**UNIT –IV**

**Why Hadoop?**

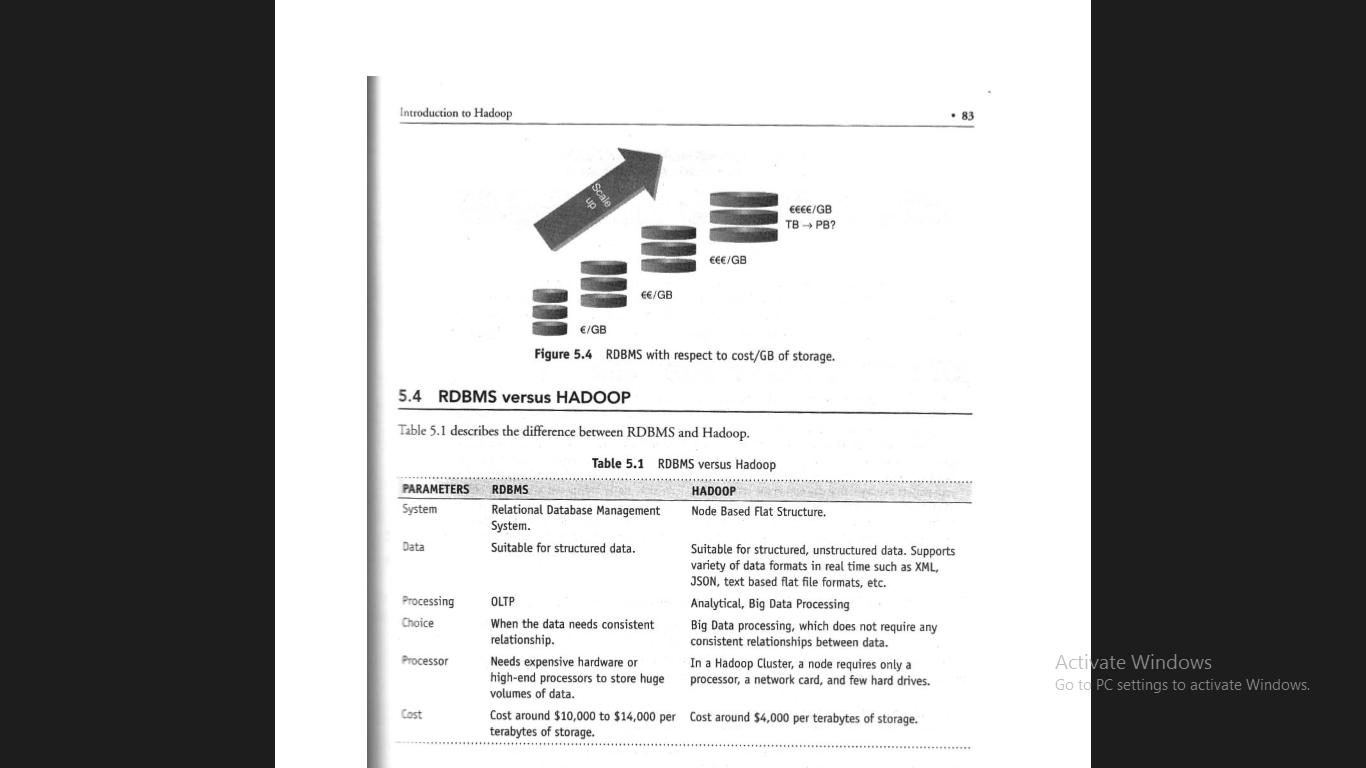
1. **Key Considerations of Hadoop** 
   * **Low Cost**: Hadoop is open-source and runs on commodity hardware, making it cost-effective.
   * **Computing Power**: Hadoop’s distributed computing model allows it to process large datasets quickly by utilizing multiple nodes.
   * **Scalability**: Nodes can be added to the Hadoop cluster without much administration effort.
   * **Storage Flexibility**: Unlike traditional databases, Hadoop can handle unstructured data like images, videos, and free-form text.
   * **Inherent Data Protection**: Hadoop handles hardware failures by replicating data across multiple nodes and automatically redistributing jobs if a node fails.

**Hadoop Framework** : Hadoop uses clusters—groups of machines working together—to distribute and replicate data across nodes, ensuring failover protection. This architecture supports:

* + **Data distribution across nodes** with redundancy.
  + **Parallel data processing** on locally available resources.
  + **Automatic failover** to handle node failures.

**Why Not RDBMS?**

* Traditional RDBMS systems are not designed for storing and processing large-scale unstructured data like images and videos.
* RDBMS scalability comes with high costs and limitations in distributed data processing.
* Hadoop is more suited for big data analytics and machine learning tasks compared to RDBMS, which struggles to manage the volume, variety, and velocity of big data.



**Hadoop Overview**

Hadoop is an open-source software framework designed to store and process massive amounts of data in a distributed manner using clusters of commodity hardware. Its primary tasks are:

1. **Massive data storage**: Storing large datasets efficiently.
2. **Faster data processing**: Parallel processing of data for quicker results.

**Key Aspects of Hadoop**

The key aspects of Hadoop are described in the figure:

1. **Open-source software**: Hadoop is free to download, use, and modify. Users can contribute to its development.
2. **Framework**: It provides all necessary components to develop and execute data processing tasks, including tools and libraries.
3. **Distributed**: Data is divided and stored across multiple computers (nodes). Computation is performed in parallel across these nodes for efficiency and reliability.
4. **Massive storage**: Hadoop is designed to handle enormous volumes of data across a network of low-cost hardware, ensuring scalability and redundancy.
5. **Faster processing**: Hadoop enables parallel processing of data, ensuring faster response times for large-scale data queries.

**Hadoop Distributors**

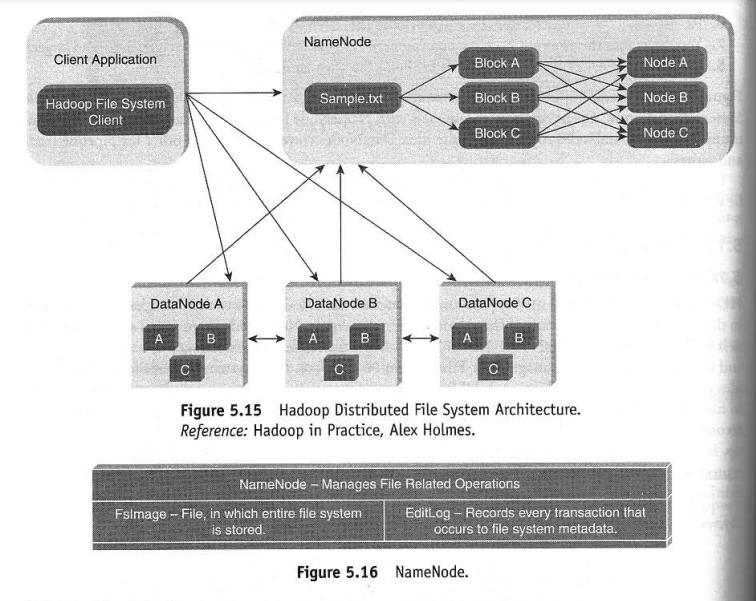
* Lists major companies that provide Hadoop distributions:
  + **Cloudera** (CDH 4.0, CDH 5.0)
  + **Hortonworks** (HDP 1.0, HDP 2.0)
  + **MAPR** (M3, M5, M8)
  + **Apache Hadoop** (Hadoop 1.0, Hadoop 2.0)
* These companies offer Apache Hadoop with commercial support and additional tools/utilities.

**HDFS - Hadoop Distributed File System**

* **Key Points of HDFS:**
  1. **Storage component** of Hadoop.
  2. **Distributed File System** (stores data across multiple machines).
  3. **Modeled after Google File System (GFS)**.
  4. **Optimized for high throughput** (uses large block sizes and moves computation closer to data).
  5. **Supports file replication** (enhances fault tolerance in software and hardware failures).
  6. **Automatically re-replicates data blocks** if a node fails.
  7. **Efficient for large files** (gigabytes and above).
  8. **Works on top of native file systems**.

**HDFS Daemons**

* **NameNode**:
  + It is responsible for managing the **File System Namespace** in HDFS.
  + The NameNode breaks large files into smaller pieces called **blocks**.
  + A **rack ID** is used to identify the location of DataNodes within the rack.
  + The NameNode manages operations like:
    - **Read** and **write** requests.
    - **File creation**, **deletion**, and block replication.
  + **File System Namespace Management**:
    - The namespace contains a mapping of file names to blocks and file properties.
    - This mapping is stored in a file called **FsImage**, which represents the current state of the file system.
  + **EditLog**:
    - The EditLog is a transaction log that records every file system operation (e.g., file creation or deletion).
    - On restart, the NameNode reads the EditLog and applies all transactions to the FsImage to ensure consistency.
  + **FsImage Update Process**:
    - Once the NameNode has applied all transactions to the FsImage, it flushes the updated FsImage to disk and truncates the old EditLog to free up space.
  + **Single Point**:
    - There is **one NameNode per cluster**.



**Hadoop Distributed File System**

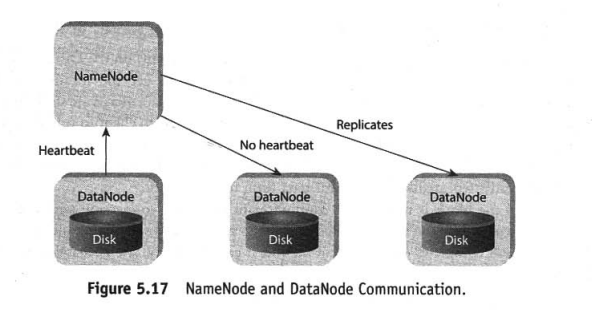
* **Client Application**: The client interacts with the HDFS through the Hadoop File System Client, which communicates with the NameNode to manage file operations.
* **NameNode**: Manages the metadata of the file system (e.g., the locations of file blocks, replication status).
  + In this case, a file (Sample.txt) is broken into **three blocks (A, B, and C)**.
  + Each block is replicated across multiple **DataNodes** for fault tolerance and data availability.
* **DataNodes**: Store the actual data blocks. The replication factor ensures that multiple copies of each block are distributed across different nodes for redundancy.
  + For example:
    - Block A is stored in **DataNode A, B, and C**.
    - Block B and Block C are also replicated across these three DataNodes.

**DataNode**

* Multiple DataNodes exist within a Hadoop cluster.
* **Communication**:
  + DataNodes send **heartbeat messages** to the NameNode at regular intervals to confirm they are active and functional.
  + If the NameNode does not receive a heartbeat from a DataNode within a specified time, it assumes that the DataNode has failed.
  + The NameNode then replicates the blocks from the failed DataNode to other active DataNodes to maintain the replication factor and prevent data loss.

**NameNode Operations**

* **FsImage**: Stores the entire file system’s metadata, including the block-to-file mappings and file properties.
* **EditLog**: A log of every file system operation (e.g., file creations, deletions). It helps in recovering and updating the FsImage when the system restarts.

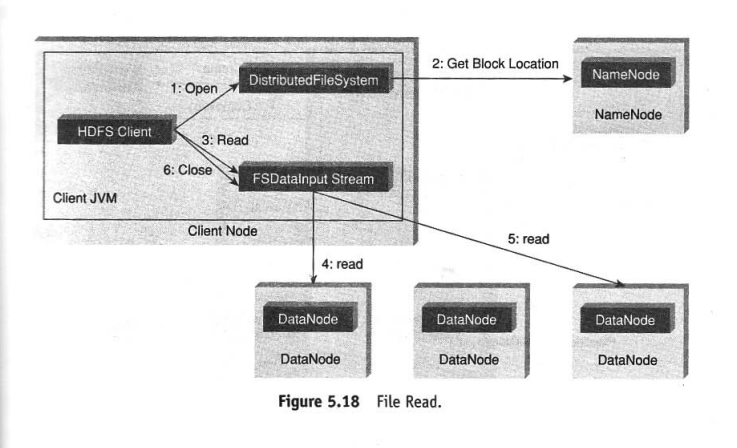


**Secondary NameNode**

* The **Secondary NameNode** is not a real-time backup of the NameNode. Instead, it periodically takes snapshots of the HDFS metadata.
* These snapshots help reduce the burden on the NameNode and assist in recovery scenarios.

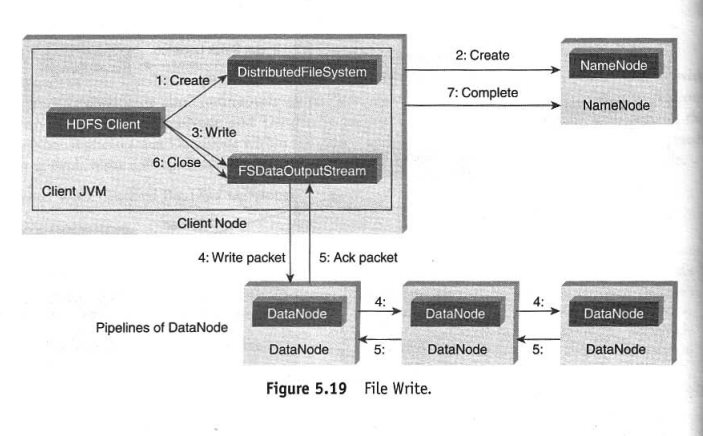
**Anatomy of File Read**

1. The **HDFS Client** opens a file using the DistributedFileSystem.
2. It communicates with the **NameNode** to get the location of data blocks.
3. An **FSDataInputStream** is created to read the file’s data from the DataNodes.
4. The client reads the data sequentially from the DataNodes, moving to the next block as necessary.
5. After all blocks are read, the client closes the FSDataInputStream.



**Anatomy of File Write (Figure 5.19)**

1. The client calls create() on the DistributedFileSystem to create a new file.
2. An **RPC call** is made to the NameNode to register the file.
3. The client receives an **FSDataOutputStream** to write the data.
4. Data is split into packets and sent to the DataNodes in a pipeline fashion.
5. Each DataNode sends an acknowledgment back to the client after receiving the data.
6. Once all blocks are written, the file creation is completed.

**How MapReduce Works**

* The **input dataset** is split into **independent chunks**, which are processed by **map tasks**.
* Each **map task** runs **independently and in parallel**, producing **intermediate data**.
* This intermediate data is stored on the **local disk** of the server.
* The **output from map tasks is shuffled and sorted** based on **keys**.
* The sorted output is sent to **reduce tasks**, which **combine the outputs** from multiple mappers.
* The final **reduced output** is then stored in the file system.

**Features of MapReduce**

* **Task Scheduling & Monitoring**: The framework handles **scheduling**, **monitoring**, and **re-executing failed tasks**.

**Components of MapReduce Architecture**

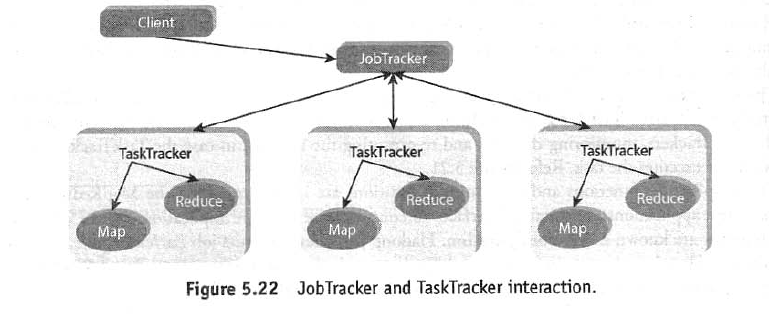
* **JobTracker (Master Node)**
  + There is **one JobTracker per cluster**.
  + Responsible for **scheduling** tasks to worker nodes (**TaskTrackers**).
  + **Monitors tasks** and **re-executes** them if needed.
* **TaskTracker (Slave Node)**
  + Each cluster node has **one TaskTracker**.
  + **Executes tasks** assigned by the JobTracker.

**---------------JobTracker-------------**

* **Acts as a master daemon responsible for managing job execution in Hadoop.**
* **When you submit code to the cluster, JobTracker decides which task to assign to which node.**
* **Monitors running tasks and re-schedules failed tasks to a different node after a predefined number of retries.**
* **There is only one JobTracker per Hadoop cluster.**

**------------------TaskTracker-------------**

* **Acts as a slave daemon, responsible for executing tasks assigned by JobTracker.**
* **Each slave node has a single TaskTracker, which spawns multiple JVMs to execute parallel map and reduce tasks.**
* **TaskTracker sends a heartbeat message to JobTracker. If JobTracker does not receive a heartbeat, it assumes the TaskTracker has failed and reschedules the task to another node.**
* **When a job is submitted, JobTracker partitions and assigns MapReduce tasks to different TaskTrackers.**



**Hadoop Configuration**

Hadoop configuration involves setting up and tuning various parameters to optimize its performance for specific workloads. Here’s a structured guide to configuring Hadoop:

**1. Core Configuration Files**

Hadoop's configuration is managed through XML files located in the $HADOOP\_HOME/etc/hadoop/ directory.

**1.1. core-site.xml**

Defines common settings, such as the NameNode address and Hadoop's I/O behavior.

<configuration>

<property>

<name>fs.defaultFS</name>

<value>hdfs://namenode:9000</value>

<description>HDFS default file system URI</description>

</property>

<property>

<name>hadoop.tmp.dir</name>

<value>/tmp/hadoop</value>

<description>Temporary directory for Hadoop files</description>

</property>

</configuration>

**1.2. hdfs-site.xml**

Configures the HDFS NameNode, DataNode, and replication factors.

<configuration>

<property>

<name>dfs.replication</name>

<value>3</value>

<description>Number of copies of each data block</description>

</property>

<property>

<name>dfs.namenode.name.dir</name>

<value>file:///var/hadoop/hdfs/namenode</value>

<description>Path for storing NameNode metadata</description>

</property>

<property>

<name>dfs.datanode.data.dir</name>

<value>file:///var/hadoop/hdfs/datanode</value>

<description>Path for storing DataNode blocks</description>

</property>

</configuration>

**1.3. mapred-site.xml**

Configures MapReduce job execution.

<configuration>

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value>

<description>Uses YARN for resource management</description>

</property>

</configuration>

**1.4. yarn-site.xml**

Configures YARN, the cluster resource manager.

<configuration>

<property>

<name>yarn.resourcemanager.hostname</name>

<value>resourcemanager</value>

<description>ResourceManager hostname</description>

</property>

<property>

<name>yarn.nodemanager.aux-services</name>

<value>mapreduce\_shuffle</value>

<description>Enables MapReduce shuffle service</description>

</property>

</configuration>

**2. Setting Up Hadoop Cluster**

**2.1. Format NameNode**

hdfsnamenode -format

**2.2. Start Hadoop Services**

start-dfs.sh # Start HDFS (NameNode, DataNode, SecondaryNameNode)

start-yarn.sh # Start YARN (ResourceManager, NodeManager)

**2.3. Verify Hadoop Services**

jps # Check running Hadoop processes

**2.4. Access Hadoop Web UI**

* NameNode UI: http://namenode:9870
* ResourceManager UI: http://resourcemanager:8088

**HBase: NoSQL, Distributed, Column-Oriented Database on Hadoop**

**HBase** is an **open-source, distributed, NoSQL database** that runs on top of **Hadoop’s HDFS**. It is **column-oriented** and optimized for handling **real-time, random read/write access** to large datasets. Unlike traditional relational databases, HBase does not use tables with fixed schemas; instead, it stores data in a flexible, column-family format.

**Why Was HBase Developed?**

Traditional **RDBMS (Relational Database Management Systems)** struggle with **scalability and real-time access** when dealing with large volumes of data. Hadoop's **HDFS (Hadoop Distributed File System)** allows distributed storage but does not provide a way to efficiently retrieve small pieces of data in real-time. HBase was developed to address these issues by offering:

* **Fast, scalable, and real-time access to structured data** stored in Hadoop.
* **Random access read/write operations**, unlike HDFS, which is optimized for batch processing.

HBase is **modeled after Google’s BigTable**, a distributed storage system designed for managing structured data across thousands of machines.

**Key Features of HBase**

1. **NoSQL (Schema-less) Storage**
   * Unlike RDBMS, which uses fixed schemas, HBase allows **flexible, column-family-based storage**.
   * This makes it ideal for semi-structured or unstructured big data.
2. **Distributed & Scalable**
   * Runs on multiple machines in a Hadoop cluster.
   * Scales horizontally by adding more nodes.
3. **Column-Oriented Storage**
   * Data is stored in **columns instead of rows**, improving read performance for large datasets.
4. **Real-Time, Random Read/Write Access**
   * Unlike batch-processing systems like Hive, HBase supports **low-latency** operations.
5. **Strong Consistency**
   * Unlike some NoSQL databases, HBase provides **consistent reads and writes**.
6. **Automatic Sharding (Region Splitting)**
   * Tables automatically **split into regions** and distribute across nodes.
7. **Fault Tolerance**
   * Built on **HDFS**, meaning data is **replicated across multiple nodes** for reliability.

**HBase vs. Traditional Databases**

| **Feature** | **HBase (NoSQL)** | **Traditional RDBMS** |
| --- | --- | --- |
| Data Model | Column-oriented, NoSQL | Row-oriented, SQL-based |
| Schema | Dynamic, schema-less | Fixed schema |
| Scalability | Horizontal scaling (adds more nodes) | Vertical scaling (adds more CPU, RAM) |
| Transaction Support | No full ACID support | ACID transactions supported |
| Best Use Case | Real-time big data applications | Small to medium structured datasets |

**Example: Traditional Row-Oriented vs. HBase Column-Oriented Storage**

**Row-Oriented Storage (RDBMS - MySQL, PostgreSQL, etc.)**

In a traditional relational database, data is stored **row by row**, which is optimized for transactional workloads (OLTP - Online Transaction Processing).

| **ID** | **Name** | **Age** | **City** | **Phone** |
| --- | --- | --- | --- | --- |
| 101 | Alice | 25 | New York | 123-456-7890 |
| 102 | Bob | 30 | London | 987-654-3210 |
| 103 | Charlie | 35 | Berlin | 555-123-4567 |

* **Problem:** If we need to query only the "Age" column for analysis, we still need to **scan entire rows**.
* **Row-Based Query:**

SELECT Age FROM Users;

* + This retrieves the **entire row and extracts "Age"**, leading to unnecessary I/O operations.

**Column-Oriented Storage (HBase)**

HBase stores data **by column families**, allowing efficient access to specific columns.

**Table: users**

| **Row Key** | **Column Family: Personal (Name, Age)** | **Column Family: Contact (City, Phone)** |
| --- | --- | --- |
| 101 | Name: Alice, Age: 25 | City: New York, Phone: 123-456-7890 |
| 102 | Name: Bob, Age: 30 | City: London, Phone: 987-654-3210 |
| 103 | Name: Charlie, Age: 35 | City: Berlin, Phone: 555-123-4567 |

* Data is stored **column by column**, making it **faster to retrieve specific columns**.

**Advantages:**

* If we need **only "Age"**, HBase can **scan only the "Age" column** instead of the entire row.
* Queries fetch **only necessary columns**, reducing disk I/O.

**Column-Based Query in HBase**

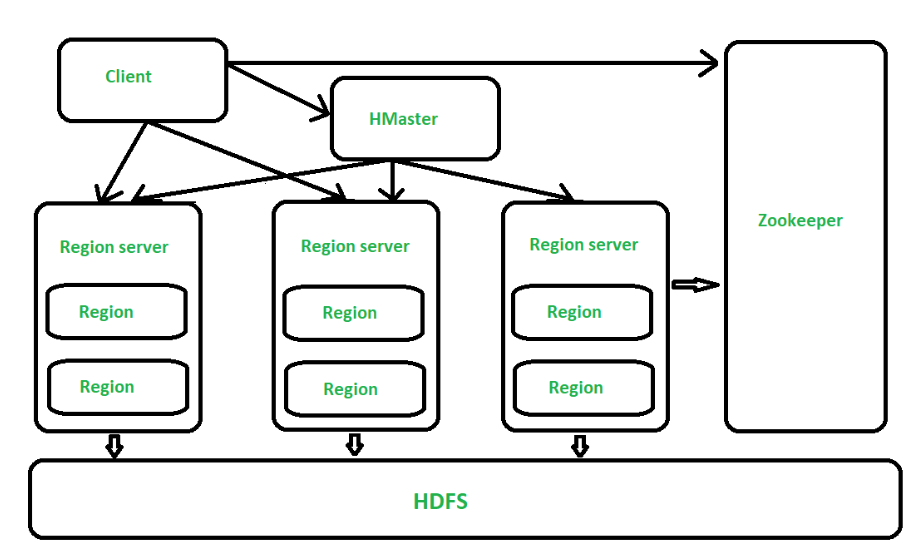
Using HBase shell:

scan 'users', {COLUMNS => ['Personal:Age']}

* **Efficient:** Fetches only the "Age" column without scanning unnecessary data.

**Architecture of HBase**

**HBase architecture** has 3 main components: HMaster, Region Server, Zookeeper.



**HMaster (Master Server in HBase)**

The **HMaster** is the **main process that manages the HBase cluster**. It is responsible for:  
- **Assigning Regions** to Region Servers.  
- Performing **DDL (Data Definition Language) operations** such as creating, deleting, and modifying tables.  
- **Monitoring all Region Servers** in the cluster.  
- Running **background threads** for tasks like load balancing and failover handling.  
- Handling**automatic region splitting** when a region grows beyond a predefined size.

In a **distributed environment**, multiple HMaster instances can be set up, but only **one is active at a time**. If the active HMaster fails, **ZooKeeper** promotes a standby HMaster.

**Region Server – The Processing Unit of HBase**

In **HBase**, tables are **horizontally partitioned** into smaller units called **Regions**, each containing a **range of row keys**. The **Region Server** is responsible for managing these **Regions** and executing **read/write operations** on them.

**Key Functions of a Region Server:**

* + **Handles read and write operations** on a set of **Region**
  + **Manages multiple Regions**, where each Region holds a subset of a table.
  + Runs on **HDFS DataNodes** in the Hadoop cluster, ensuring **distributed storage**.
  + Stores data in **HFiles (on HDFS) for persistent storage**.
  + **Automatically splits** Regions when they grow beyond a set threshold.

**Region Structure in HBase:**

A **Region** consists of:

* **Row Keys** (range-based partitioning of data).
* **Column Families**, where data is stored logically.
* **HFiles**, which store actual data in HDFS.
* **Default Region Size:256 MB**. If a Region grows beyond this, HBase **splits it into two smaller Regions** to maintain load balancing.

**Zookeeper**   
-It is like a coordinator in HBase.

-It provides services like maintaining configuration information, naming, providing distributed synchronization, server failure notification etc.

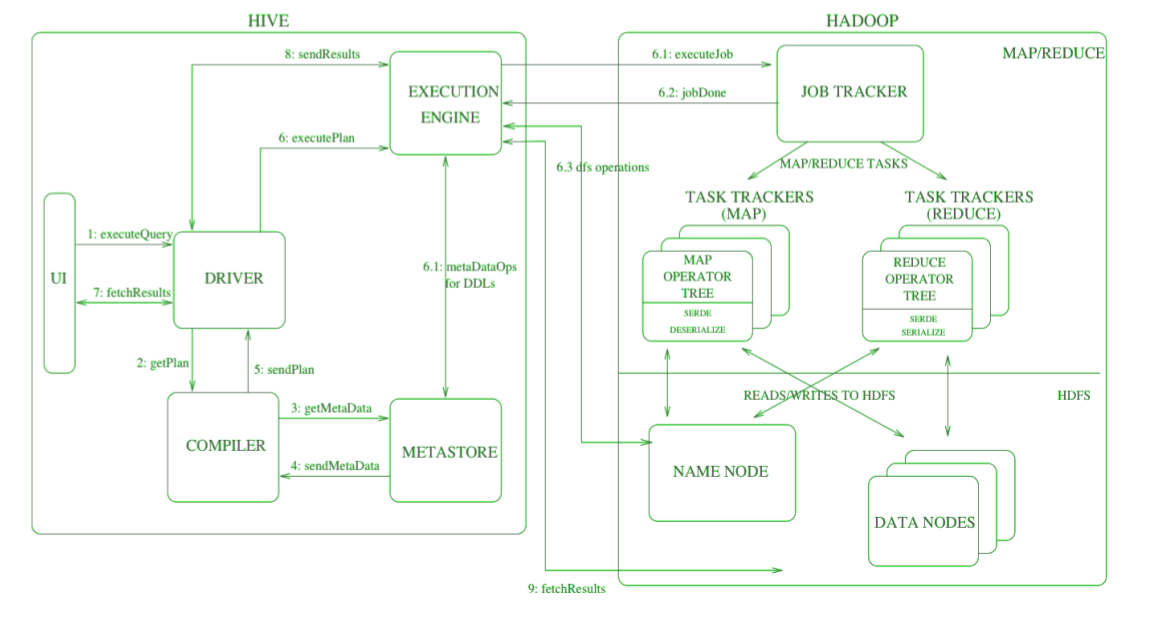
-Clients communicate with region servers via zookeeper.

HIVE

**Apache Hive** is a **data warehouse infrastructure** built on top of **Hadoop** that allows users to process and analyze large datasets using **SQL-like queries**. Instead of writing complex **MapReduce programs**, users can write **Hive Query Language (HiveQL)**, which is then converted into **MapReduce, Tez, or Spark jobs** for execution.

**Key Features of Hive**

* **SQL-Like Interface** – Makes working with Hadoop easier for analysts and data engineers.
* **Best for Batch Processing & Analytics** – Ideal for summarization, ad-hoc queries, and reporting.
* **Scalable & Fault-Tolerant** – Works on Hadoop’s distributed storage (HDFS).
* **Schema on Read** – You can query structured & semi-structured data without enforcing a schema before loading.
* **Supports Various Execution Engines** – Runs queries using **MapReduce (default), Tez, or Spark**.



**Main Components of Hive Architecture**

**Metastore (Stores Schema & Metadata)**

* Stores **table schemas, partitions, column types**, and other metadata.
* Uses **MySQL or PostgreSQL** as a backend database.
* Enables **query optimization** and efficient storage handling.

**Driver (Compiles & Optimizes HiveQL Queries)**

* Parses **HiveQL queries** and checks for syntax errors.
* Communicates with **Metastore** to retrieve table information.
* Sends the optimized query plan to the **Execution Engine**.

**Execution Engine (Converts Queries into MapReduce/Tez/Spark Jobs)**

* Translates HiveQL queries into **MapReduce, Tez, or Spark** jobs.
* Breaks the query into **stages** and optimizes execution flow.
* **data retrieval from HDFS** and result computation.

**HDFS (Hadoop Distributed File System) – Storage Layer**

* Stores raw **structured and semi-structured data**.
* Uses a **distributed architecture**, meaning data is **split across multiple nodes** for **scalability and fault tolerance**.
* Tables in Hive are typically stored as **CSV, JSON**.

**How a Hive Query is Executed?**

**User submits a HiveQL query** (e.g., SELECT \* FROM sales WHERE region='Asia';).  
**The Driver parses the query** and checks syntax & metadata (via Metastore).  
**Execution Engine optimizes** the query and generates a **query plan**.  
**MapReduce, Tez, or Spark jobs** are launched to process the data from HDFS.  
**Results are computed and returned** to the user.

**Apache Pig**

Apache Pig is a **high-level scripting language** used for **processing large datasets** in **Hadoop**. It provides an easy way to write **complex data transformations** using a **procedural scripting language** called **Pig Latin**.

**Key Features of Pig:**

* **High-Level Language** – Uses **Pig Latin**, which is simpler than writing raw **MapReduce** programs.
* **Automatically Converts Scripts into MapReduce Jobs** – Optimizes execution to run efficiently on Hadoop.
* **Supports Both Structured & Unstructured Data** – Works with text, JSON, CSV, and binary formats like Avro.
* **Example Pig Latin Script:**

-- Load data from HDFS

sales\_data = LOAD 'hdfs://path/to/sales.csv' USING PigStorage(',')

AS (id:int, item:chararray, amount:float);

-- Filter records where amount > 100

high\_value\_sales = FILTER sales\_data BY amount > 100;

-- Group sales by item

grouped\_sales = GROUP high\_value\_sales BY item;

-- Calculate total sales per item

total\_sales = FOREACH grouped\_sales GENERATE group, SUM(high\_value\_sales.amount);

-- Store results back in HDFS

STORE total\_sales INTO 'hdfs://path/to/output' USING PigStorage(',');

**Pig - Architecture**



**Hive vs Pig**

Both **Apache Hive and Apache Pig** simplify **Big Data processing** but serve different use cases. Here’s a **comparison**:

| **Feature** | **Hive** | **Pig** |
| --- | --- | --- |
| **Type** | SQL-like query engine | Scripting-based data flow |
| **Best for** | Querying structured data | Complex data transformations |
| **Language** | HiveQL (Declarative) | Pig Latin (Procedural) |
| **Ease of Use** | Easier for SQL users | Easier for programmers |
| **Execution** | Converts queries into MR/Tez/Spark jobs | Converts scripts into MapReduce jobs |
| **Use Case** | Data warehousing & analytics | ETL, log processing, data cleaning |

**Introduction to Data Analytics with R**

**Why R for Data Analytics?**

R is a powerful open-source programming language that is widely used in **data analytics, statistical computing, and machine learning**. It provides a **comprehensive environment** for handling, visualizing, and analyzing large datasets efficiently. Below are some of the key reasons why R is a popular choice for data analytics:

1. **Open-source & Free**
   * R is freely available, making it accessible to researchers, data scientists, and businesses.
   * Large and active community support provides numerous free libraries and resources.
2. **Statistical Computing Capabilities**
   * R is designed for advanced **statistical analysis and data modeling**.
   * Provides inbuilt functions for regression, hypothesis testing, time series analysis, and more.
3. **Rich Ecosystem of Machine Learning Libraries**
   * R supports a variety of machine learning techniques through powerful libraries such as:
     + **caret** – Unified framework for ML models
     + **randomForest** – Random Forest for classification and regression
     + **xgboost** – Gradient boosting algorithm for predictive modeling
4. **Visualization Capabilities**
   * R excels in **data visualization and storytelling**, making it easy to explore and communicate insights.
   * Popular visualization libraries include:
     + **ggplot2** – Advanced data visualization
     + **lattice** – Multi-panel statistical graphics
     + **plotly** – Interactive graphs and dashboards
5. **Integration with Big Data Technologies**
   * R can **handle large datasets** and integrate with Big Data frameworks such as:
     + **Hadoop** – Parallel computing with R using the RHadoop package
     + **Spark** – Distributed ML and big data processing via SparkR
     + **BigR** – Enables R to work with **Big Data** stored in HDFS

**Key Steps in Data Analytics with R**

To perform data analytics in R, a structured workflow is typically followed. Below are the key steps:

**Step 1: Data Collection**

The first step in data analytics is **importing data** from different sources into R. Common data sources include:

* **CSV files** → read.csv("data.csv")
* **Excel files** → readxl::read\_excel("data.xlsx")
* **Databases** (MySQL, PostgreSQL, MongoDB) → DBI and RMySQL
* **Web scraping** (APIs, JSON, XML) → httr, rvest

**Step 2: Data Preprocessing**

Before analysis, raw data needs to be **cleaned and transformed**:

* **Handling Missing Values**
  + Remove missing data → na.omit(dataset)
  + Impute missing values → mean(dataset$column, na.rm = TRUE)
* **Data Transformation**
  + Convert categorical variables → as.factor(dataset$column)
  + Normalize numerical data → scale(dataset$column)

**Step 3: Exploratory Data Analysis (EDA)**

EDA helps in understanding the **distribution, patterns, and relationships** in data.

* **Descriptive Statistics**
  + Summary of data → summary(dataset)
  + Mean, median, standard deviation → mean(), sd(), quantile()
* **Data Visualization**
  + **Univariate Analysis** → Histograms, box plots (ggplot2)
  + **Bivariate Analysis** → Scatter plots, correlation heatmaps

**Step 4: Model Building (Supervised & Unsupervised Learning)**

Depending on the problem type, different machine learning techniques are applied:

* **Supervised Learning (Labeled Data)**
  + Regression: **Linear Regression, Random Forest Regression**
  + Classification: **Logistic Regression, Decision Trees, SVM**
* **Unsupervised Learning (Unlabeled Data)**
  + Clustering: **k-Means, Hierarchical Clustering, DBSCAN**
  + Dimensionality Reduction: **PCA**

**Step 5: Model Evaluation**

After training, models are **evaluated** using various performance metrics:

* **Regression Metrics**
  + **RMSE (Root Mean Squared Error)** → Measures error in prediction
  + **R² (R-Squared)** → Measures model accuracy
* **Classification Metrics**
  + **Accuracy** → (Correct Predictions / Total Predictions)
  + **Precision & Recall** → Performance of classification models
  + **ROC Curve & AUC Score** → pROC package for model evaluation

**Step 6: Deployment & Interpretation of Results**

Once the model is validated, it is **deployed for real-world use**.

* **Deploying as an API** using **Plumber**
* **Deploying on web applications** with **Shiny**
* **Interpreting results** and generating reports using **R Markdown**

**Introduction to Collaborative Filtering**

Collaborative Filtering recommends items by analyzing past interactions between **users and items**.

**How does it work?**

* **User-based filtering:** "People similar to you liked these items."
* **Item-based filtering:** "If you liked this item, you may like similar items."
* **Hybrid Filtering:** Combines **both** user-based and item-based filtering.

**Example Use Cases**

* **E-commerce:** Suggesting products based on past purchases.
* **Streaming Platforms:** Recommending movies based on viewing history.
* **Online Learning:** Suggesting courses based on user activity

**2. Types of Collaborative Filtering**

**2.1. User-Based Collaborative Filtering**

Finds **similar users** and recommends items liked by similar users.

* Example: If **User A** and **User B** have similar movie preferences, then **User A** will get recommendations based on **User B's** likes.

**Mathematical Approach:**

* Measures similarity using **Cosine Similarity** or **Pearson Correlation**.

**2.2. Item-Based Collaborative Filtering**

Finds **similar items** and recommends them to users who liked similar items.

**Example:** If many users who purchased **"iPhone 13"** also bought **"AirPods Pro"**, then a user who buys **"iPhone 13"** will get a recommendation for **"AirPods Pro"** since these items are frequently bought together.

**2.3. Hybrid Filtering**

Combines **User-based and Item-based filtering** for better recommendations.

* Used by Netflix, YouTube, and Amazon.
* New users with no history.