





Big-Data-Topics

Table of Contents

- [Big-Data-Topics](#)
-  [Big Data](#)
 - [1. What is Data](#)
 - [2. Sources of Data](#)
 - [3. Types of Data](#)
 - [4. What is Big Data](#)
 - [5. Why We Need Big Data](#)
 - [6. 4 Vs of Big Data](#)
 - [7. Big Data Sources](#)
 - [8. Challenges in Handling Big Data](#)
 - [9. Approaches to Handling Big Data](#)
 -  [10. Technologies for Big Data Processing](#)
 - [Storage Solutions](#)
 - [Processing Frameworks](#)
 - [Streaming Processing](#)
 - [Data Warehousing](#)
 - [Data Visualization](#)
 - [11. Data Life Cycle Stages \(Example: Food Delivery App\)](#)
-  [Cluster Computing](#)
 - [1. What is Computation?](#)
 - [2. What is Cluster Computing?](#)
 - [3. Components of Cluster Computing](#)
 - [4. Types of Cluster Computing](#)
 - [5. Distributed Cluster Computing](#)
 - [6. Row-Oriented Data Stores](#)
 - [7. Column oriented data stores](#)
 -  [8. Why do we have these row and column types](#)

- 1. Row-Oriented Databases
- 2. Column-Oriented Databases
- 9. Difference between cluster computing and distributed computing
- ☐ Hadoop
 - 1. What is hadoop
 - 2. Why is hadoop used?
 - 3. Key components in Hadoop
 - 4. Hadoop Architecture
 - 5. Hadoop Vendors
 - 6. Hadoop Cloud providers
 - 7. Difference between Hadoop and traditional Databases
 - 8. What are HDFS Daemons
 - 9. What is Heartbeat?
 - 10. What is a block?
 - 11. Replication
 - 12. Balancing
 - 13. HDFS Commands
 - 14. Common file formats in HDFS
 - 15. YARN
 - 16. YARN Architecture
 - 17. YARN application workflow
 - 18. What is mapreduce
 - ☐ 19. What are the stages in mapreduce?
 - Splitting
 - Mapping
 - Shuffling
 - Reducing
 - 20. Types of input format in mapreduce
 - 21. Map Reduce Architecture
 - ☐ 22. Reducer Side Join
 - 1. Input: You have two datasets, say:
 - 2. Map Phase:
 - 3. Shuffle & Sort:


- 4. Reduce Phase:
 - When to Use:
- 23. Map Side Join
- ▼ 24. MapReduce Design Patterns
 - Common Patterns:
- ▼ Hive
 - 1. What is Hive?
 - 2. Features of Hive
 - ▼ 3. Hive Architecture
 - 1. Hive Client
 - 2. Hive Services
 - 3. Hive Driver
 - 4. Metastore
 - 5. Compiler
 - 6. Execution Engine
 - ▼ 4. Hive Flow
 - 1. Getting the Request
 - 2. Driver Sends to Compiler
 - 3. Compiler Plans Execution
 - 4. Driver Sends Info to Execution Engine
 - 5. Execution Engine Runs MapReduce Jobs
 - 6. Role of Execution Engine
 - 5. Job execution flow in Hive
 - 6. Where is Metastore stored in the case of Hive
 - 7. Hive Data types
 - ▼ 8. Hive Tables
 - Managed Table
 - External Table
 - 9. Partitioning
 - 10. Static vs Dynamic partitioning
 - 11. Bucketing
 - 12. Hive Data model
 - 13. Partition vs Bucketing

- 14. Hive indexing
- ▼ SQOOP
 - 1. What is SQOOP?
 - ▼ 2. SQOOP Architecture
 - 1. Relational Database Systems (RDBMS)
 - 2. SQOOP Client
 - 3. HDFS (Hadoop Distributed File System)
 - 3. SQOOP Workflow
 - 4. SQOOP Commands
- ▼ FLUME
 - ▼ 1. What is Flume?
 - Why use flume?
 - 2. Flume Architecture
 - ▼ 3. Flume Configuration
 - Source
 - Channel
 - Sink
 - Workflow:
- ▼ PIG
 - ▼ 1. What is PIG?
 - Why use PIG?
 - ▼ 2. PIG Architecture
 - Workflow:
 - ▼ 3. PIG Commands and usecases
 - ▼ Common Commands:
 - Load Data
 - Filter Data
 - Store Data
 - Use Cases:
 - 4. Sqoop,Flume,Pig
- ▼ HBase
 - 1. What is HBase?
 - ▼ 2. HBase Architecture

- HMaster
- RegionServer
- Zookeeper
- HDFS
- 3. How is data stored in HBase
- 4. Features of HBase
- 5. Use cases of HBase
- 6. HDFS vs HBase
- ☐ KAFKA
 - 1. What is KAFKA
 - ☐ 2. KAFKA architecture
 - Producer
 - Topic
 - Broker
 - Partition
 - Consumer
 - Consumer Group
 - ZooKeeper
 - 3. Features of KAFKA
 - ☐ 4. KAFKA Workflow
 - 1. Producer sends data to Kafka
 - 2. Kafka Topic receives and organizes the data
 - 3. Brokers store and manage messages
 - 4. Consumers read the messages
 - 5. ZooKeeper
 - 6. Consumers process and act on messages
 - 7. Retention and Replay
 - 5. Summary
- ☐ Spark
 - 1. What is spark?
 - ☐ 2. Why Spark?
 - Hadoop is slower
 - Spark is faster

- How does Spark do it?
- Spark supports many programming languages
- ▼ 3. Spark Implementation
 - 1. Standalone Mode
 - 2. With Other Cluster Managers
- 4. Hadoop vs Spark
- 5. What Hadoop gives spark
- ▼ 6. Spark Components
 - 1. Spark Core
 - 2. Spark SQL
 - 3. Spark Streaming
 - 4. MLlib
 - 5. GraphX
- ▼ 7. Spark architecture
 - 1. Driver Program
 - 2. SparkContext
 - 3. Cluster Manager
 - 4. Worker Nodes
 - 5. Executors
- ▼ 8. Spark flow – How a Spark Program Runs
 - 1. User writes Spark code and submits it
 - 2. Driver creates SparkContext
 - 3. SparkContext connects to the Cluster Manager
 - 4. Cluster Manager allocates worker nodes and starts Executors
 - 5. Driver sends tasks to Executors
 - 6. Executors perform the tasks and return results
 - 7. Driver combines the results and gives the output to the user
- 9. Executors
- ▼ 10. Driver Side
 - Key components on the driver side:
- ▼ 11. Executor Side
 - Key components on the executor side:
- ▼ 12. RDD – Resilient Distributed Datasets

- What is it?
- Key Properties of RDDs:
- Why are RDDs useful?
- 13. RDD Features
- 14. DAG (Directed Acyclic Graph)
- ▼ 15. Transformation
 - Example:
 - Types of Transformations:
- ▼ Pyspark
 - PySpark Architecture
 - 1. Create SparkSession and Load CSV Data
 - 2. Replace Nulls in "city" Column with "unknown"
 - 3. Delete Rows Where "jobtitle" Is Null
 - 4. Replace Null Salary with Mean
 - 5. Who Has Higher Average Salary – Male or Female?
 - 6. Inspecting the DataFrame
 - 7. Drop All Rows Containing Any Null
 - 8. Filter Out Rows With Null in a Specific Column
 - 9. Create a New Column in Uppercase (city)
 - 10. Drop Duplicates
 - 11. Rename a Column
 - 12. Filter Rows Based on Condition
 - 13. Filter Names Ending with Pattern
 - 14. Filter Names Starting with "Dr."
 - 15. Filter IDs Between 1 and 5
 - 16. Substring Example
 - 17. Multiple Filters (AND condition)
 - 18. Add a Calculated Column (bonus 10% of salary)
 - 19. Group By & Aggregation
 - 20. Output Result to File
 - ▼ Examples
 - 1. Clean Missing Data
 - 2. Average Salary per Department

- 3. Rename & Filter Female Employees
- 4. Add Bonus & Save to File
- 5. Remove Duplicates and Filter by Pattern
- 6. Complex Filter with Aggregation
-  Scala
 - 1. What is Scala and why is it used in Big Data?
 - 2. What's the difference between val and var?
 - 3. What is a case class?
 - 4. What is a higher-order function?
 - 5. What's the difference between map, flatMap, and filter?
 - 6. What is Option in Scala?
 - 7. What are immutable collections?
 - 8. How is Scala both object-oriented and functional?
 - 9. What is a trait?
 - 10. Why do we use Scala with Spark instead of Java?
 - 11. What is match function in scala
 - 12. For loop in scala
- Big Data Component Comparison Table

Big Data

1. What is Data

Data refers to raw, unorganized facts, figures, or information that can be processed and analyzed to gain insights or knowledge.

It can be any set of characters, such as text, words, numbers, pictures, or sounds, that can be translated into a usable form.



2. Sources of Data

- **Human-generated data:**
Surveys and questionnaires — direct collection of opinions, preferences, and facts.
- **Machine-generated data:**

- **Sensors:** IoT devices, environmental, medical, industrial sensors collecting real-time data (temperature, location, etc.).
- **Web logs/Server logs:** Records of website visits, user interactions, errors, server activity.
- **Business and organizational data:**
Sales and marketing data like campaign performance, lead generation, sales figures.
- **Open and publicly available data:**
Public APIs from platforms such as Twitter, Wikipedia, etc.



3. Types of Data

- **Structured Data:**
Highly organized, stored in tables with rows and columns in relational databases. Each column has a clearly defined type (numbers, dates, text).
- **Unstructured Data:**
Does not follow a predefined format or organization, such as text, images, video. More challenging to process and analyze with traditional methods but makes up a large part of modern data.



4. What is Big Data

Big data refers to extremely large and diverse datasets that are so voluminous, complex, and rapidly growing that traditional data processing tools and methods cannot handle them effectively.



5. Why We Need Big Data

- **Deeper Insights & Better Decisions:**
Enables predictive maintenance, fraud detection, resource management, leading to cost reductions.
- **Enhanced Customer Experience**





6. 4 Vs of Big Data

1. **Volume** — The quantity or scale of data generated and stored.
2. **Velocity** — The speed at which data is generated, collected, and processed (real-time or near real-time).
3. **Variety** — Diversity of data types and sources (numbers, text, images, audio, video).
4. **Veracity** — Quality, accuracy, trustworthiness, and consistency of data.



7. Big Data Sources

1. **Users (Human-Generated Data):**
Data from human interaction, rich in context and intent, often unstructured.
2. **Applications (Software-Generated Data):**
Data from apps reflecting business processes, user interactions, and operational performance.
3. **Systems (Machine-Generated/Infrastructure Data):**
Data from IT infrastructure, networks, operational systems without direct human input. Critical for performance, security, stability monitoring.
4. **Sensors (IoT and Physical World Data):**
Devices that detect physical inputs and transmit data, forming the backbone of IoT.



8. Challenges in Handling Big Data

1. **Storage and Scalability**
Storing ever-growing, massive volumes of diverse data efficiently and affordably.
2. **Processing Speed (Velocity)**
The speed at which data arrives often outpaces the ability of traditional systems to process it quickly enough for timely decision-making.
3. **Data Quality and Integration (Variety & Veracity)**
Integrating data from diverse, often incompatible sources, and then cleaning it to remove inconsistencies, errors, and duplicates is complex and time-consuming.
4. **Security and Compliance**
Safeguarding enormous volumes of data from cyberattacks and ensuring strict

adherence to data privacy laws across various jurisdictions is a constant challenge.

5. **Data Visualization**

Transforming vast, intricate datasets into digestible visual formats that communicate meaningful insights to diverse audiences is a major design and technical challenge.



9. **Approaches to Handling Big Data**

Big data means huge amounts of data—too big for a normal computer to handle. So, we need smart ways to store, process, and manage it. Here are the key approaches:

1. **Distributed Computing**

Instead of using one big computer, we use many smaller computers (nodes) working together.

Example: Apache Hadoop, Spark

2. **Processing**

- **Batch Processing**

Data is collected over time and processed in chunks.

Example: Daily sales reports

- **Stream Processing**

Data is processed in real-time as it arrives.

Example: Credit card fraud detection

3. **Data Partitioning and Sharding**

Feature	Partitioning	Sharding
Meaning	Splitting a large dataset into smaller parts	A special type of partitioning stored on separate servers
Context	Used in Spark, Hive, etc.	Used in distributed databases / NoSQL
Storage	Can be on one or more servers	Each shard is usually on a different server
Goal	Improve query performance, speed	Handle scalability and system load
Example	Split data by year	Users 1–1000 on Server A, 1001–2000 on Server B

4. **ETL vs ELT**

Feature	ETL (Extract → Transform → Load)	ELT (Extract → Load → Transform)
When transformed	Before loading	After loading
System type	Traditional systems	Modern cloud systems
Speed and control	Slower but more controlled	Faster, scales with big data



10. Technologies for Big Data Processing

Storage Solutions

- Google Cloud Storage
- HDFS (Hadoop Distributed File System)
- Amazon S3

Processing Frameworks

- Apache Spark
- Hadoop MapReduce
- Google BigQuery

Streaming Processing

- Apache Kafka
- Google Dataflow

Data Warehousing

- Google BigQuery
- Snowflake
- Amazon Redshift

Data Visualization

- Tableau
- PowerBI

- Google Looker Studio



11. Data Life Cycle Stages (Example: Food Delivery App)

1. Generation

A user places an order → data is generated.

2. Collection

App collects user location, payment, restaurant, etc.

3. Processing

System checks availability, assigns delivery person.

4. Storage

Data is saved in databases.

5. Management

Remove duplicates, secure storage, keep it updated.

6. Analysis

Analyze: popular items, peak hours, etc.

7. Visualization

Dashboards show trends with graphs/charts.

8. Interpretation

Managers decide on offers, staffing, etc.



Cluster Computing

1. What is Computation?

Work done by computers to solve problems or analyze data.

2. What is Cluster Computing?

Multiple computers (nodes) work together like one big computer, sharing processing and memory.



3. Components of Cluster Computing

1. **Nodes** – Individual computers in the cluster.
2. **Network** – High-speed connection between nodes.
3. **Cluster Manager** – Assigns tasks and handles failures.
4. **Storage** – Shared/distributed storage for data and results.



4. Types of Cluster Computing

- **High Performance Clusters (HPC)**
Speed-oriented. Used in simulations, research, etc.
- **High Availability Clusters (HA)**
Reliability-focused. Ensures no downtime.
- **Load Balancing Clusters**
Spreads work evenly to prevent overload.



5. Distributed Cluster Computing

A type of cluster where nodes work independently on parts of a big problem and collaborate.



6. Row-Oriented Data Stores

Store data row-by-row like in traditional SQL:

ID	Name	Age	
1	Alice	25	→ stored as: 1, Alice, 25
2	Bob	30	→ stored as: 2, Bob, 30



7. Column oriented data stores

Column-oriented data stores save data column by column instead of row by row.

ID	Name	Age
1	Alice	25
2	Bob	30

Stored as:

ID: 1, 2

Name: Alice, Bob

Age: 25, 30



8. Why do we have these row and column types

Different types of data and queries need different ways to store data efficiently. So, databases use row-based or column-based storage depending on the use case:

1. Row-Oriented Databases

- Designed for transactional tasks where you usually need to access or update all data in a row at once. -- - Examples: Banking systems, online stores — where you read/write full records like user info or orders.

2. Column-Oriented Databases

- Optimized for analytical tasks where you mostly need data from a few columns across many rows.
- Great for reports, statistics, and data analysis where reading entire rows would be wasteful.

Row stores: Fast for writing and reading full records.

Column stores: Fast for reading and aggregating specific columns on large datasets.



9. Difference between cluster computing and distributed computing

Feature	Cluster Computing	Distributed Computing
Node Location	Nodes are close together (same place)	Nodes are spread out geographically
Communication Speed	Very fast between nodes	Usually slower communication
Control	Centralized management	Decentralized control
Node Dependency	Nodes depend on each other	Nodes work independently
Scalability	Limited scalability	Highly scalable



Hadoop

1. What is hadoop

- Hadoop is an open-source framework that helps you store and process huge amounts of data.
- It works by spreading data and tasks across many computers (a distributed system) so big data can be handled efficiently.



2. Why is hadoop used?

- Scalability
 - Can process huge data by spreading the work across many computers.
- Cost Effectiveness
 - Runs on cheap, regular hardware instead of expensive machines.
- Fault Tolerance
 - Keeps working even if some computers fail by making copies of data.
- Flexibility
 - Handles all types of data — structured (tables), semi-structured (JSON), and unstructured (videos, texts).

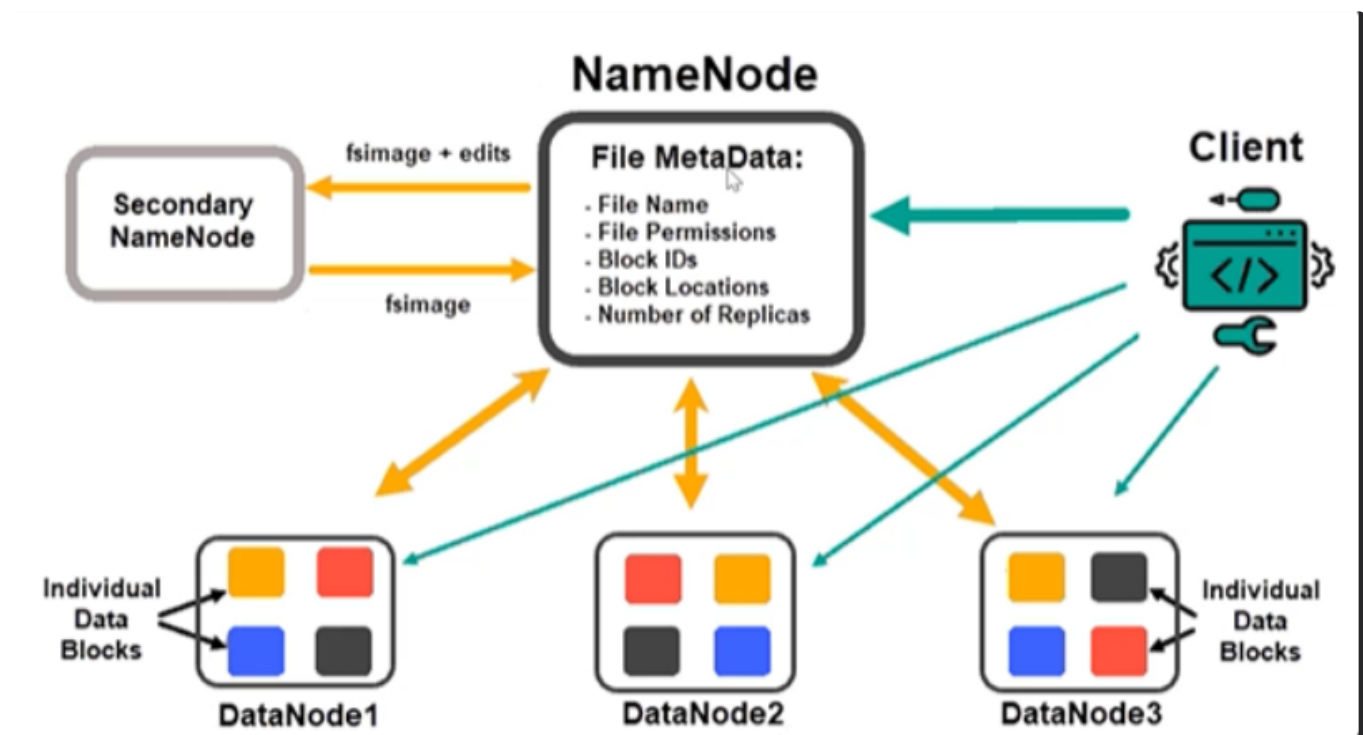


3. Key components in Hadoop

- HDFS (Hadoop Distributed File System)
 - Stores big data by splitting it across many computers.
- MapReduce
 - Processes data by breaking tasks into small parts, running them in parallel, then combining results.
- YARN (Yet Another Resource Negotiator)
 - Manages resources and schedules tasks across the cluster to keep everything running smoothly.



4. Hadoop Architecture



- Hadoop uses a master-slave design.
- The NameNode is the master that manages metadata (information about files).
- DataNodes are the slaves that actually store the file data.
- When a client sends a file
 - The NameNode creates and stores metadata (like file location, blocks info).
 - The file's actual data is split and stored across multiple DataNodes.

- The NameNode does not store the actual data, only the metadata.
- If the NameNode fails, a Secondary NameNode steps in to take over.



5. Hadoop Vendors

These are companies or organizations that provide Hadoop distributions, support, or services:

- **Apache** — The original open-source project that created Hadoop.
- **Cloudera** — Offers enterprise Hadoop solutions and tools.
- **Hortonworks** — Provided Hadoop-based products (now merged with Cloudera).
- **IBM** — Provides Hadoop services and integrates it with their own platforms.



6. Hadoop Cloud providers

Cloud companies offer managed Hadoop services to make it easier to run big data jobs without setting up your own cluster:

- Amazon – Elastic MapReduce (EMR)
- Microsoft Azure – HDInsight
- Google Cloud – Dataproc



7. Difference between Hadoop and traditional Databases

Feature	Hadoop	Traditional Databases
Data Types	Handles unstructured, semi-, and structured data	Handles structured data only
Scaling	Horizontal scaling (add more computers)	Vertical scaling (upgrade hardware)
Processing Style	Batch processing for large datasets	Real-time processing
Cost	Usually low cost (uses cheap hardware)	Usually high cost





8. What are HDFS Daemons

- HDFS Daemons are background processes that run on Hadoop's nodes to manage storage and data flow. The main ones are:
- **NameNode:** Manages metadata and file system namespace (the “master” daemon).
- **DataNode:** Stores the actual data blocks on the nodes (the “worker” daemon).
- **Secondary NameNode:** Helps the NameNode by periodically saving metadata backups.



9. What is Heartbeat?

- A heartbeat is a regular signal sent by each DataNode to the NameNode.
- It tells the NameNode that the DataNode is alive and working.
- If the NameNode stops receiving heartbeats from a DataNode, it assumes that node is dead or unreachable.



10. What is a block?

- In Hadoop, a block is a fixed-size chunk of a file.
- Instead of storing the whole file in one place, Hadoop splits large files into smaller blocks (usually 128 MB or 64 MB each).
- These blocks are then distributed and stored across multiple DataNodes.



11. Replication

- Hadoop makes multiple copies of each data block to keep it safe.
- The default replication factor is 3, meaning each block is stored in 3 different places.
- These copies are stored on different DataNodes to keep data safe and available
- This ensures data is not lost if one node fails.



12. Balancing

- When a DataNode crashes, all the blocks stored on it become unavailable, causing under-replication (fewer copies than needed).
- The NameNode detects this and asks other DataNodes that have copies of those blocks to create new replicas.
- This process restores the required number of replicas and keeps the data evenly distributed across the cluster.



13. HDFS Commands

- `hadoop version`
 - Shows the Hadoop version installed.
- `hadoop fs -mkdir /path/dir_name`
 - Creates a new directory in HDFS.
- `hadoop fs -ls /path`
 - Lists files and folders in the specified HDFS path.
- `hadoop fs -put <localsource> <hdfs dest>`
 - Uploads a file from your local system to HDFS.
- `hdfs dfs -cat /hadoop/test`
 - Displays the content of a file stored in HDFS.
- `hadoop fs -cp source target`
 - Copies files within HDFS from source to target location.
- `hadoop fs -copyFromLocal source_local destination_hdfs`
 - Copies a file from local filesystem to HDFS.
- `hadoop fs -copyToLocal source_hdfs destination_local`
 - Copies a file from HDFS to local filesystem.



14. Common file formats in HDFS

- Text file
 - Plain text, easy to read but not efficient for large data.

- Sequence file format
 - Binary format that stores key-value pairs, good for intermediate data.
- Avro
 - Compact, fast, supports schema evolution (changing structure over time).
- Parquet
 - Columnar storage format, great for fast queries and compression.
- ORC
 - Another efficient columnar format, optimized for Hive.
- RCFile
 - Early columnar storage format used in Hadoop ecosystems.
- JSON
 - Text format for semi-structured data, human-readable and flexible.

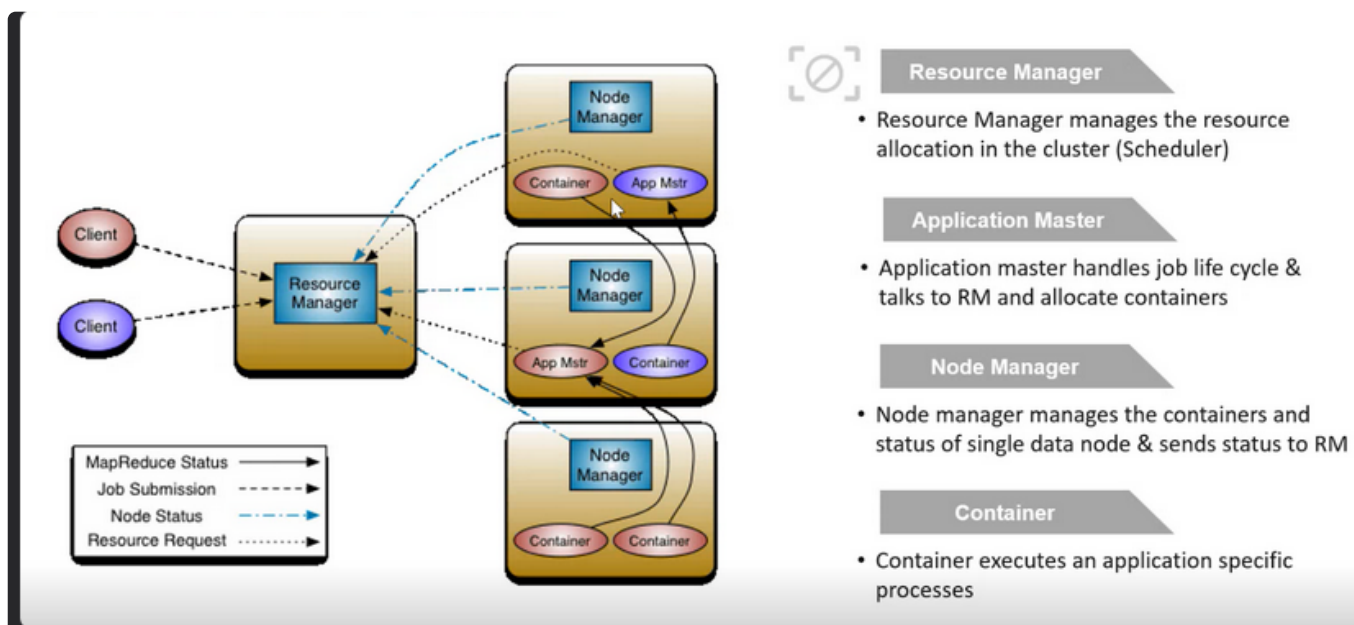


15. YARN

- YARN is responsible for allocating resources (like CPU and memory) to different applications running on the Hadoop cluster.
- It schedules tasks so that the cluster's work gets done efficiently.
- YARN also manages and monitors workloads, making sure jobs run smoothly and resources are used well.

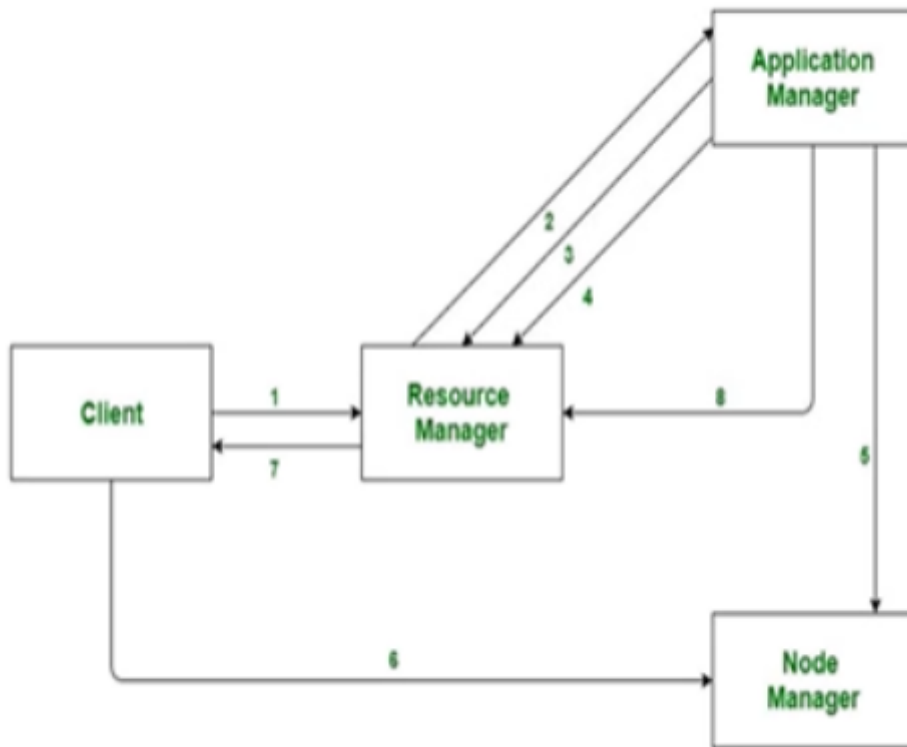


16. YARN Architecture



- **Resource Manager**
 - Acts like a manager that decides how resources (CPU, memory) are allocated across the cluster.
- **Node Manager**
 - Works like a team lead (TL) on each node, managing resources and communicating with the Resource Manager.
- **Application Master**
 - Manages the lifecycle of applications, requests resources from the Resource Manager, and assigns work (containers) to Node Managers.
- **Container**
 - The place where the actual processing happens — it's a unit of computation with allocated resources.

17. YARN application workflow



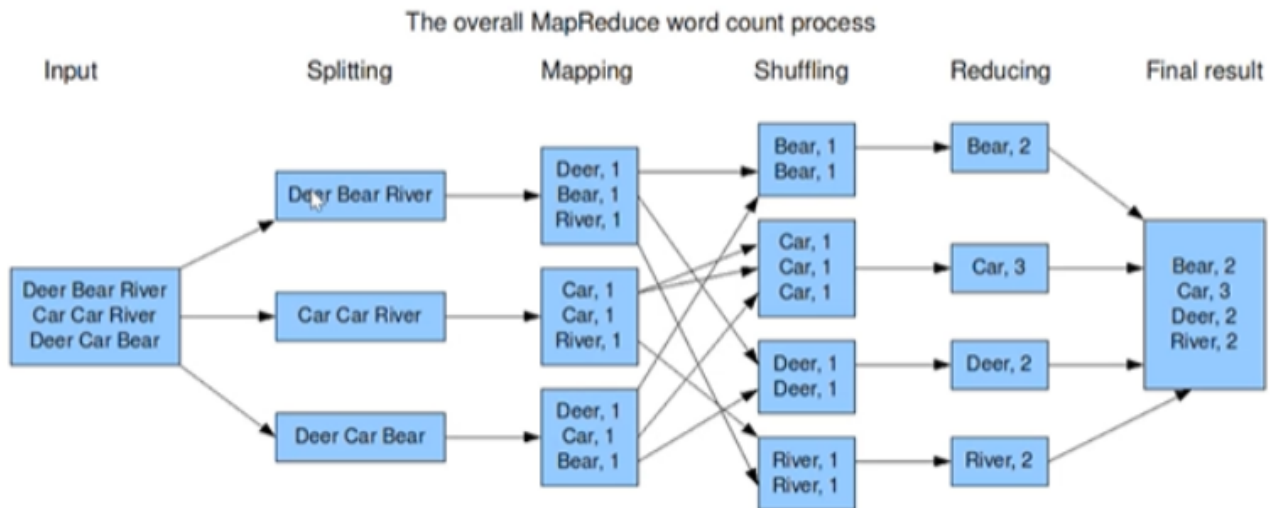
- **1. Client submits an application**
 - You give a task (like a data query) to the system.
 - This task needs resources to run.
- **2. Resource Manager allocates a container**
 - A small portion of resources is assigned to **start the Application Master**.
- **3. Application Master registers with Resource Manager**
 - It introduces itself and says, “I’m handling this application.”
- **4. Application Master requests containers**
 - It asks the Resource Manager for more resources to actually run the job.
- **5. Application Master tells Node Manager to launch containers**
 - It instructs nodes to start work using the resources provided.
- **6. Code runs in containers**
 - The actual job (like processing data) happens here.
- **7. Client checks status**
 - You can ask the Resource Manager or Application Master to see if the job is done.
- **8. Application Master finishes and unregisters**
 - Once the job is complete, it signs off and releases the resources.

18. What is mapreduce

- MapReduce is a framework that lets you write programs to process huge amounts of data in parallel using a cluster of computers.
- Mapper (Map Job)
 - Reads the input data and converts it into key-value pairs.
 - Example: It might take a document and output words with a count.
- Reducer
 - Takes the output from multiple mappers.
 - Aggregates or summarizes the data into a smaller, final result.
 - Example: It adds up all word counts across documents.
- Simple flow
 - Input → Mapper → key-value pairs
 - key-value pairs → Reducer → Final result



19. What are the stages in mapreduce?



Splitting

- Big data is divided into smaller chunks (blocks).
- These chunks are sent to different mapper nodes for parallel processing.

Mapping

- Each mapper processes its data and produces key-value pairs.
- Example: ("Dear", 1), ("Bear", 1), ("River", 1)

Shuffling

- The output from mappers is sorted and grouped by key.
- All values for the same key are brought together before reducing.
 - Eg:
 - Bear is grouped together
 - ("Bear", 1)
 - ("Bear", 1)
 - Car is grouped together
 - ("Car", 1)
 - ("Car", 1)
 - ("Car", 1)

Reducing

- The reducer takes the grouped key-value pairs and aggregates or processes them.
- Final result is generated here (e.g., total word count).
- Eg:
 - Bear has the following entries
 - ("Bear", 1)
 - ("Bear", 1)
 - After reducing, we get the "Bear" word count
 - ("Bear", 2)



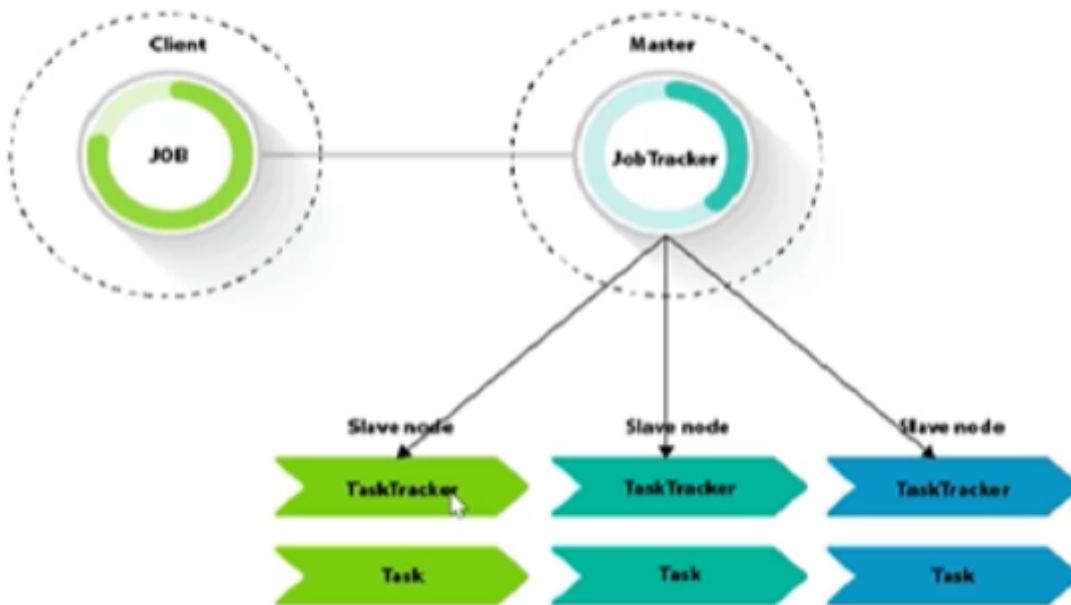
20. Types of input format in mapreduce

- **FileInputFormat**
 - The base class for all input formats.
 - Splits data into chunks (input splits) to feed mappers.
- **TextInputFormat** (*default*)
 - Reads data **line by line**.

- Each line is given a key (line offset) and value (line content).
- **KeyValueTextInputFormat**
 - Treats each line as a **key-value pair**, split by a tab or a custom separator.
- **SequenceFileInputFormat**
 - Reads **binary files** made up of key-value pairs (used for performance).
- **SequenceFileAsTextInputFormat**
 - Reads sequence files, but **converts binary keys and values to text**.
- **NLineInputFormat**
 - Sends **N lines to each mapper** (instead of one line per record).
 - Useful for controlling mapper load.



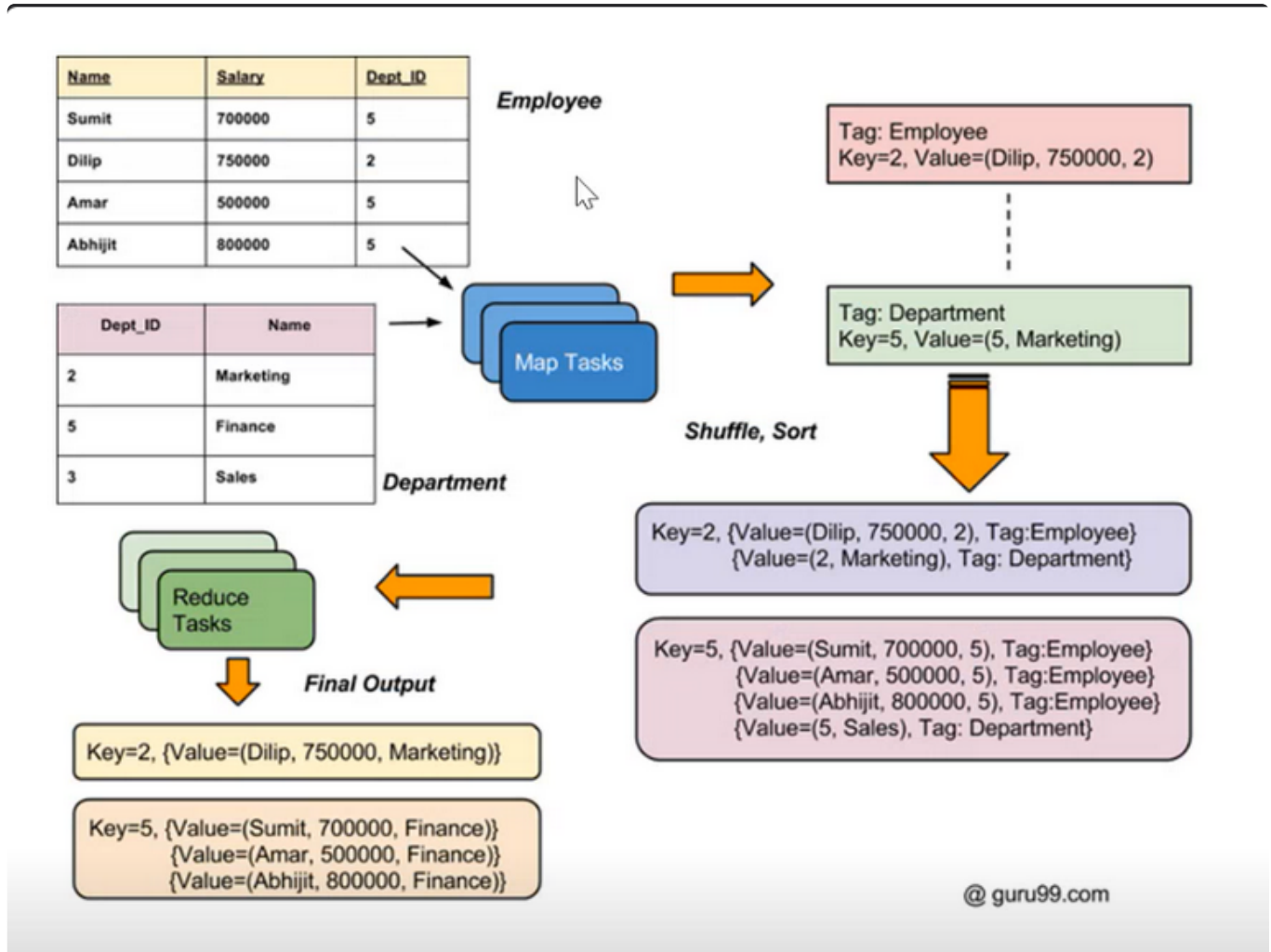
21. Map Reduce Architecture



- **Job Tracker**
 - Acts like the **master**.
 - It **receives the job** from the client, splits it into tasks, and assigns them to Task Trackers.
 - Monitors the **overall progress** of the job.

- **Task Tracker**
 - Acts like the **worker**.
 - It **runs the actual tasks** (Map or Reduce) on the **DataNode**.
 - Sends progress updates back to the Job Tracker.
- **Job Tracker** = Manager of the job
- **Task Tracker** = Worker that does the job on the data

22. Reducer Side Join



- A **Reducer-Side Join** is a technique used in MapReduce to **combine (join)** two or more datasets **based on a common key** (like joining tables in SQL).

1. Input: You have two datasets, say:

- One with **employee info**
- One with **department info**
- Both share a common key, like `department_id`.

2. Map Phase:

- Each mapper reads both datasets.
- For every record, it **emits a key-value pair** where the key is the common field (e.g., `department_id`).
- The value includes the record and a **tag** to show which dataset it came from.

3. Shuffle & Sort:

- MapReduce groups all records with the **same key** (same `department_id`) together.
- These grouped records are sent to a **single reducer**.

4. Reduce Phase:

- The reducer gets all data for one key (e.g., all employees and the department info for `department_id = 101`).
- It then **joins the records** by combining values from both datasets that match the key.

Imagine:

- Employee file:
`101, John`
`102, Alice`
- Department file:
`101, HR`
`102, IT`

The join result will be:

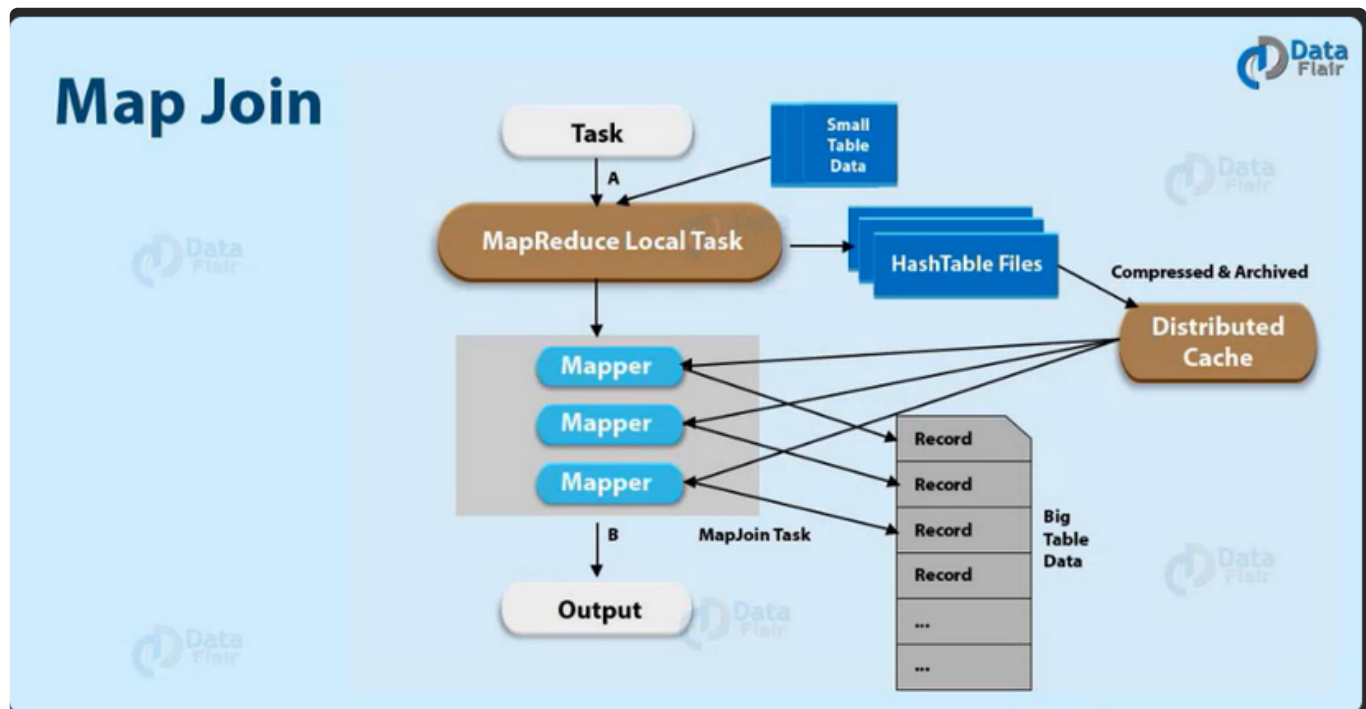
```
101, John, HR
102, Alice, IT
```

When to Use:

- When **datasets are large** and you want full control over the join logic.
- Works well even if data is **not sorted or partitioned** in advance.



23. Map Side Join



A Map-Side Join is a way to join a large dataset with a small dataset directly in the Map phase, without needing a Reducer.

1. The small dataset is loaded into memory on all mapper nodes.
2. The mappers then read the large dataset.
3. For each record in the large dataset, the mapper joins it with the matching record from the small dataset (already in memory).
4. The join is completed in the map phase, so no reducer is needed.

- Benefits
 - No shuffle phase → avoids moving big data between nodes.
 - Much faster than reducer-side join for this use case.
 - Cost-effective for joining small + large datasets.

24. MapReduce Design Patterns

MapReduce design patterns are **common templates or strategies** that help developers write efficient and organized data processing jobs.

They are useful for tasks like:

- **Filtering** data

- **Joining** datasets
- **Summarizing** information
- **Sorting** data
- **Transforming** data formats

Common Patterns:

1. **Input → Map → Reduce → Output**

- The standard pattern (used in word count, aggregation, etc.)

2. **Input → Map → Output**

- No reduce step needed (used for filtering or simple data transformation).

3. **Input → Multiple Maps → Reduce → Output**

- Different mappers handle different input types (used in joins or multi-source input).

4. **Input → Map → Combiner → Reduce → Output**

- Uses a **combiner** between Map and Reduce to **optimize** by reducing data early.



Hive

1. What is Hive?

- Apache Hive is a data warehouse tool on top of hadoop
- Uses SQL like language (HiveQL) for querying large datasets
- Converts SQL queries into MapReduce/Spark Jobs.
- Supports structured and semi structured data
- Hive can be used for variety of data processing tasks such as Data warehousing, ETL Pipelines



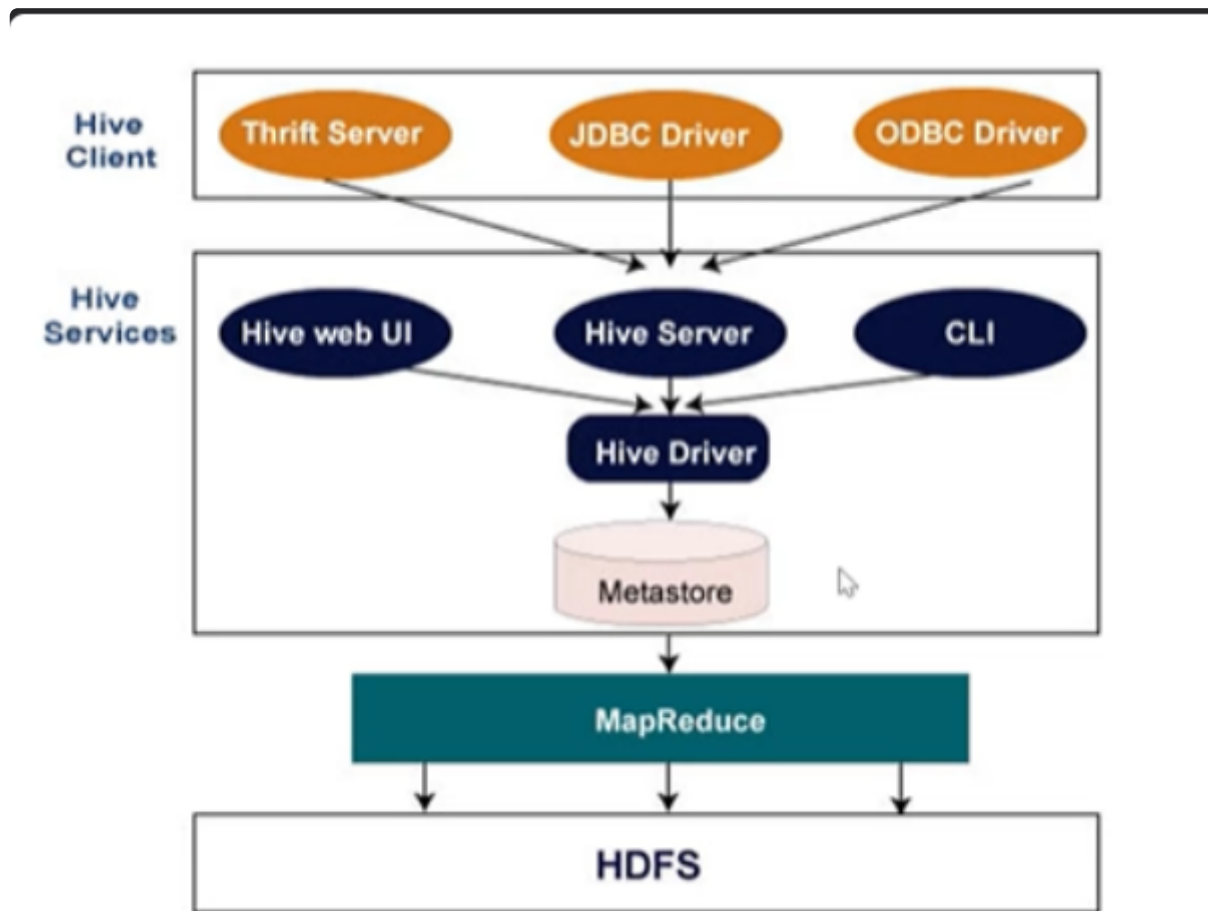
2. Features of Hive

- **Schema Storage:** Stores table definitions (schema) in a database and keeps the actual data in HDFS.
- **Designed for OLAP:** Built for Online Analytical Processing — ideal for complex queries and reporting.

- SQL-Like Language: Uses HiveQL (or HQL), a language similar to SQL, to query data easily.
- Fast, Scalable, Extensible:
 - Can handle large datasets quickly,
 - Easily grows as data grows,
 - Allows adding custom functions to extend capabilities.



3. Hive Architecture



1. Hive Client

This is how users or other applications connect to Hive. It includes:

- **Thrift Server**
- **JDBC Driver**
- **ODBC Driver**

These allow external tools to talk to Hive and send queries.

2. Hive Services

- **Hive Web UI**
 - A website interface where users can type queries and interact with Hive easily.
- **Hive Server (Apache Thrift Server)**
 - Receives queries from clients.
 - Passes queries to the **Hive Driver**.
 - Sends jobs for processing (using MapReduce or Spark).
- **CLI (Command Line Interface)**
 - Lets users interact with Hive via command line.

3. Hive Driver

- Takes queries from users.
- Manages the session and communication between the client and Hive system.
- Uses APIs like JDBC or ODBC for interaction.
- Pulls metadata (info about data structure)

4. Metastore

- Stores **metadata**: details about tables, columns, partitions, and how data is stored in HDFS.
- Acts like a **catalog** or **library index** for the data.

5. Compiler

- Reads and understands the query.
- Checks if the query is valid (semantic analysis).
- Creates an **execution plan** (step-by-step instructions) based on metadata from the Metastore.

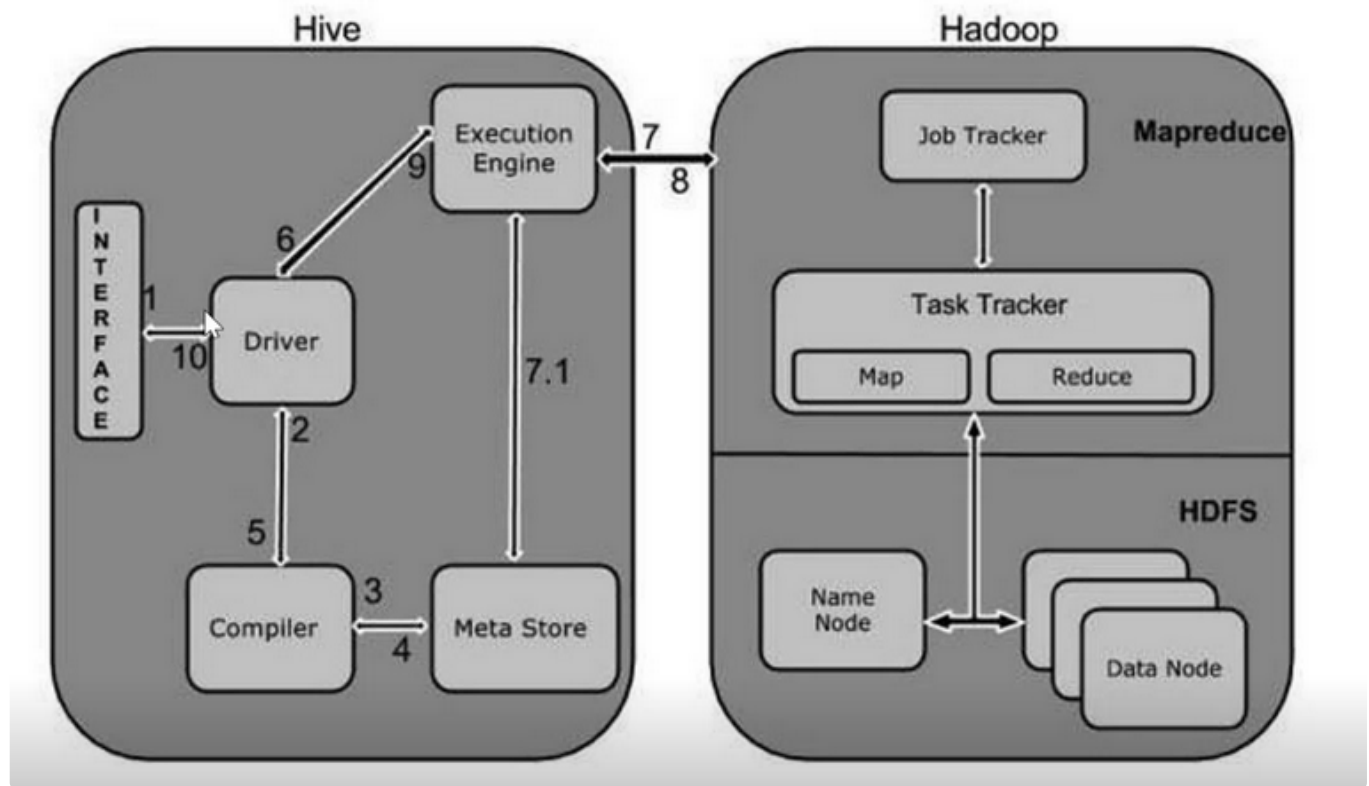
6. Execution Engine

- Runs the execution plan created by the compiler.
- Breaks the work into stages and manages dependencies.
- Sends tasks to the Hadoop cluster to process the data.

Hive turns your SQL-like queries into big data jobs by:

Client → Server → Driver → Compiler → Execution Engine → Hadoop

4. Hive Flow



1. Getting the Request

- You send a query through an interface (like Hive Web UI, CLI, or a program).

2. Driver Sends to Compiler

- The **Hive Driver** receives the query and passes it to the **Compiler**.

3. Compiler Plans Execution

- The Compiler parses the query and creates an **execution plan**.
- It **checks the Metastore** to understand the data structure (tables, columns, partitions).

4. Driver Sends Info to Execution Engine

- Once the plan is ready, the Driver sends it to the **Execution Engine**.

5. Execution Engine Runs MapReduce Jobs

- The query is broken down into tasks (MapReduce jobs).
- For example, if the table has 10 rows and the query is to fetch only female records:

- **10 mappers** are created, each processes one row.
- Each mapper converts its row into key-value pairs.
- The **Reducer filters** and collects only female rows.
- Data is fetched from **HDFS DataNodes**.

6. Role of Execution Engine

- It turns the query plan into a **DAG (Directed Acyclic Graph)** of stages.



5. Job execution flow in Hive

- **1. Execute Query**
 - You submit a query through Hive's interface (like Command Line, Web UI, or a program using ODBC/JDBC).
 - The interface sends this query to the **Hive Driver** for execution.
- **2. Get Plan**
 - The **Driver** creates a session for this query and sends it to the **Compiler**.
 - The Compiler's job is to create an **execution plan** for the query.
- **3. Get Metadata**
 - The Compiler requests metadata (info about tables, columns, partitions) from the **Metastore**.
- **4. Send Metadata**
 - The Metastore sends the requested metadata back to the Compiler.
- **5. Send Plan**
 - The Compiler sends the completed **execution plan** back to the Driver.
- **6. Execute Plan**
 - The Driver passes this execution plan to the **Execution Engine**.
- **7. Fetch Results**
 - After the plan runs, results are fetched by the Driver.
- **8. Send Results**
 - The Driver sends the results back to the **user interface** where you see them.



6. Where is Metastore stored in the case of Hive

- The Metastore is stored externally, outside of Hadoop's HDFS.
- Usually, it's kept in a relational database like MySQL, PostgreSQL, or Oracle.
- This external database stores all the metadata about tables, schemas, partitions, and more.



7. Hive Data types

- **1. Column Types**
 - Basic data types used for table columns like:
 - `INT` (integer numbers)
 - `STRING` (text)
 - `FLOAT` , `DOUBLE` (decimal numbers)
 - `BOOLEAN` (true/false)
- **2. Literals**
 - Fixed values written directly in queries, such as:
 - Numbers (`123`)
 - Strings (`'Hello'`)
 - Boolean (`TRUE` , `FALSE`)
- **3. Null Values**
 - Represents **missing or unknown data** in tables.
 - Hive handles `NULL` to indicate the absence of a value.
- **4. Complex Types**
 - Advanced data types to store complex data structures:
 - **ARRAY**: ordered list of elements
 - **MAP**: key-value pairs
 - **STRUCT**: group of fields with different types



8. Hive Tables

Managed Table

- Hive controls **both metadata and actual data**.
- When you **drop (delete)** the table, the **data is also deleted** from HDFS.
- Ideal when Hive is the only system managing the data.

External Table

- Hive manages **only the metadata**.
- The **data stays in its original location**, even if you drop the table.
- Useful when data is **shared with other tools** or stored outside Hive's control.



9. Partitioning

- Improves query performance by reducing data scanned
- Hive divides a table into partitions based on columns value
- Common use case: Partitioning sales data by date or region



10. Static vs Dynamic partitioning

- Static Partitioning
 - Manually specify partition values while inserting data
- Dynamic Partitioning
 - Hive determines partitions automatically



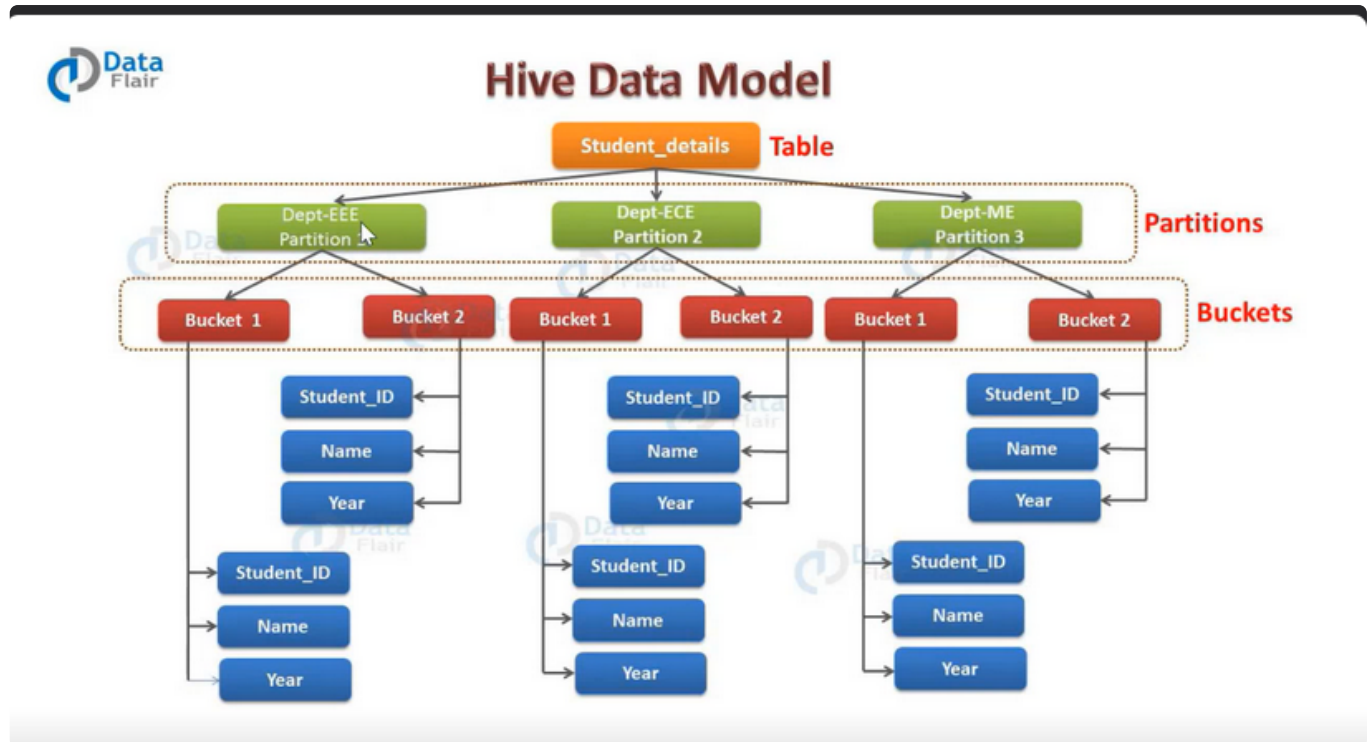
11. Bucketing

- Further divides partitioning into equal sized buckets based on hash function
- Helps with joins
- Bucketing calculation
 - Table size = 2300MB
 - Block size = 128MB
 - Each data node size

- $2300\text{MB}/128\text{MB} = 17.96$
- $2^n = 17.96$
- $n = \text{bucket size}$



12. Hive Data model



13. Partition vs Bucketing

Feature	Partitioning	Bucketing
Column Type	Partition Column (e.g., year)	Bucket Column (e.g., user_id)
How It Divides	Based on column values (logical chunks)	Based on hashing the column value
Storage Layout	Creates directories for each partition	Creates files inside each directory
Type	Dynamic (partitions can be created on the fly)	Static (you fix the number of buckets)
Use Case	Helps in filtering and reducing scanned data	Helps in joins and sampling queries





14. Hive indexing

Hive indexing helps make queries faster by creating an index on specific columns—like a shortcut to quickly find data.

- Compact Index
 - Simple index storing a list of keys and their locations.
- Bitmap Index
 - Uses bitmaps (arrays of bits) to represent data presence, efficient for columns with few unique values.
- Indexes are stored in separate tables from the main data.
- When the main table is updated, indexes must be rebuilt to stay accurate.
- Hive indexing is less common now because techniques like partitioning and bucketing often improve performance more effectively.

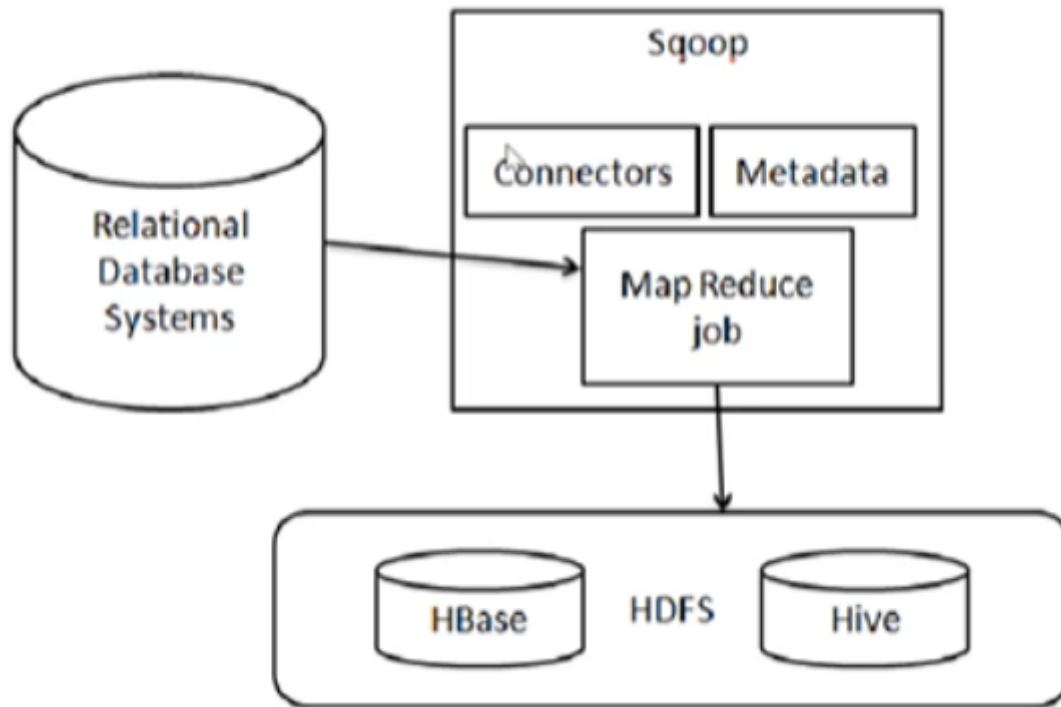


SQOOP

1. What is SQOOP?

- **SQOOP** is a tool that helps move large amounts of data **between traditional databases (RDBMS)** like MySQL or Oracle and **Hadoop**.
- It makes importing and exporting data **fast and efficient**.
- Supports **parallel processing** to speed up data transfer.
- Reduces the amount of coding or manual work needed to move data.
- Can be set up to run **automatically on a schedule** for regular data updates and analytics.

2. SQOOP Architecture



1. Relational Database Systems (RDBMS)

- Source or target databases like MySQL, Oracle, SQL Server where data is imported from or exported to.

2. SQOOP Client

- The tool you run to move data.
- Uses **connectors** like **JDBC** or **ODBC** to talk with databases.
- Manages **metadata** about data and jobs.
- Uses **MapReduce framework** to perform data import/export in parallel, speeding up the process.

3. HDFS (Hadoop Distributed File System)

- The storage system for Hadoop where data is stored after import.
- Can also interact with systems like **HBase** and **Hive** for further data management and querying.



3. SQOOP Workflow

- Importing data from RDBMS to HDFS
- Exporting data from HDFS to RDBMS

4. SQOOP Commands

- **List Databases**

```
sqoop list-databases --connect jdbc:mysql://localhost/ --username root
```

Lists all databases available in the MySQL server.

- **List Tables**

```
sqoop list-tables --connect jdbc:mysql://localhost/userdb --username root
```

Lists all tables inside the `userdb` database.

- **Import Data**

```
sqoop import --connect jdbc:mysql://host/db --table employees --target-dir /data/employees
```

Imports the `employees` table from the database into HDFS directory `/data/employees`.

- **Export Data**

```
sqoop export --connect jdbc:mysql://host/db --table employees --export-dir /data/employees
```

Exports data from HDFS `/data/employees` back into the `employees` table in the database.

- **Create Job**

```
sqoop job --create myjob --import --connect jdbc:mysql://localhost/db --username root --table employee -m 1
```

Creates a reusable import job named `myjob`.

- **List Jobs**


```
sqoop job --list
```

Lists all saved jobs.

- **Show Job Details**

```
sqoop job --show myjob
```

Displays the details of the job named `myjob`.

- **Execute Job**

```
sqoop job --exec myjob
```

Runs the saved job `myjob`.



FLUME

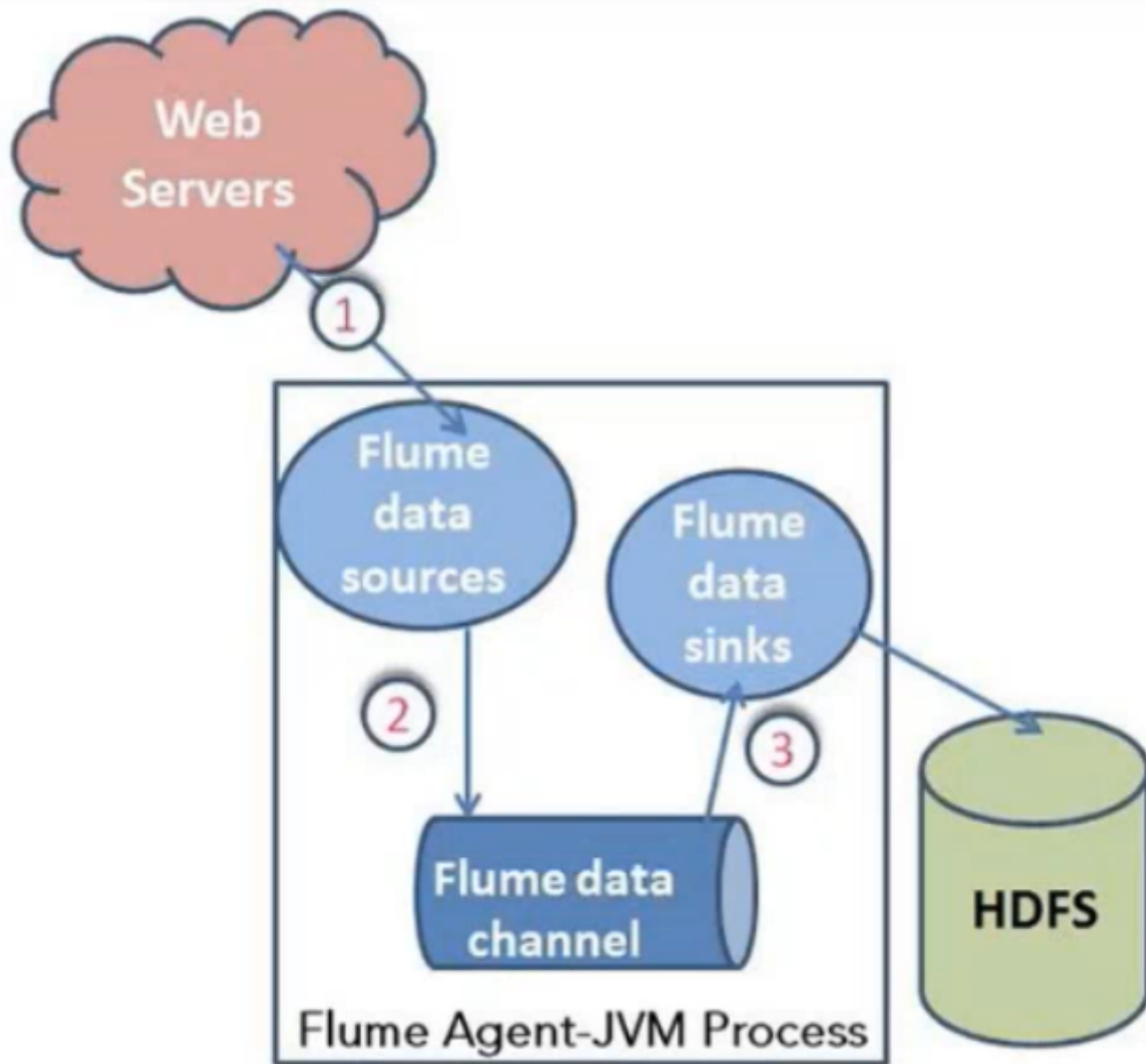
1. What is Flume?

- Apache Flume is a distributed tool designed to collect, aggregate, and move large amounts of streaming data (like logs) into Hadoop's HDFS or other storage systems.
- It's mainly used to handle real-time streaming data efficiently.
- Flume is scalable, so it can grow to handle more data easily.
- It's also fault tolerant, meaning it can handle failures without losing data.

Why use flume?

- To collect data continuously in real-time from many sources.
- To aggregate and transport this data reliably to a central storage like HDFS for analysis.

2. Flume Architecture



- **External sources** (like web servers or application logs) generate **events** (data).
- These events are sent to the **Flume Source**, which understands the format of the incoming data.
- The **Flume Source** receives the events and stores them in one or more **Channels**.
- A **Channel** acts like a temporary storage or buffer, holding events until they are processed.
- Channels can use memory or the local file system to store events.
- The **Flume Sink** takes events from the channel and writes them to an external system like **HDFS**, **HBase**, or other storage.
- **Source** → **Channel** → **Sink** is the flow of data inside Flume.
- Channels make sure data is safely stored until sinks can process it.
- This design helps Flume handle large volumes of streaming data reliably and efficiently.

3. Flume Configuration

Source

- The place where data originates — e.g., log files, web servers, or other applications sending events.

Channel

- The temporary storage that holds data coming from the source until it is processed.
- Can be memory-based, file-based, or even use systems like Kafka.

Sink

- The destination where the data is finally stored — e.g., HDFS, HBase, or other storage systems.

Workflow:

- Data flows **in this order**:
Source → **Channel** → **Sink**
- This setup makes sure data moves smoothly and reliably from its origin to the storage.



PIG

1. What is PIG?

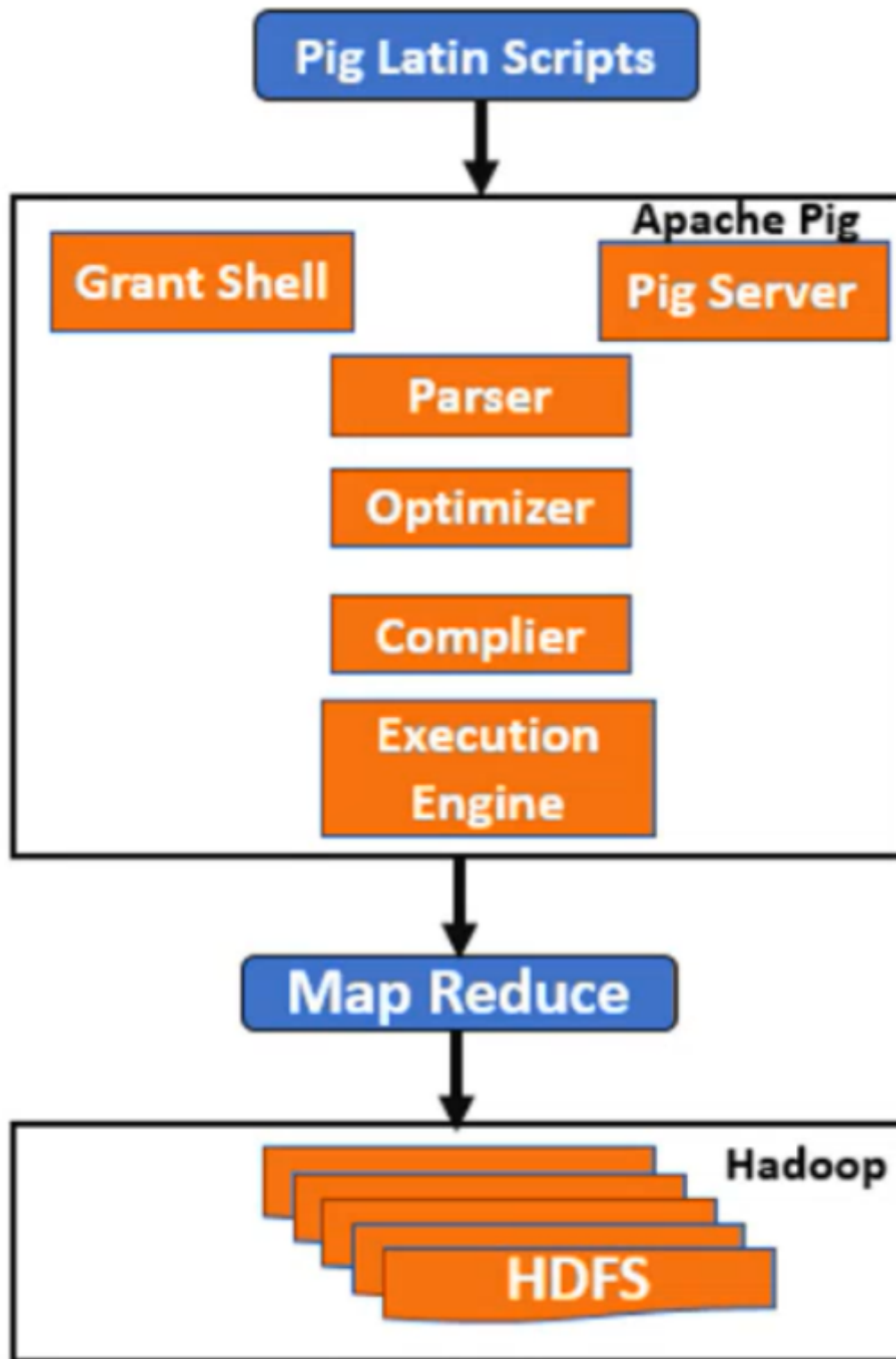
- Pig is a high-level scripting language called Pig Latin designed to process large datasets on Hadoop.
- It was created to make common data tasks like ETL (Extract, Transform, Load), aggregations, and joins easier and faster to write compared to writing raw MapReduce code.
- Pig lets developers focus on what to do with the data rather than how to do it with complex MapReduce programs.

Why use PIG?

- It simplifies complex MapReduce operations with easy-to-write scripts.

- Supports schema-less data processing, meaning you don't always need to define the data structure upfront.

2. PIG Architecture



- **Storage:**
 - Pig uses **HDFS** to store the data.
- **Processing:**
 - Pig converts your scripts into **MapReduce** (or Tez, Spark) jobs to process data in parallel.
- **Components:**
 - **Pig Latin Scripts:** The code you write to describe data operations.
 - **Parser:** Reads your Pig scripts and turns them into a DAG (Directed Acyclic Graph) showing the order of operations.
 - **Optimizer:** Improves the execution plan to make the process efficient.
 - **Compiler:** Converts the optimized plan into actual MapReduce (or Tez/Spark) jobs.
 - **Execution Engine:** Runs the jobs on the cluster.

Workflow:

1. **Load data** from HDFS
2. **Transform** it using Pig Latin operations (filter, join, group, etc.)
3. **Store results** back to HDFS or other storage

3. PIG Commands and usecases

Common Commands:

Load Data

```
employees = LOAD 'data/employees' USING PigStorage(',');
```

Loads the `employees` data from a CSV file.

Filter Data

```
filtered = FILTER employees BY salary > 50000;
```

Filters employees with salary greater than 50,000.

Store Data

```
STORE filtered INTO 'output' USING PigStorage(',');
```

Saves the filtered data back to HDFS in CSV format.

Use Cases:

- **Data transformations in ETL pipelines** — cleaning, filtering, and preparing data.
- **Processing semi-structured data** like logs or JSON where schema may not be fixed.

4. Sqoop,Flume,Pig

- SQOOP -> Best for structured data transfers
- Flume -> Best for log and real time event ingestion
- Pig -> Best for complex data processing



HBase

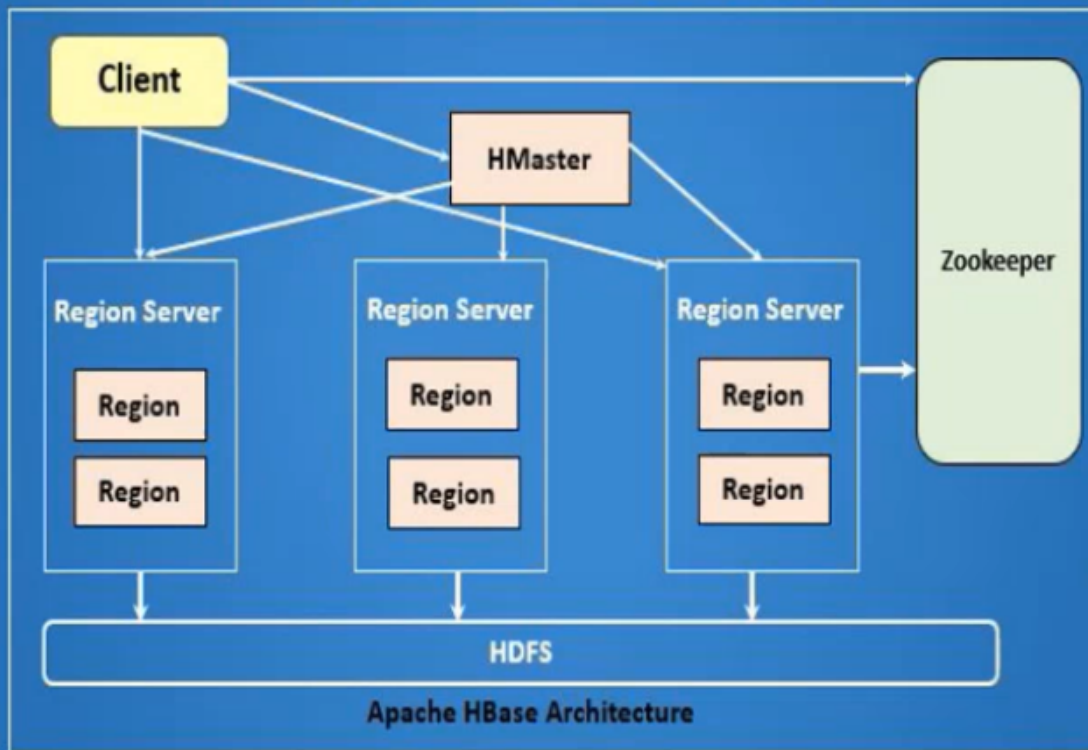
1. What is HBase?

- HBase is a NoSQL Distributed database built on top of Hadoop
- Provides real time read/write access to large datasets
- Stores data in key-value format



2. HBase Architecture

HBase Architecture



www.educba.com

HMaster

- The master node that manages the cluster, including regions and balancing the load across servers.

RegionServer

- Worker nodes that handle the actual read and write requests from clients.

Zookeeper

- A coordination service that helps manage the distributed nature of HBase, keeping track of server status and ensuring smooth operation.

HDFS

- The storage layer where the actual data is stored persistently across the cluster.

HBase uses a master-worker model with **HMaster** controlling the system, **RegionServers** processing data requests, **Zookeeper** coordinating everything, and **HDFS** storing the data reliably.



3. How is data stored in HBase

edureka!

Row Key		Column Family			
Row Key		Customers		Products	
Customer ID	Customer Name	City & Country	Product Name	Price	Column Qualifiers
1	Sam Smith	California, US	Mike	\$500	Cell
2	Arijit Singh	Goa, India	Speakers	\$1000	
3	Ellie Goulding	London, UK	Headphones	\$800	
4	Wiz Khalifa	North Dakota, US	Guitar	\$2500	

Figure: HBase Table

- Its a column oriented database, and tables in HBase are sorted by row.
- The table schema is defined by column families, which are nothing but key value pairs

4. Features of HBase

- Scalable
 - Handle petabytes of data
- Automatic Sharding
 - Data is split into regions
- Strong consistency
 - ACID Properties for a single row
- Column oriented Storage
 - Efficient for analytics

5. Use cases of HBase

- Real time analytics (Fraud detection)
- Time series data (eg: sensor data)
- Social Media (eg: User engagement tracking)

6. HDFS vs HBase

Feature	HDFS	HBase
Type	Distributed file storage system	NoSQL database running on top of HDFS
Fault Tolerance	Highly fault tolerant	Partially fault tolerant, highly consistent
Read/Write Pattern	Sequential read/write	Supports random read/write access
Data Modification	Write once, read many	Supports dynamic data updates
Architecture	Rigid, designed for large batch jobs	Flexible, supports real-time access
Use Case	Offline batch processing	Real-time processing and quick lookups
Latency	High latency for access operations	Low latency for small, random data reads/writes

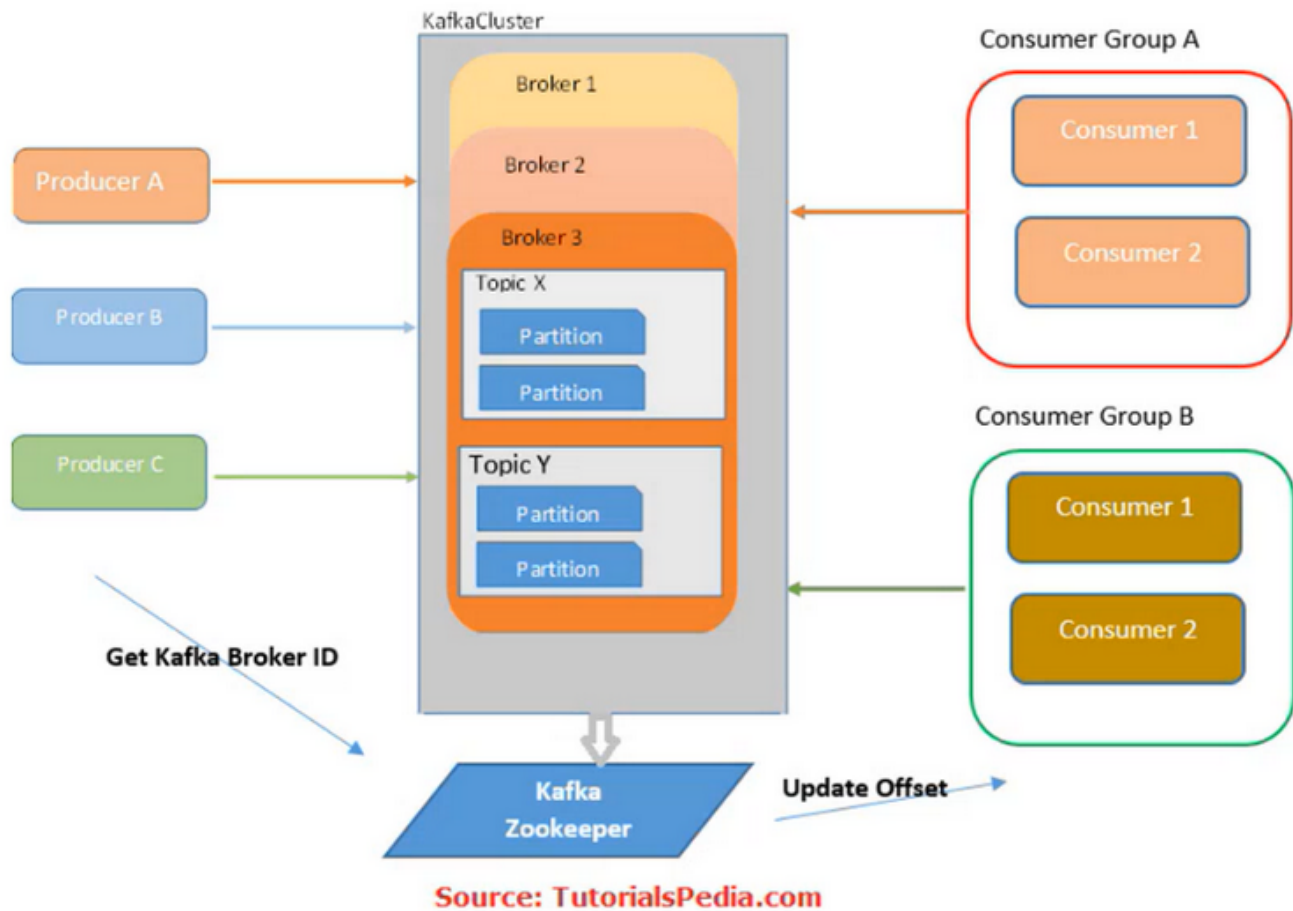


KAFKA

1. What is KAFKA

- It lets apps **send** (produce), **store**, and **receive** (consume) data **in real-time**.
- It is used to build **real-time pipelines** (like plumbing for data) and **event-driven applications** (apps that react to events like "user clicked" or "sensor sent reading").
- Distributed event streaming platform
- Used for real time data pipelines and event driven applications
- Why use KAFKA?
 - **Log Aggregation** – Combine logs from different servers/applications in one place.
 - **Real-Time Analytics** – Process data as it arrives (e.g., fraud detection).
 - **IoT Streaming** – Collect data from sensors/devices in real-time.

2. KAFKA architecture



Producer

- Sends (writes/publishes) data/messages to **Kafka Topics**.
- Example: A temperature sensor sending readings every second.

Topic

- A **named category** or **channel** where data is stored.
- Think of it like a folder where similar messages go.

Broker

- A **Kafka server** that stores the data.
- Kafka has many brokers working together in a **cluster**.
- Example: Broker 1, Broker 2, etc.

Partition

- A topic is **split** into parts (called partitions) for **parallel processing**.

- This allows Kafka to handle **huge data volumes**.

Consumer

- An app that **reads** messages from topics.
- Example: A dashboard that shows real-time temperature graphs.

Consumer Group

- A group of consumers that share the load.
- Each consumer gets a **slice** of the data.

ZooKeeper

- Helps **manage brokers**, **elect leaders**, and keep the **cluster organized**.
- Example: If a broker goes down, ZooKeeper helps reassign tasks.

3. Features of KAFKA

Feature	What it means
High Throughput	Can handle millions of messages/sec
Low Latency	Messages move quickly , nearly real-time
Scalable	Add more brokers/consumers easily → Kafka grows with your data
Fault Tolerant	Messages are replicated → no data loss if a server dies
Durable	Data can be stored for days/weeks
Replayable	Consumers can re-read messages from any point
Distributed	Kafka works as a cluster of servers , not just one

4. KAFKA Workflow

Imagine you're running a **food delivery app** like Zomato or Swiggy. Here's how Kafka would help in handling events like "**Order Placed**", "**Location Updates**", or "**Delivery Status Changes**".

1. Producer sends data to Kafka

- A **Producer** is an app or service that generates data.

- Example: When a user places an order → your backend sends a message:
`{ "order_id": 123, "status": "placed" }`
- This message is **sent to a Kafka Topic**, say `orders`.

```
producer.send("orders", value={"order_id": 123, "status": "placed"})
```

2. Kafka Topic receives and organizes the data

- A **Topic** is like a **channel** or **folder** in Kafka where messages of the same kind are stored.
- You can think of a Topic as:
 - `orders` topic → stores all order-related messages
 - `locations` topic → stores all GPS updates
 - `payments` topic → stores all payment statuses
- Topics are **split into partitions** to improve **scalability**.
 - E.g., Topic `orders` has 3 partitions: `orders-0`, `orders-1`, `orders-2`
 - Kafka decides (or producer specifies) which partition the message goes into.

Topic: `orders`

├─ Partition 0 → messages: `order#100`, `order#103`
├─ Partition 1 → messages: `order#101`, `order#104`
├─ Partition 2 → messages: `order#102`, `order#105`

3. Brokers store and manage messages

- A **Broker** is a Kafka server. Each **partition lives inside a broker**.
- Kafka **clusters** have multiple brokers (e.g., 3 or more) for **load balancing** and **fault tolerance**.
- Each partition has:
 - One **Leader Broker** → handles reads/writes
 - One or more **Follower Brokers** → keep **replicas** of the data
 - If a leader broker crashes, a follower is **promoted** to leader.

Broker 1 → `orders-0` (Leader)

Broker 2 → `orders-1` (Leader), `orders-0` (Replica)

Broker 3 → `orders-2` (Leader), `orders-1` (Replica)

4. Consumers read the messages

- A **Consumer** subscribes to a topic and reads messages.
- Example: A delivery tracking service consumes messages from the `orders` topic.
- Consumers **keep track of their position** using an **Offset**.

```
Offset: 0, 1, 2, 3, ...
```

- The consumer reads messages **sequentially** per partition.
- Multiple consumers can **share work** using **Consumer Groups**.

```
Consumer Group: order-tracker
- Consumer 1 → reads Partition 0
- Consumer 2 → reads Partition 1
- Consumer 3 → reads Partition 2
```

5. ZooKeeper

- Helps **coordinate brokers** and manage **cluster state**.
- Handles:
 - Electing **leader brokers**
 - Notifying if a broker crashes
 - Maintaining **metadata** about topics and partitions

6. Consumers process and act on messages

Once data is consumed, apps can:

- Update the **database**
- Trigger **notifications**
- Train **ML models**
- Send **data to dashboards**

7. Retention and Replay

Kafka **stores messages** for a configurable **retention period** (e.g., 7 days).

- A new consumer can **replay messages** from the beginning if needed.
- This makes Kafka perfect for **reprocessing data** without needing a new source.

5. Summary

Component	Role
Producer	Sends data to Kafka
Topic	Organizes data into categories
Partition	Splits topic data for scalability
Broker	Kafka server that stores data
Consumer	Reads data
Consumer Group	Shares load of reading
ZooKeeper	Manages Kafka cluster (older versions)



Spark

1. What is spark?

Apache Spark is a **powerful tool** used to **process big data really fast**.

Imagine you have a huge pile of data (like all the transactions in Amazon or all posts on Instagram). Processing that much data on one computer is slow. **Spark helps by splitting the work across many computers** and doing it all at once — this is called **parallel processing**.



2. Why Spark?

Hadoop is slower

Hadoop stores data on disk at every step. Reading from and writing to disk again and again takes time, which makes Hadoop slower.

Spark is faster

Spark keeps most of the data in memory (RAM) during processing. This reduces time spent on disk operations and makes it much faster—sometimes up to 100 times faster than Hadoop for certain tasks.

How does Spark do it?

- Uses memory instead of disk for most steps
- Splits the work across many machines and runs in parallel
- Builds a plan ahead of time (called a DAG – Directed Acyclic Graph) to optimize how the work is done

Spark supports many programming languages

Spark lets you write code in multiple languages, such as:

- Python (called PySpark)
- Java
- Scala
- R



3. Spark Implementation

Spark is flexible. You can run it in different ways depending on your setup.

1. Standalone Mode

Spark can run on its own, without needing any other system. This is called **Standalone mode**. It's simple to set up and is often used for small to medium-scale projects.

2. With Other Cluster Managers

Spark can also work with other big data systems that manage clusters. The most common one is **Hadoop YARN** (Yet Another Resource Negotiator).



4. Hadoop vs Spark

Feature	Hadoop	Spark
Main Role	Distributed storage and compute	Distributed compute only
Processing Engine	MapReduce	Spark Engine

Feature	Hadoop	Spark
Storage	Writes to disk after every step	Stores data in memory (RAM)
Speed	Slower (due to disk I/O)	Faster (due to in-memory processing)
Programming Model	Map → Reduce (less flexible)	Generalized (can handle many use cases)
Supports Loops/Iterations	No (inefficient for repeated tasks)	Yes (very efficient for loops & ML tasks)
Type of Processing	Batch processing only	Batch + Real-time (streaming)
Ease of Use	More complex, requires more tuning	Easier, supports multiple APIs (Python, etc.)
Best For	Large batch jobs with simple logic	Fast, complex, or repeated processing tasks



5. What Hadoop gives spark

Component	What It Does	How Spark Uses It
YARN	A resource manager (task manager)	Helps Spark assign tasks to different computers in the cluster.
HDFS	Hadoop Distributed File System	Spark can read/write data from/to HDFS. If Spark runs out of memory (RAM), it can store overflow data in HDFS.
Disaster Recovery	Built-in redundancy in Hadoop	If a machine fails, Hadoop ensures data is not lost . Spark benefits from this reliability.



6. Spark Components

Spark
Streaming

MLib

Spark SQL

Graph X

Apache Spark Core

- Apache Spark is made up of several important parts. At the heart of everything is **Spark Core**, and around it are specialized libraries for different types of data processing.

1. Spark Core

This is the base engine of Spark. It handles things like task scheduling, memory management, and fault recovery. All other components are built on top of Spark Core.

2. Spark SQL

Spark SQL allows you to process **structured data** using SQL-like queries.

You can work with tables, join datasets, and even connect to Hive or read from files like CSV, JSON, and Parquet.

It supports both SQL syntax and the DataFrame API.

3. Spark Streaming

This is used for **real-time data processing**.

For example, if you want to analyze live Twitter feeds, server logs, or sensor data—Spark Streaming breaks the data into small batches and processes them almost instantly.

4. MLlib

MLlib is Spark's built-in **machine learning library**.

It has tools for:

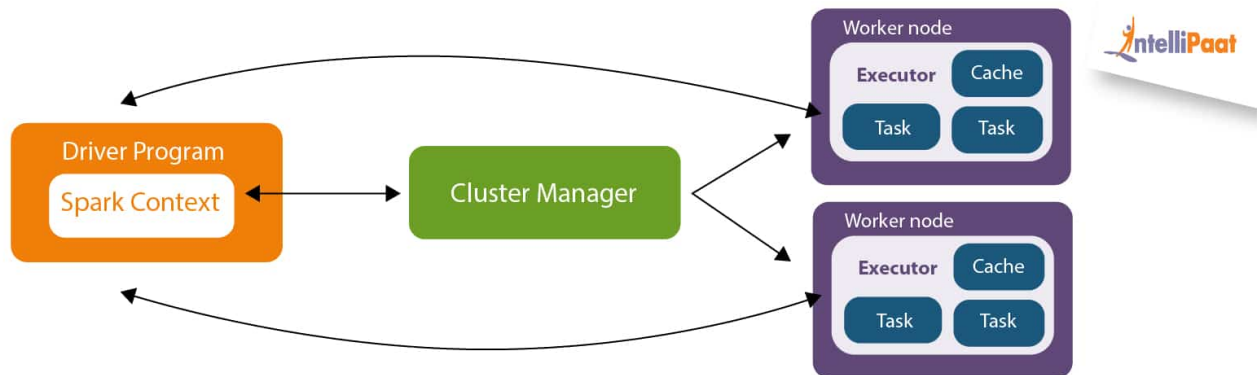
- Classification
- Regression
- Clustering
- Recommendation systems
 - It's useful for training and testing models using distributed data.

5. GraphX

- GraphX is used for **graph processing**.
- It helps analyze data where relationships matter—like social networks, user connections, or shortest path problems. You can build and query graphs directly using Spark.



7. Spark architecture



- Driver program
- spark context
- cluster manager
- worker node
- executors

When you run a Spark job, several parts work together to split the work, send it to different machines, and bring the results back. Let's break it down step by step.

1. Driver Program

- This is the **main program** you write in Python, Scala, or Java.
- It runs on your machine or a master node and **controls the entire job**.
- It contains your code and logic for reading data, transforming it, and saving results.

2. SparkContext

- The driver program creates a **SparkContext** — the entry point to Spark.
- It tells Spark how to access the cluster, how many resources to use, and keeps track of job progress.
- Think of SparkContext as the coordinator that talks to the cluster.

3. Cluster Manager

- Spark needs machines to run tasks. The **cluster manager** connects Spark to those machines.
- It decides which machines are free, and gives Spark the resources it needs.
- Examples:
 - **Standalone** (built-in Spark manager)
 - **YARN** (from Hadoop)

4. Worker Nodes

- These are the **machines where the actual work happens**.
- Each worker node runs one or more **executors** to process data.

5. Executors

- An **executor** is a process launched on each worker node.
- Executors:
 - **Run your tasks**
 - **Store data in memory**
 - **Send results back to the driver**
- Executors live for the whole duration of your Spark job.

In short:

- Your **driver program** starts everything.
- **SparkContext** talks to the **cluster manager**, which gives machines.
- Spark sends code to **executors** on **worker nodes**, which do the work and return the results.



8. Spark flow – How a Spark Program Runs

1. User writes spark code and submits
2. Driver creates spark context
3. spark context connects to cluster manager
4. Cluster manager allocates worker nodes, starts executors

5. driver sends tasks to executors
6. executors perform task and return to driver
7. Result combine and send to user

Let's understand what happens step by step when you run a Spark job:

1. User writes Spark code and submits it

- You write your Spark code using Python (PySpark), Scala, or Java. This code defines what data to load, how to process it, and where to save results.

2. Driver creates SparkContext

- The main program (called the **Driver**) starts and creates a **SparkContext**. This is the entry point to interact with Spark and start a job.

3. SparkContext connects to the Cluster Manager

- SparkContext requests resources from a **Cluster Manager** (like YARN, Kubernetes, or Spark's own standalone manager).

4. Cluster Manager allocates worker nodes and starts Executors

- The cluster manager assigns **worker machines** and launches **executors** (processes that will actually run the code) on them.

5. Driver sends tasks to Executors

- Based on your code, the Driver breaks the job into smaller tasks and sends them to the executors.

6. Executors perform the tasks and return results

- Each executor performs its assigned part of the job—like reading data, applying filters, doing joins, etc.—and then returns the results to the Driver.

7. Driver combines the results and gives the output to the user

- Once all tasks are complete, the Driver program collects and combines the results and either shows the final result or stores it to a file/database as defined in your code.



9. Executors

Executors are the workers in Spark. Once the cluster manager assigns resources, **executors are started on worker nodes**. They are responsible for:

- **Running the actual tasks** assigned by the driver
- **Storing data in memory** if caching is used
- **Sending results** back to the driver
- **Sending regular heartbeat signals** to the driver to say "I'm alive"
- If the driver doesn't hear from an executor for a while, it assumes it failed and may reassign the task to another executor.

Executors live **only for the duration of the Spark application**, and they are shut down when the job ends.



10. Driver Side

The **Driver** is the main controller of a Spark job. It not only sends tasks but also listens and monitors everything happening across the cluster.

Key components on the driver side:

- **Spark Driver**
 - The top-level program that starts everything. It creates the `SparkContext` and coordinates the job.
- **RPC Environment**
 - RPC stands for **Remote Procedure Call**.
 - This is the **communication layer** that allows different parts of the system (like the driver and executors) to send messages to each other, even if they're running on different machines.
- **Heartbeat Receiver**
 - The driver regularly **receives heartbeat signals from executors**. This helps it know which executors are alive and working.
- **Task Scheduler**
 - This part is responsible for:
 - Scheduling tasks to executors

- Monitoring executor health
- If an executor fails to send a heartbeat, the scheduler may **reschedule the task** on another executor



11. Executor Side

The **Executor** is a worker that runs on a worker node and carries out the actual tasks.

Key components on the executor side:

- **Spark Executor**

It performs the tasks given by the driver and stores data in memory if needed.

- **RPC Environment**

Just like the driver, executors also have an RPC setup to **communicate with the driver**.

- **Heartbeat Sender**

The executor **sends heartbeat signals** to the driver at regular intervals.

This tells the driver, "I'm still working and healthy."

In simple terms:

- The **driver schedules the work**, and **executors do the work**.
- They **talk to each other using RPC**.
- Executors keep **sending heartbeats** to tell the driver they're alive.
- The **driver listens**, and if it doesn't get a heartbeat, it assumes something went wrong and takes action.



12. RDD – Resilient Distributed Datasets

- An **RDD** is the **basic building block of data** in Spark.
- It represents a **large collection of data**, split across many computers, that can be processed in parallel.

What is it?

RDD is how Spark **stores and processes data internally**. It's like a super-powered version of a list or array that is spread across many machines.

Key Properties of RDDs:

- **Immutable**

Once an RDD is created, it **can't be changed**. You can create a new RDD by transforming an existing one, but the original stays the same.

- **Resilient**

If a part of the data is lost (e.g., a node crashes), Spark can **rebuild that part** using the original data and transformation steps. So RDDs are fault-tolerant.

- **Distributed**

The data is **automatically split across multiple nodes** (computers), and each part can be processed separately and in parallel.

Why are RDDs useful?

- **Fast**

Since they can be stored in **memory (RAM)** and processed in parallel, RDDs are much faster than traditional disk-based systems.

- **Safe**

Thanks to fault tolerance, your job won't crash if a node fails. Spark knows how to **recover lost data**.

- **Scalable**

RDDs can handle **very large datasets**, easily scaling across hundreds or thousands of machines.



13. RDD Features

- **Immutable**

- Once created, an RDD cannot be changed. You can only create new RDDs by transforming existing ones. This helps avoid accidental data changes.

- **Partitioned**

- RDDs split the data into **multiple parts called partitions**. Each partition can be processed independently and in parallel on different machines.

- **Lazy Evaluated**

- Spark **does not immediately process RDD operations** when you write them. Instead, it builds a plan (called a DAG) and waits until an action (like counting or saving) is called. This helps optimize the whole job before running.
- **Persistence (Caching)**
 - You can tell Spark to **keep an RDD in memory or on disk** after it's computed. This is useful if you use the same data multiple times, making operations faster.
- **Fault Tolerance**
 - If a partition of data is lost due to a machine failure, Spark can **recompute that partition** from the original data and transformations, ensuring no data loss.



14. DAG (Directed Acyclic Graph)

- A **DAG** is a way Spark organizes the sequence of computations it needs to perform.
 - It represents **all the steps (transformations)** you want to apply on your data.
- Each **node in the DAG is a partition of an RDD**.
- The DAG shows the order of operations, making sure data flows correctly from one step to the next.
- It is “**acyclic**”, meaning the graph doesn't loop back on itself — tasks flow forward only.
- Spark uses the DAG to **optimize the execution plan** before running the job, so it can be faster and more efficient.



15. Transformation

A **transformation** creates a new RDD by applying a function to each element of an existing RDD.

- When you apply a transformation, Spark **does not run it immediately**. Instead, it records the operation to run later.
- This means transformations are **lazy** — Spark waits until an action is called to actually do the work.
- Transformations build a **lineage** (a plan of steps) that helps Spark recover lost data if needed.

- Every transformation returns a **new RDD**, but no data is processed until an action is triggered.

Example:

```
rdd1 = sc.textFile("data.txt") # Read file and create initial RDD
rdd2 = rdd1.map(lambda x: x.split()) # Split each line into words
(transformation)
rdd3 = rdd2.filter(lambda x: len(x) > 2) # Keep words longer than 2 chars
(transformation)
rdd3.count() # Action that triggers execution and returns the count
```

Types of Transformations:

- **Narrow Transformation:**
 - Data from one partition maps to exactly one partition in the new RDD (e.g., `map()`, `filter()`). These are faster because no data shuffle is needed.
- **Wide Transformation:**
 - Data from one partition can be used by many partitions in the new RDD (e.g., `reduceByKey()`, `groupByKey()`). These cause **shuffle** — data movement across the cluster — which is slower.



Pyspark

PySpark is the Python API for Apache Spark. It enables distributed data processing using Python. Ideal for big data analytics, machine learning, and batch processing.

PySpark Architecture

- **Driver Program:** Your main Python script. Manages execution.
- **Cluster Manager:** Allocates resources. (YARN, Mesos, Standalone)
- **Executors:** Workers that run tasks and store data.
- **Tasks:** Code units sent from the driver to executors for execution.

1. Create SparkSession and Load CSV Data

```
from pyspark.sql import SparkSession

# Create a SparkSession – entry point to PySpark
spark = SparkSession.builder.appName("Practice").getOrCreate()

# Load CSV file as DataFrame (with headers)
df_csv = spark.read.option("header", True).csv("your_file.csv")

# Returns a DataFrame object containing the CSV content
```



2. Replace Nulls in "city" Column with "unknown"

```
from pyspark.sql.functions import when, col

# Replace nulls in 'city' column with "unknown"
df_csv = df_csv.withColumn(
    "clean_city",
    when(col("city").isNull(), "unknown").otherwise(col("city"))
)

# Returns DataFrame with a new column 'clean_city' containing non-null
values
```



3. Delete Rows Where "job_title" Is Null

```
# Filter out rows where 'job_title' is null
df_csv = df_csv.filter(col("job_title").isNotNull())

# Returns a DataFrame with only rows that have job_title
```



4. Replace Null Salary with Mean

```
from pyspark.sql.functions import avg

# Cast 'salary' column to float for calculation
df_csv = df_csv.withColumn("clean_salary", col("salary").substr(2,
100).cast("float"))

# Calculate mean salary
mean_salary = df_csv.select(avg("clean_salary")).first()[0]

# Replace nulls in salary with mean value
df_csv = df_csv.withColumn(
    "clean_salary",
    when(col("clean_salary").isNull(),
mean_salary).otherwise(col("clean_salary"))
)

# Returns DataFrame with null salaries filled with mean
```



5. Who Has Higher Average Salary – Male or Female?

```
# Group by gender and calculate average salary
df_csv.groupBy("gender").agg(avg("clean_salary").alias("avg_salary")).show()

# Returns table: gender vs avg_salary
```



6. Inspecting the DataFrame

```
df_csv.show() # Shows first 20 rows
df_csv.printSchema() # Shows data types
df_csv.describe().show() # Shows count, mean, stddev, min, max for numeric
columns
df_csv.columns # Returns list of column names
df_csv.count() # Returns number of rows in the DataFrame
```



7. Drop All Rows Containing Any Null

```
df_no_nulls = df_csv.na.drop()

# Returns a DataFrame with no null values at all
```



8. Filter Out Rows With Null in a Specific Column

```
df_csv = df_csv.filter(col("job_title").isNotNull())

# Same as earlier: Removes rows where 'job_title' is null
```



9. Create a New Column in Uppercase (city)

```
# Create a new column 'city_upper' with uppercase city names
df_csv = df_csv.withColumn("city_upper",
col("city").cast("string").alias("city"))
df_csv = df_csv.withColumn("city_upper", col("city_upper").upper())

# Returns DataFrame with an additional column 'city_upper'
```



10. Drop Duplicates

```
df_csv = df_csv.dropDuplicates()

# Returns DataFrame with duplicate rows removed
```



11. Rename a Column

```
df_csv = df_csv.withColumnRenamed("job_title", "designation")

# Renames 'job_title' column to 'designation'
```



12. Filter Rows Based on Condition

```
df_csv.filter(col("gender") == "Male").show()

# Returns all rows where gender is 'Male'
```



13. Filter Names Ending with Pattern

```
df_csv.filter(col("name").endswith("son")).show()

# Returns rows where 'name' ends with 'son'
```



14. Filter Names Starting with "Dr."

```
df_csv.filter(col("name").startswith("Dr.")).show()

# Returns rows where name starts with "Dr."
```



15. Filter IDs Between 1 and 5

```
df_csv.filter(col("id").between(1, 5)).show()

# Returns rows with id values from 1 to 5
```



16. Substring Example

```
from pyspark.sql.functions import substring

df_csv = df_csv.withColumn("short_name", substring("name", 1, 5))

# Adds a new column 'short_name' with first 5 letters of 'name'
```



17. Multiple Filters (AND condition)

```
df_csv.filter((col("gender") == "Male") & (col("clean_salary") >
50000)).show()

# Returns rows that match both conditions
```



18. Add a Calculated Column (bonus = 10% of salary)

```
df_csv = df_csv.withColumn("bonus", col("clean_salary") * 0.10)

# Adds 'bonus' column as 10% of salary
```



19. Group By & Aggregation

```
from pyspark.sql.functions import count

df_csv.groupBy("department").agg(
    avg("clean_salary").alias("avg_salary"),
    count("*").alias("total_employees")
).show()

# Returns: department-wise average salary and employee count
```



20. Output Result to File



```
df_csv.write.mode("overwrite").option("header",
True).csv("output/cleaned_data")

# Saves the DataFrame to CSV files (one per partition) in
'output/cleaned_data'
```

Examples



1. Clean Missing Data

```
# Load CSV with header
df = spark.read.option("header", True).csv("people.csv")

# Drop rows with null names
df = df.filter(col("name").isNotNull())

# Replace nulls in city with 'unknown'
df = df.withColumn("city", when(col("city").isNull(),
"unknown").otherwise(col("city")))
```



2. Average Salary per Department

```
# Cast salary column to float
df = df.withColumn("salary", col("salary").cast("float"))

# Group by department and get average salary
df.groupBy("department").agg(avg("salary").alias("avg_salary")).show()
```



3. Rename & Filter Female Employees

```
# Filter females with salary > 60k
df = df.filter((col("gender") == "Female") & (col("salary") > 60000))

# Rename column 'job' to 'role'
df = df.withColumnRenamed("job", "role")
df.show()
```



4. Add Bonus & Save to File

```
# Add bonus column
df = df.withColumn("bonus", col("salary") * 0.1)

# Save result as CSV
df.write.mode("overwrite").option("header", True).csv("output/bonus_data")
```



5. Remove Duplicates and Filter by Pattern

```
# Remove duplicates
df = df.dropDuplicates()

# Filter names ending with 'son'
df.filter(col("name").endswith("son")).show()
```



6. Complex Filter with Aggregation

```
# Filter: Male + IT dept + salary > 50k
df = df.filter(
    (col("gender") == "Male") &
    (col("department") == "IT") &
    (col("salary") > 50000)
)
```



```
# Calculate average salary
df.agg(avg("salary")).show()
```



Scala

1. What is Scala and why is it used in Big Data?

→ Scala is a programming language used with Spark because it's fast and works well with big data tools.



2. What's the difference between `val` and `var` ?

→ `val` is like a constant — you can't change it. `var` is a variable — you can change it.



3. What is a case class?

→ A special class in Scala that's easy to use and helps with pattern matching and comparing data.

```
case class Person(name: String, age: Int)

val p1 = Person("Alice", 30)
println(p1.name) // Alice
```



4. What is a higher-order function?

→ A function that can take another function as input or give one as output.

```
def greet(name: String, f: String => String): String = f(name)

greet("Bob", _.toUpperCase) // Output: BOB
```





5. What's the difference between `map`, `flatMap`, and `filter` ?

→ `map` changes each item, `flatMap` does that and also flattens results, `filter` keeps only matching items.

```
List(1, 2, 3).map(_ * 2)           // List(2, 4, 6)
List(Some(1), None).flatMap(x => x) // List(1)
List(1, 2, 3).filter(_ > 1)       // List(2, 3)
```



6. What is Option in Scala?

→ A way to handle missing values safely — instead of `null`, you get `Some(value)` or `None`.

```
val name: Option[String] = Some("Tom")
val missing: Option[String] = None

println(name.getOrElse("No Name")) // Tom
println(missing.getOrElse("No Name")) // No Name
```



7. What are immutable collections?

→ Lists or sets that you can't change after making them — helps avoid bugs.

```
val list = List(1, 2, 3) // Immutable
val newList = list :+ 4 // Adds 4, original stays same
```



8. How is Scala both object-oriented and functional?

→ You can write code using classes (like Java) and also use functions like building blocks.

```
// OOP
class Animal(val name: String)

// FP
val double = (x: Int) => x * 2
```



9. What is a trait?

→ A reusable block of code that you can mix into classes — like a flexible version of an interface.

```
trait Speaker {
  def speak(): String
}

class Dog extends Speaker {
  def speak() = "Woof"
}
```



10. Why do we use Scala with Spark instead of Java?

→ Scala code is shorter, works better with Spark, and makes writing big data programs easier.



11. What is match function in scala

```
def check(num: Int): String = num % 2 match {
  case 0 => "Even"
  case _ => "Odd"
}
```



12. For loop in scala

```
for (i <- 1 to 5) {  
  println(i)  
}
```



Big Data Component Comparison Table

Component	What is it?	What is it used for?
HDFS (Hadoop Distributed File System)	A distributed file system designed for storing very large files across a cluster of commodity hardware. It's the storage layer of Hadoop.	Storing massive datasets reliably and efficiently. It breaks large files into smaller blocks and distributes them across many computers, with built-in fault tolerance.
MapReduce	A programming model and software framework for writing applications that process vast amounts of data in parallel across large clusters.	Processing and analyzing large datasets in a batch processing manner. It divides a big task into smaller "map" tasks and then aggregates the results with "reduce" tasks.
YARN (Yet Another Resource Negotiator)	The resource management layer of Hadoop. It separates resource management from data processing.	Managing and allocating computing resources (CPU, memory) to different applications running on a Hadoop cluster. It allows various data processing engines to coexist and run efficiently.
Sqoop	A tool designed to transfer data between Hadoop and relational databases (like MySQL, Oracle).	Importing data from traditional relational databases into HDFS, and exporting data from HDFS back to relational databases. It's used for batch data transfers.
Hive	A data warehouse system built on top of Hadoop that provides a SQL-like querying language (HiveQL).	Querying and analyzing large datasets stored in HDFS using familiar SQL commands. It converts SQL queries into MapReduce or Tez jobs. Ideal for data warehousing and analytics.

Component	What is it?	What is it used for?
Pig	A high-level platform for analyzing large datasets using a data flow language called Pig Latin.	Performing data transformations and analysis on large datasets without writing complex Java MapReduce code. It's often used by data scientists for ETL (Extract, Transform, Load) processes.
Kafka	A distributed streaming platform that acts as a publish-subscribe messaging system.	Handling real-time data feeds, building real-time streaming data pipelines, and processing real-time events. It's great for high-throughput, low-latency data ingestion.
HBase	A NoSQL, column-oriented distributed database built on top of HDFS.	Providing real-time, random read/write access to large datasets. It's suitable for applications that need fast lookups on massive tables, like online applications or real-time analytics dashboards.
Scala	A high-level, multi-paradigm programming language that runs on the Java Virtual Machine (JVM).	Developing big data applications, especially with Apache Spark, due to its conciseness, strong type system, and functional programming features. It's widely used for complex data processing.
Flume	A distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data from various sources to HDFS or other centralized data stores.	Ingesting streaming data (like log files, social media feeds, clickstreams) into Hadoop for further processing and analysis. It's an excellent tool for data collection.