reducer.
    2. Reducer output = **final result** → written to disk.
- **Use case example**: Counting occurrences of similar words in a file before sending to Reducer.
- **How it works**:
    - Set the combiner class in the driver program (often same as Reducer class).
    - Runs locally on Mapper output to aggregate data early.
- **Sample code**:

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

FileInputFormat.addInputPath(job, new Path("/mapreducedemos/lines.txt"));

FileOutputFormat.setOutputPath(job, new Path("/mapreducedemos/output/wordcount/"));
```

- **Execution**:
  Run the job with Hadoop command:

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

- **Output location**: Stored in part-r-00000 file by default in the output directory.
- **Example output** (word count):

```
python-repl

CopyEdit

hadoop   2

hive     2

pig      1

session  3

...
```

8.5 Practitioner

- Purpose:
  - Runs after the Map phase and before the Reduce phase.
  - Decides how intermediate key–value pairs are divided among reducers.
  - Default method: Hash partitioner.
- Number of partitions = number of reducers.
- Example task: Count occurrences of similar words and partition them based on the first alphabet.
- How it works:
  - Custom WordCountPartitioner checks the first letter of the word.
  - Assigns a partition number (1–26 for A–Z, plus default).
  - Ensures that all words starting with the same letter go to the same reducer.

Code: **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;
    }
}
```

Driver Program (relevant part)

```
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/"));
```

Input Data (`lines.txt`)

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

Example Output (part-r-00008 for 'H')

```
Hadoop      2
Hive        2
```

8.6 Searching:

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

In this case, the program scans an input file (e.g., student.csv) and finds lines containing the given keyword ("Jack"), then outputs the matching record along with its filename and position in the file.

1. Driver Program — 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();
        conf.set("keyword", "Jack"); // Keyword to search

        Job job = Job.getInstance(conf, "Keyword Search");
        job.setJarByClass(WordSearcher.class);

        job.setMapperClass(WordSearchMapper.class);
        job.setReducerClass(WordSearchReducer.class);


        job.setOutputKeyClass(Text.class);

        job.setOutputValueClass(Text.class);


        job.setInputFormatClass(TextInputFormat.class);

        job.setOutputFormatClass(TextOutputFormat.class);
```

```
        job.setNumReduceTasks(1);

        FileInputFormat.setInputPaths(job, new Path("/mapreduce/student.csv"));
        FileOutputFormat.setOutputPath(job, new Path("/mapreduce/output/search"));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

2. Mapper — 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.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;

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        Configuration configuration = context.getConfiguration();
        keyword = configuration.get("keyword");
    }


    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        FileSplit fileSplit = (FileSplit) context.getInputSplit();
        String fileName = fileSplit.getPath().getName();
        pos++;

        if (value.toString().contains(keyword)) {
            context.write(value, new Text(fileName + "," + pos));
```

```
        }
    }
}
```

4. Reducer — 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> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
InterruptedException {
        for (Text value : values) {
            context.write(key, value);
        }
    }
}
```

4. Input File (student.csv)

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

5. Output File (/mapreduce/output/search/part-r-00000)

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

8.7 Sorting

1. Understanding the Problem

You have a CSV file (student.csv) with the following data:

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

We need to **sort the data by the student name** (the second column).

2. How the MapReduce Works

1. **Mapper** — reads each line, splits it into fields, and **emits** the student name as the key, the rest of the data as the value.
2. **Reducer** — receives keys (names) in **sorted order** automatically from Hadoop, then writes out the corresponding data.

3. Corrected Code

SortStudNames.java

```
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(",");
            if (token.length == 3) {
                context.write(new Text(token[1]), new Text(token[0] + "," + token[1] + "," + token[2]));
            }
        }
    }

    public static class SortReducer extends Reducer<Text, Text, NullWritable, Text> {
        protected 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 Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Sort by Student Name");

        job.setJarByClass(SortStudNames.class);
        job.setMapperClass(SortMapper.class);
        job.setReducerClass(SortReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

4. How to Run

hadoop jar SortStudNames.jar SortStudNames /mapreduce/student.csv /mapreduce/output/sorted

5. Expected Output

Hadoop sorts keys (names) alphabetically, so output will be:

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

6. Summary

- **Input**: CSV with IDs, Names, Scores.
- **Processing**: Sort by name (second field).
- **Output**: Alphabetically sorted list of students.
- **Key Trick**: Use the student name as the Mapper output key so Hadoop handles sorting automaticall

8.8 COMPRESSION

- **Why compress MapReduce output?**
  1. Saves storage space.
  2. Speeds up data transfer across the network.
- **How to enable compression in the Driver program:**

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

  - GzipCodec → compression algorithm.
  - CompressionCodec → interface for compression/decompression.
- **Effect:** Output files are compressed in .gz format.