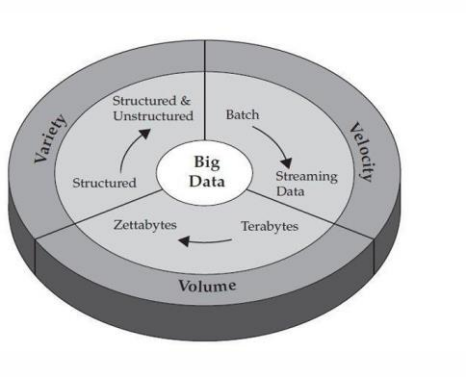
Big Data Basics: Notes on Hadoop MapReduce, Hive, and Spark

"Every note here marks the start of a new learning adventure."

Lecture 1 introduction

The most widely accepted definition of big data in terms of three characteristics volume , variety and velocity.



- **Volume:** Refers to the sheer size or magnitude of the data. For instance, large companies like Google or Facebook process billions of records and transactions daily. This creates an enormous amount of data that needs to be managed efficiently.
- **Variety:** This means data comes from many different sources and in various formats. It could be structured data (like numbers in a database), unstructured data (like images or videos), or semi-structured data (like XML files). The point is, not only is the volume of data increasing, but the different types of data are also expanding.
- **Velocity:** This refers to how quickly the data is generated and processed in real-time. Data streaming happens at lightning speed, like sensor data from smart devices or live feeds from social media platforms, which need to be analyzed and acted upon immediately.

The three types of data variety in Big Data are typically categorized as:

1. **Structured Data:**
    - Data that is organized in a predefined format, usually in rows and columns (like in a database or spreadsheet).
    - Examples: Databases, and financial transactions.
2. **Unstructured Data:**
    - Data that doesn't follow a predefined structure or model. It is more complex and harder to analyze.
    - Examples: Text documents, videos, images, social media posts, and emails.
3. **Semi-structured Data:**
    - Data that doesn't reside in a strict relational database format, but still has some organizational properties that make it easier to analyze.
    - Examples: XML files, JSON documents, and log files.

There are generally three types of data velocity in Big Data, based on how quickly data is generated and processed:

1. Batch Processing:

Data is collected, stored, and then processed in large chunks or batches at specific intervals. It doesn't require real-time analysis but is used for tasks where the data isn't time-sensitive.

- **Examples**:
    - Payroll systems that process data once a month.
    - End-of-day transaction processing in banks.

- ETL (Extract, Transform, Load) jobs that run daily to move and process data in databases.

2. Real-time Processing (or Streaming Data):

Data is processed almost instantly as it is generated. This type of velocity is used when businesses need to act on data immediately or within seconds of it being created.

- **Examples**:
  - Stock market trading systems where prices fluctuate in real-time.
  - Social media feeds, where posts, comments, and reactions are processed instantly.
  - Internet of Things (IoT) devices like smart thermostats or sensors, which collect and transmit data in real time for immediate action.

3. Near Real-time Processing:

Similar to real-time, but there might be slight delays in processing the data, typically a few seconds or minutes. It's used when quick reactions are necessary but not necessarily immediate.

- **Examples**:
  - Fraud detection systems in credit card transactions, where analysis happens within a few seconds to prevent fraudulent activity.
  - Monitoring systems for network security, where alerts are generated after brief processing delays

Big Data refers to data that is so large in scale, diverse in type, or generated at such high speed that it requires new technological approaches and architectures to process and analyze it. The goal of using these tools is to extract insights that can unlock new sources of value for businesses.

Lecture 2

💡 **Big Data: A Technology, Not a Science**

Apache Hadoop is a software framework used to store and process Big Data. It is designed to handle large volumes of structured and unstructured data across distributed clusters of computers. Key features of Hadoop include:

Hadoop's core components include **HDFS (Hadoop Distributed File System) for storage** and Big data processing engine like

-MapReduce

-Spark

-Flink

for processing, making it a powerful tool in the Big Data ecosystem.

💡 a **cluster** refers to a group of interconnected computers that work together as a single system to store and process large volumes of data

We can say that Hadoop works in a distributed and parallel manner. This is a key characteristic of Hadoop's architecture:

- **Distributed:** Hadoop distributes data storage and processing tasks across multiple computers in a cluster.
- **Parallel:** It performs computations in parallel across these distributed nodes, allowing for efficient processing of large datasets.

The processing engines mentioned earlier (MapReduce, Spark, and Flink) are designed to leverage this distributed and parallel nature of Hadoop, enabling it to handle Big Data effectively.

MapReduce vs Spark vs Flink

	MapReduce	Apache Spark	Apache Flink
Execution models	Batch processing		
		- Near real time processing	- Real time processing
Interactive analysis	No support for interactive analysis	Support interactive analysis	
Iterative processing	Not suitable for iterative processing	Suitable for iterative processing	Native iterative processing

💡 **Real-Time Data**:

- Data is processed and delivered **immediately** as it is generated

Near Real-Time Data:

- Data is processed and delivered with a **slight delay**, **Apache Spark** uses **micro-batching** for near real-time processing

> you would use
> **Apache Flink** when the **allowed latency** is lower than what **Apache Spark** can provide.
>
> if your application's **latency requirements** are stringent and you need real-time responses, **Apache Flink** is the better choice over **Apache Spark**.

💡 **Interactive Analysis**:

- **MapReduce**: Doesn't support **interactive analysis**. Once a job starts, you have to wait for it to finish without the ability to query or adjust during processing.
- **Apache Spark**: Supports **interactive analysis**, which allows users to run queries and see results quickly, making it more flexible for data exploration.
- **Apache Flink**: Similarly supports **interactive analysis**, allowing users to interact with data during processing.

💡 **Spark and Flink** provide **in-memory processing** capabilities, allowing for faster data access and reduced execution times, while **MapReduce** relies on **disk-based processing**, which can introduce significant latency.

Apache Spark and **MapReduce** are primarily **batch processing engines**, while **Apache Flink** is designed as a **streaming processing engine** but can also handle batch workloads effectively. Additionally, Spark's Structured Streaming enables it to interact with **near real-time data** processing

Generally, any system that supports **streaming processing** can also support **batch processing**.

Pre-processing tools such as **Apache Hive** and **Apache Solr** rely on **MapReduce** as a fundamental mechanism for executing operations.

When using Hive, queries written in a SQL-like language are transformed into MapReduce tasks, enabling the processing of large datasets stored in **HDFS**. In contrast, Solr is used as a search engine for data analysis, where data can be processed using **MapReduce** before being ingested into Solr, enhancing search efficiency.

processing patterns with Hadoop have evolved to meet various application needs:

1. **Interactive SQL**: Replacing MapReduce with distributed query engines like Impala and Hive on Tez allows for quick responses to SQL queries while handling large datasets.
2. **Iterative Processing**: Using tools like Spark to keep intermediate data in memory facilitates faster processing of iterative algorithms.
3. **Stream Processing**: Development of tools like Storm and Spark Streaming for real-time data processing and distributed computations.
4. **Search**: Utilizing platforms like Solr and ElasticSearch for efficient searching and indexing across large datasets stored in HDFS.

MapReduce Engine

Key Concepts of MapReduce

- Distributed computing model for large-scale data processing
- Operates in two main phases: Map and Reduce
- Designed for batch processing, not real-time analysis

Strengths and Limitations

- Strengths:
  - Highly scalable for processing massive datasets
  - Fault-tolerant architecture
- Limitations:
  - Not suitable for interactive or real-time queries
  - High latency for complex operations

MapReduce in the Hadoop Ecosystem

- Works in conjunction with HDFS for distributed storage
- Complemented by higher-level tools like Hive and Pig for easier data manipulation

Use Cases and Considerations

- Ideal for batch processing tasks like log analysis and data transformations
- Consider alternatives (e.g., Spark, Flink) for tasks requiring lower latency or iterative processing

Understanding MapReduce's role and limitations within the Hadoop ecosystem is crucial for designing effective big data processing strategies.

Stream Processing

💡 Stream processing is the continuous incorporation of new data to compute results in real-time. It deals with unbounded input data, typically a series of events arriving at the processing system (e.g., credit card transactions, website clicks, or IoT sensor readings).

Key Characteristics of Stream Processing:

- Processes data in real-time or near real-time
- Handles unbounded, continuous data streams
- Provides low-latency results
- Enables incremental computation

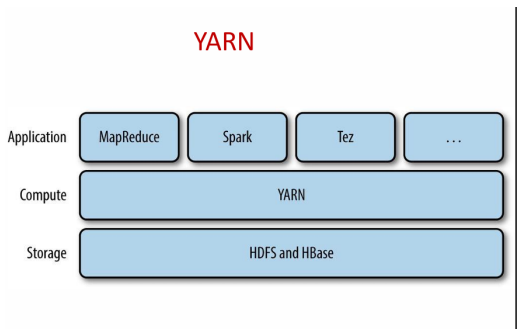
When to Use Stream Processing:

1. **Lower Latency Requirements:** When applications need quick responses (in minutes, seconds, or milliseconds), stream processing systems can maintain state in memory for better performance.
2. **Efficiency in Updating Results:** Stream processing can be more efficient than repeated batch jobs for updating results, as it automatically incrementalizes the computation.

Stream processing is crucial in scenarios where real-time insights and rapid decision-making are essential, such as fraud detection, real-time analytics, and IoT applications.

YARN (Yet Another Resource Negotiator)

YARN is a crucial component in the Hadoop ecosystem, introduced in Hadoop 2. It serves as a cluster resource management system, enabling diverse distributed programs to run on data in a Hadoop cluster.



Key Features of YARN:

- Allows any distributed program, not just MapReduce, to run on Hadoop clusters
- Acts as a resource manager for the Hadoop ecosystem
- Enables more flexible and efficient resource allocation
- Supports multiple processing models beyond MapReduce

YARN's introduction has been a game-changer for Hadoop, as it opened up the platform to a wider range of processing models and applications, significantly expanding Hadoop's versatility and usefulness in big data processing.

Importance of YARN:

- **Greater control**: Ability to choose the appropriate model based on project needs
- **Resource distribution**: Efficiently allocates resources across different applications, improving overall system performance

💡 The transition from Hadoop 1 to Hadoop 2 with YARN has transformed data handling, opening up possibilities for diverse frameworks and increasing the efficiency and flexibility of data analysis.

Hadoop 1 vs Hadoop 2 (with YARN)

Hadoop 1:

- Entirely dependent on **MapReduce** for data processing
- Limited flexibility in application types
- Resource management tightly coupled with MapReduce

Hadoop 2 with YARN:

- **YARN** acts as a resource manager, allowing for greater flexibility
- Supports multiple processing models:
- **MapReduce**: Still available for batch processing
- **Spark**: Faster, more efficient framework for data analysis and machine learning
- **Flink**: Powerful stream processing system for real-time data analysis

---

> When we say **Map reduce** we mean **Hadoop**

Difference Between Schema on Write and Schema on Read:

Schema on Write:

The schema for the data is defined before writing the data to the database. This means you specify the data types and format before storing it.
→ Traditional database systems like **MySQL** and **Oracle**.

Schema on Read:

The schema is defined only when reading the data. This means you store the data as is, and then determine how to use it when needed.

→ Big data storage systems like **Hadoop** and **NoSQL databases** (such as **MongoDB**).

MapReduce vs traditional database

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Transactions	ACID	None
Structure	Schema-on-write	Schema-on-read
Integrity	High	Low
Scaling	Nonlinear	Linear

Transactions

Definition: Transactions are a sequence of operations performed as a single logical unit of work. They ensure data integrity and consistency, especially in concurrent environments where multiple transactions might occur simultaneously.

- **Traditional RDBMS**:
  - Supports **ACID** transactions, which stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**.

> 💡 **ACID**
>
> - **Atomicity**: Ensures that all operations within a transaction are completed successfully; if one operation fails, the entire transaction fails.
> - **Consistency**: Guarantees that the database remains in a valid state before and after the transaction.
> - **Isolation**: Ensures that concurrent transactions do not interfere with each other.
> - **Durability**: Guarantees that once a transaction is committed, it remains so, even in the case of a system failure.

  - This robust transaction support is essential for applications requiring strict data consistency, such as banking systems.
- **MapReduce**:
  - Does not support traditional transactions. In this framework, data is typically written once and read many times, which is often sufficient for analytical applications.

Structure

Definition: The structure refers to how data is organized and defined within a database system, including the schema that dictates the data's format.

- **Traditional RDBMS**:
  - Uses a **schema-on-write** approach.
  - This enforces strict data organization and consistency, ensuring that all data adheres to the defined structure.
- **MapReduce**:
  - Employs a **schema-on-read** approach.
  - The schema is applied when the data is read or queried, meaning that data can be interpreted in various ways based on the requirements of the query. This flexibility is particularly advantageous in big data scenarios, where data types may vary significantly.

Integrity

Definition: Data integrity refers to the accuracy and consistency of data stored in a database, ensuring that the data remains valid and reliable over its lifecycle.

- **Traditional RDBMS**:
  - Maintains **high data integrity** due to strict schema enforcement and support for ACID transactions. This ensures that all data adheres to the defined rules and relationships, preventing invalid data entries and ensuring reliable data handling.
  - Features such as constraints (e.g., primary keys, foreign keys, and unique constraints) further enhance data integrity by restricting the type of data that can be entered.
- **MapReduce**:
  - Generally has **lower integrity guarantees** since it does not enforce a schema when data is written. This can lead to situations where the data may not conform to any particular structure, potentially resulting in inconsistencies.
  - The focus on flexibility and scalability over strict integrity allows for faster processing of vast amounts of data, but it does come with the trade-off of potential data quality issues.

Scaling

Definition: Scaling refers to the ability of a system to handle increasing amounts of work, often by adding more resources.

- **Traditional RDBMS**:
  - Scaling is typically **nonlinear**. This means that simply adding more resources (such as CPU or memory) does not necessarily lead to a proportional increase in performance, especially as data volume increases.
  - Traditional RDBMS systems can face limitations when scaling vertically (adding more power to a single server) and may require complex sharding or clustering strategies to handle larger datasets effectively.
- **MapReduce**:
  - Scales **linearly**, which means that as you add more nodes (servers) to the system, the performance increases proportionately.
  - This linear scalability makes MapReduce particularly well-suited for big data processing, allowing organizations to efficiently handle and analyze massive datasets by simply adding more hardware resources as needed.

MapReduce vs traditional database

Feature	MapReduce (Hadoop)	RDBMS (Relational Database Management System)
Type of Operations	Batch processing of large datasets.	Point queries or updates, handling small amounts of data.
Data Handling	Reads the entire dataset in each operation.	Retrieves or updates specific parts of data using indexes.
Write and Read	Write once, read many times.	Continuous reads and writes with frequent updates.
Data Type	Semi-structured or unstructured data.	Structured data with a predefined schema.
Data Organization (Normalization)	Denormalized, does not require normalization.	Requires normalization to ensure data integrity and reduce redundancy.
Handling Large Data	Scales linearly with data size and supports parallel processing.	Performance may degrade with large data sizes.
Joins	Supports joins, but they are less common compared to RDBMS.	Heavily relies on joins to retrieve data from multiple tables.
Structured Data	Can easily handle unstructured or semi-structured data.	Requires structured data with a predefined schema.
Flexibility	Flexible with "schema-on-read," interpreting data at processing time.	Requires data loading and validation before querying (schema-on-write).
Use Cases	Analyzing server logs or processing big data in batch mode.	Systems like financial databases or inventory management with frequent updates.
Technological Evolution	Systems like Hive are becoming more interactive, resembling RDBMS.	Traditional databases are incorporating features from Hadoop, such as support for unstructured data.

Lecture 3 MapReduce

💡 **Hadoop**

is a free framework designed to process big data by distributing tasks across multiple simple computers (known as a cluster).

This framework allows developers to handle large amounts of data by distributing operations across these machines and coordinating their efforts.

Hadoop is built using **Java**, but developers can also use other languages like **R**, **Python**, or **Ruby** to build applications that leverage Hadoop's capabilities.

Difference Between Hadoop 1 and Hadoop 2

Hadoop 1:

- **MapReduce**
  - It handled distributed data processing by executing **Map** and **Reduce** tasks over large datasets.
  - **MapReduce** was responsible for everything, including resource management and task scheduling.
  - **Limitations**: The system was constrained because all operations were dependent solely on **MapReduce**.

    This made it impossible to run other types of tasks or applications apart from MapReduce jobs.
- **HDFS (Hadoop Distributed File System)**:
  - A distributed file system that breaks large data into smaller chunks and distributes them across machines in the cluster.
  - In Hadoop 1, HDFS managed data storage, ensuring data availability and fault tolerance.

💡 in **Hadoop 1**, the default and only processing engine is **MapReduce**. It was responsible for both processing the data and managing resources (like scheduling tasks and allocating cluster resources).

Hadoop 2:

MapReduce

- A system for parallel processing of large data sets that implements the MapReduce model of distributed programming.

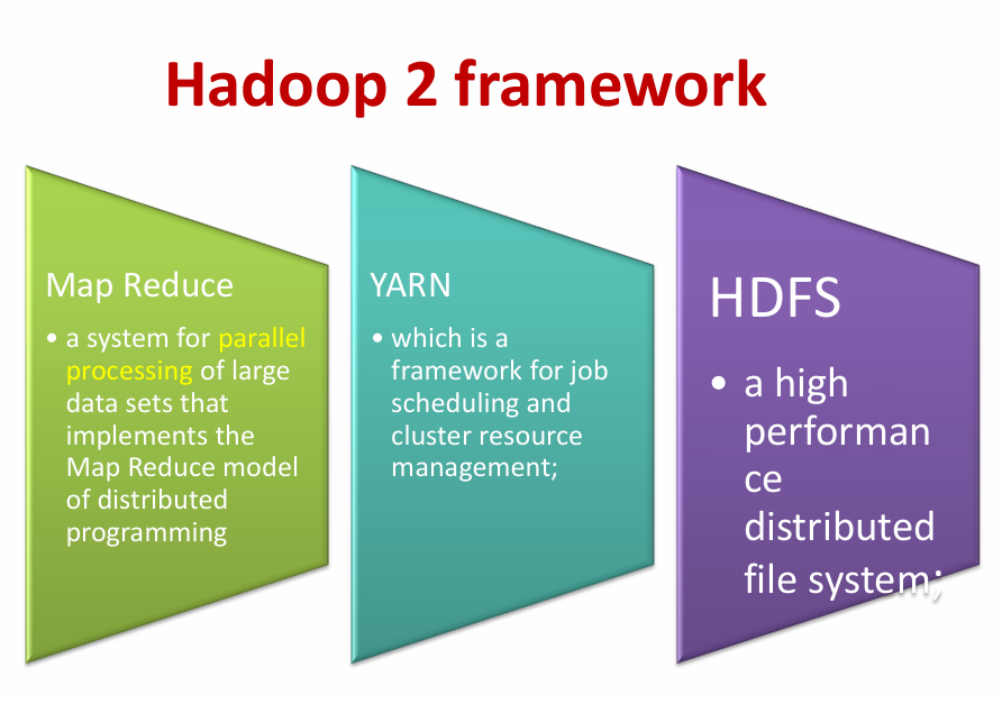
YARN

- A framework for job scheduling and cluster resource management, which separates resource management from MapReduce, increasing flexibility and allowing non-MapReduce applications like machine learning and graph processing to run.

💡 YARN is responsible for accepting tasks, allocating the resources that those tasks need, and then scheduling them to run on the devices available in the group.

HDFS

- A high-performance distributed file system with enhancements such as high availability (HA) to ensure data reliability,



Features of Hadoop

Open Source	**Reliability**	**Economic**
Distributed Processing	**High Availability**	**Easy to use**
Fault Tolerance:	**Scalability**	**Data Locality**

Economic:

The **Economic** feature of **Apache Hadoop** highlights its cost-effectiveness. Here's why:

- **Commodity Hardware**: Hadoop doesn't require expensive, specialized machines. It can run on ordinary, low-cost hardware (commodity hardware) that is readily available in the market. This reduces the overall infrastructure cost.
- **Cost Savings**: Since Hadoop uses commodity hardware, businesses can achieve massive cost savings, especially when dealing with large-scale data storage and processing.
- **Scalability Without Downtime**: One of the most valuable aspects is the ability to add more nodes (servers) to the cluster as the need for resources grows. This can be done **on the fly**, meaning you can expand the system without any downtime or service interruption, which is crucial for businesses that need continuous availability.
- **Minimal Pre-planning**: Because of Hadoop's flexibility, there's no need for extensive upfront planning for future growth. You can scale the system as the data or processing demands increase, keeping costs under control while ensuring performance.

💡 **Hadoop** uses low-cost and modest **commodity hardware**, making it ideal for handling large amounts of data at a low cost and with high flexibility.

On the other hand, **HTFC** (High-Performance Computing Systems) uses **specialized and expensive hardware** to achieve high performance in applications that require intensive computations and fast processing speed.

Open Source:

Apache Hadoop is an open source project. It means its code can be modified according to business requirements.

Distributed Processing:

As data is stored in a distributed manner in HDFS across the cluster, data is processed in parallel on a cluster of nodes

Fault Tolerance

Hadoop provides **fault tolerance** by replicating data across multiple nodes.

how it works based on your scenario:

- **Data Distribution**: A file is divided into blocks (e.g., 128 MB each) and distributed across multiple nodes in the cluster (e.g., 10 machines).
- **Replication**: Each block is stored on three nodes (one primary and two replicas) to ensure data redundancy.
- **Automatic Recovery**: If a node fails, Hadoop automatically detects the failure and retrieves data from other nodes, ensuring continuous data availability without manual intervention.

Reliability:

Due to replication of data in the cluster, data is reliably stored on the cluster of machine despite machine failures

High Availability:

Data is highly available and accessible despite hardware failure due to multiple copies of data. If a machine or few hardware crashes, then data will be accessed from another
path.

💡

Example:

1. **File Size**: Assume you have a file that is 768 MB in size.
2. **Block Size**: The file is divided into 6 blocks (each 128 MB).
3. **Block Distribution**: These blocks are distributed across a cluster of 10 nodes, and each block is replicated on 2 additional nodes for fault tolerance.
    - **Block 1**: Stored on Node 1, Node 2, Node 3
    - **Block 2**: Stored on Node 2, Node 4, Node 5
    - **Block 3**: Stored on Node 3, Node 6, Node 1
    - **Block 4**: Stored on Node 4, Node 5, Node 2
    - **Block 5**: Stored on Node 5, Node 1, Node 3
    - **Block 6**: Stored on Node 6, Node 2, Node 4

    If, for example, **Node 1** goes down, Hadoop can still access the data for **Block 1** from **Node 2** or **Node 3**, ensuring that data remains available and that processing can continue seamlessly.

    This architecture provides robust fault tolerance, making Hadoop suitable for handling large-scale data processing tasks.

Scalability in Hadoop:

Hadoop is highly scalable, meaning you can easily add new hardware (nodes) to the cluster as data or processing demands grow. This **horizontal scalability** allows you to expand the cluster by adding more nodes without causing any downtime. As a result, Hadoop can handle increased workloads and larger datasets seamlessly, making it suitable for growing data needs in a cost-effective manner.

Vertical Scalability (Database, DWH,...):

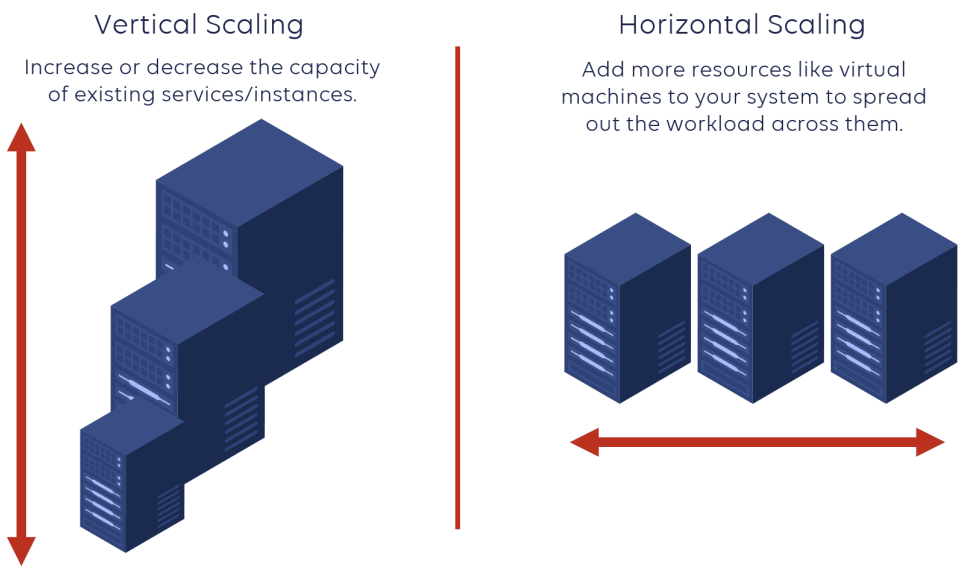
- **Definition**: Vertical scalability means improving performance by adding more resources (CPU, RAM, or storage) to a **single machine**.
- **Example**: If your database server is running slow, you can upgrade it by adding more memory or processing power to handle larger workloads.

Horizontal Scalability (Hadoop):

- **Definition**: Horizontal scalability means improving performance by adding more **machines (nodes)** to a cluster instead of upgrading a single machine.
- **Example**: If your Hadoop cluster needs more capacity, you simply add more nodes (servers) to the cluster to distribute the workload across many machines.

Key Differences:

- **Vertical Scalability**: Scaling **up** a single machine by upgrading hardware.
- **Horizontal Scalability**: Scaling **out** by adding more machines to share the load.



Easy to use:

No need of client to deal with distributed computing, the framework takes care of all the things. So this feature of Hadoop is easy to use.

📝 The **client** can be the **programmer or developer** who interacts with the Hadoop system.

Data Locality

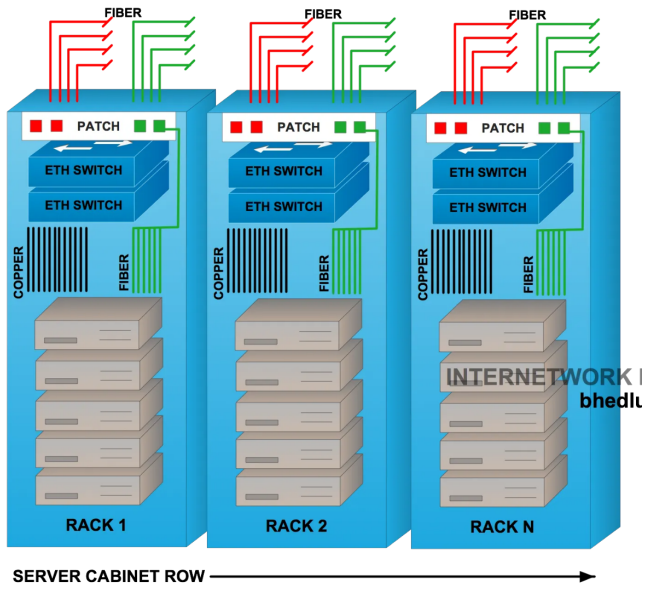
in Hadoop When you run a **MapReduce** job , instead of moving the large amount of data to the location where the program is running (which would take a lot of time and resources), Hadoop does something smart: it moves the **program** (the MapReduce code) to the **location where the data is stored**.

So instead of transferring the **data** across the network, Hadoop transfers the **code** to the nodes where the data is located.

Why is this useful?

- **Faster**: The code is smaller and can move faster than large chunks of data, saving time and resources.
- **More Efficient**: The data stays in place, reducing network traffic and making the analysis process quicker.

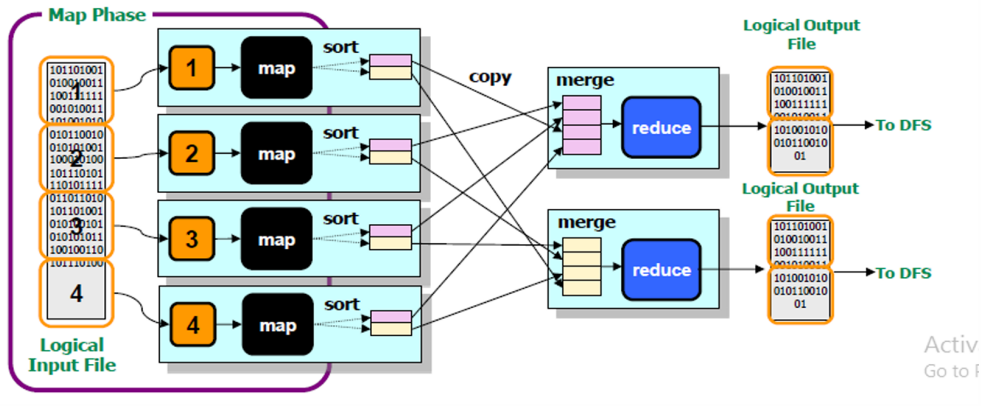
This is what **Data Locality** means: move the code to the data, not the other way around.



Example:

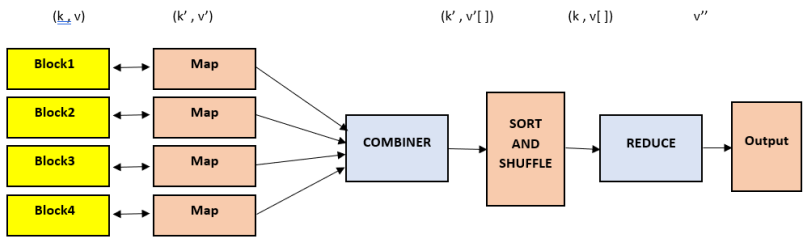
Imagine you have a huge file divided into several parts, and each part is stored on different machines in the network (cluster). Instead of moving all the parts to a single machine to process, Hadoop sends the code that processes the data to each machine where the parts are stored. Each machine then processes its part locally.

MapReduce framework



The **MapReduce** framework is a programming model designed for processing large datasets in a distributed computing environment. It consists of two main phases: **Map** and **Reduce**.

- **Phases**: MapReduce consists of two main phases:
  - **Map Phase**: Processes input data in parallel, transforming it into intermediate key-value pairs.
  - **Reduce Phase**: Aggregates and processes the intermediate key-value pairs to produce final output key-value pairs.



Key-Value Pairs: Both the input and output for each phase are in the form of key-value pairs. The types of these pairs can be specified by the programmer.

📝 In the MapReduce framework, the input to the mapper consists of key-value pairs, where:

1. **Key**: The key is typically the **offset number** of the row in the input file. This offset represents the position of the data in the file, which allows the mapper to know where each piece of data is located.
2. **Value**: The value is the actual **data in the row**.

> 📝 **offset number** is the position of a specific piece of data in a file, measured in bytes from the beginning of the file.

- **Functions**:
  - **Map Function**: Defined by the programmer to determine how input data is processed into intermediate pairs.
  - **Reduce Function**: Defined by the programmer to aggregate the intermediate pairs into final results.
- **Mapper Class**: A generic class that specifies the types for input keys, input values, output keys, and output values, providing flexibility in handling different data types.

> 📝
> - **Blocks** are the physical storage units in HDFS,
> - while **splits** are logical divisions used for processing data in the MapReduce framework.
> - When a file is uploaded to HDFS, it gets divided into blocks, and during MapReduce processing, those blocks can be further divided into splits that determine how the data is fed to the mappers.

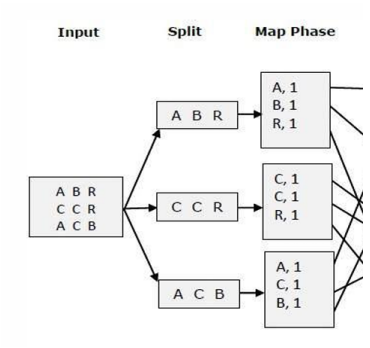
MapReduce: Mappers

- **Role of Mappers**:
  - Mappers are small programs that read portions of input data, interpret, filter, or transform this data, and produce a stream of `<key, value>` pairs.

	Input	Output
Mapper	portions of input data (Row)	`<key, value` pairs

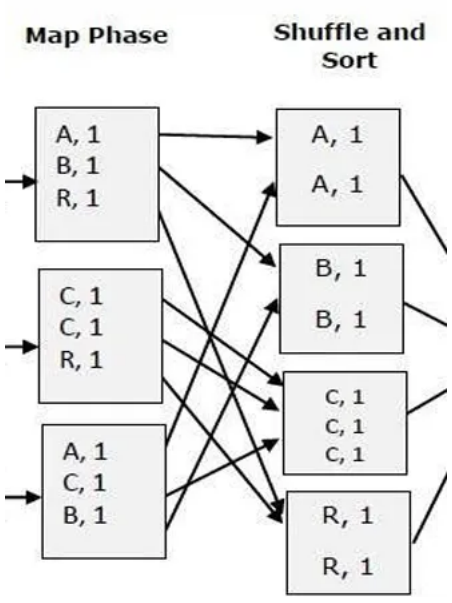
- **Data Distribution**:
  - The MapReduce framework automatically distributes the mapper program to every machine that has a block of the data being processed. This allows for parallel processing.

- **Scalability**:
  - As the file size and cluster size increase, more mappers are involved in processing, enhancing efficiency.



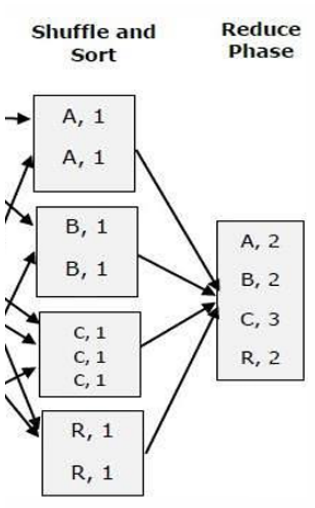
MapReduce: Shuffle Phase

- **Shuffle**: This phase occurs between the Map and Reduce phases and is responsible for redistributing the data based on the output keys of the mappers. It groups all intermediate values associated with the same key so they can be processed by the reducer.



MapReduce: Reducers

- **Role of Reducers**:
  - Reducers are small programs that sort and/or aggregate values associated with the assigned keys.
  - **Parallelism**:
    - The number of unique keys determines the level of parallelism during the Reduce phase.



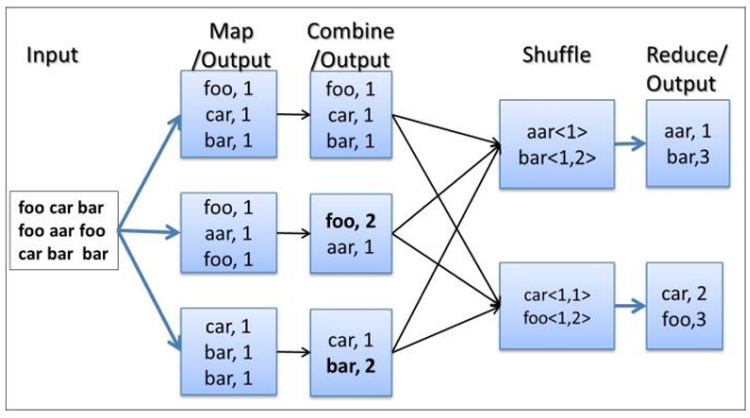
> 📝 More unique keys lead to more reducers working in parallel.

- **Output**:
  - After completing their tasks (e.g., summing total sales), reducers emit key-value pairs that are written to storage for future MapReduce jobs.

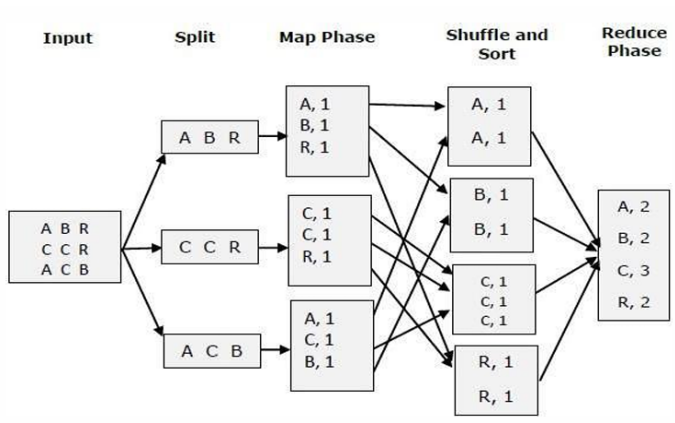
MapReduce: Combiners

- **Purpose of Combiners**:
  - To minimize the amount of data shuffled between map and reduce tasks, thus improving efficiency. Combiners act as mini-reducers that aggregate map outputs before sending them to reducers.
- **User-defined Function**:
  - Similar to the reduce function, users can specify a combiner function that runs on the output of the map tasks, reducing the volume of data transferred.

MapReduce : word count example **with combiner**



MapReduce : word count example **without combiner**



⚠
- **Combiners**: Work to reduce the output data from each Mapper before sending it to the Reducers.
- **Reducers**: Handle the aggregated data from all Mappers after the Shuffle phase to generate the final output.

📝
- **Mapper**: Transforms input data into intermediate key-value pairs.
- **Shuffle**: Groups intermediate data by keys.
- **Reducer**: Aggregates grouped data into final key-value pairs.
- **Combiner (optional)**: Performs partial aggregation to reduce data before it reaches the reducer.

Data Flow in MapReduce

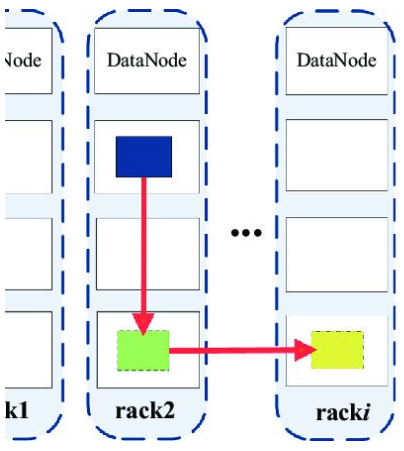
1. **Job Definition**:
   - A MapReduce job is a unit of work that includes **input data**, the **MapReduce program**, and **configuration information.**
2. **Task Division**:
   - Hadoop divides the job into two types of tasks: **map tasks** and **reduce tasks**.
   - These tasks are scheduled using YARN (Yet Another Resource Negotiator) and executed on nodes in the cluster. If a task fails, it is automatically rescheduled on a different node.
3. **Input Splits**:
   - Input data is divided into fixed-size pieces called **input splits** (or simply splits).
   - A map task is created for each split, which processes records using a user-defined map function.
4. **Parallel Processing**:
   - The advantage of having many splits is that the processing time for each split is reduced, allowing for better load balancing when processed in parallel.
   - A faster machine can process more splits than a slower machine, improving overall efficiency.
5. **Load Balancing**:
   - Even with identical machines, load balancing is essential due to potential failures or concurrent jobs.
   - Fine-grained splits enhance load balancing, but if they are too small, the overhead of managing these splits may outweigh the benefits.
6. **Optimal Split Size**:
   - The recommended split size is typically the size of an HDFS (Hadoop Distributed File System) block, which is 128 MB by default.
   - This size can be adjusted at the cluster level or specified for individual files.

Data Locality in Hadoop

Running Tasks on Local Data:

- Hadoop aims to execute **map tasks** on nodes where the input data is located in the **Hadoop Distributed File System (HDFS)**.

  This minimizes the use of cluster bandwidth and speeds up processing.
- If all nodes hosting the relevant HDFS block replicas are busy, the job scheduler will look for available slots on nodes within the **same rack** as the blocks. If no local options are available, it may resort to using **off-rack nodes**, resulting in inter-rack data transfers.



Optimal Split Size:

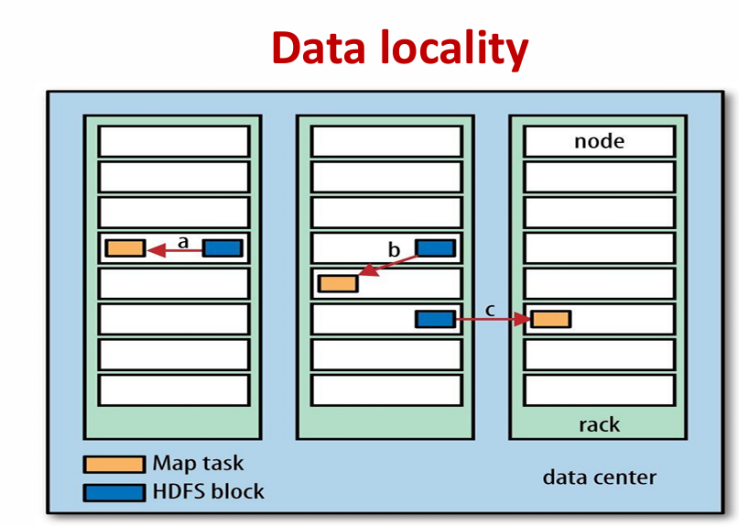
- The **optimal split size** in Hadoop is aligned with the block size of HDFS. This is because the split size should match the largest input size guaranteed to be stored on a single node.
- If a split spans multiple blocks, there's a risk that no single node holds all the data, leading to inefficient data transfer across the network.

Map Output Storage:

- **Map tasks** write their output to the local disk rather than directly to HDFS. This is because the output is intermediate data, which is further processed by **reduce tasks** to generate the final output. Storing intermediate data locally speeds up the process by reducing the need for network transfers.

Data Locality Levels:

- **Data-local** (Task runs on the same node as the data)
- **Rack-local** (Task runs on a node in the same rack as the data)
- **Off-rack** (Task runs on a node in a different rack from the data)



Data-local (a), rack-local (b), and off-rack (c) map tasks

1. **Reduce Tasks**:
   - Unlike map tasks, **reduce tasks** do not benefit from data locality. The input for a reduce task typically comprises outputs from all the mappers, which means they must often gather data from multiple nodes, potentially leading to increased latency.

Lecture 4 MapReduce

Reading Data in MapReduce

Hadoop can process a variety of data formats, ranging from flat text files to databases.

There are three main Java classes provided in Hadoop to read data in MapReduce:

InputSplitter

The InputSplitter divides the data into smaller units known as **"splits."** Each split is assigned to an individual Mapper task for processing. The InputSplitter determines the number of splits generated and how they are distributed across the tasks.

RecordReader

The RecordReader reads data from each split and converts it into **key-value pairs** that can be used by the Mapper. It prepares the data in a format suitable for processing within the MapReduce framework.

InputFormat

The InputFormat defines the type of data and the processing method. Different types of InputFormats are available, such as TextInput Format and KeyValueTextInputFormat, and the choice depends on the data format.

InputSplitter: Splits data into smaller chunks for distribution across multiple tasks.

RecordReader: Converts splits into key-value pairs.

InputFormat: Specifies the data type and how to read it.

MapReduce : InputFormat

What is Hadoop InputFormat?

Hadoop InputFormat describes the input-specification for execution of the Map-Reduce job.

InputFormat describes how to split up and read input files. In MapReduce job execution, InputFormat is the first step. It is also responsible for creating the input splits and dividing them into records.

Types of InputFormat in MapReduce:

- TextInputFormat
- KeyValueTextInputFormat
- NlineInputFormat
- SequenceFileInputFormat

InputFormat	Description	Key	Value	File Type
TextInputFormat	Default format that reads lines of text from input files.	Byte offset of each line.	Contents of the line.	Text files
KeyValueTextInputFormat	Parses each line into key-value pairs based on a tab character.	Everything up to the first tab.	Remainder of the line after the tab.	Text files
NLineInputFormat	Assigns each mapper a fixed number of lines.	Byte offset of each line.	Line contents.	Text files
SequenceFileInputFormat	Reads Hadoop-specific high-performance binary format files.	User-defined.	User-defined.	Binary

1. TextInputFormat in Hadoop MapReduce

It is the default InputFormat. This InputFormat treats each line of each input file as a separate record.

Record Structure

> ***Each record corresponds to a single line of input.***
>
> The **key** is represented as a **LongWritable**, indicating the **byte offset** within the file at which the line begins.
>
> The **value** is the line's contents, stored as a **Text** object, excluding any line terminators (like newline or carriage return).

💡 TextInputFormat is useful for unformatted data or line-based records like log files.

⚠ In Hadoop, the data received and sent to and from the mapper and reducer needs to be of Hadoop Data Types such as Text, IntWritable, LongWritable, and others.
The reason for using these Hadoop types is that they are designed to be Serializable, so that they can be partitioned and sent over the network in the distributed system, and this ensures high performance and efficiency when transferring data between different nodes in the Hadoop system.

In a Hadoop environment, Serialization and Deserialization are vital, especially when moving data between Mapper and Reducer.

Hadoop is built on Java, but using traditional data types in Java can negatively impact performance due to limitations associated with Serialization.

Solutions and techniques used:
Using custom types in Hadoop LongWritable, IntWritable, Text and other custom types in Hadoop are specifically designed to improve performance.
These types provide more efficient serialization and help reduce the size transferred over the network.

- **Serialization**: This is the process of converting data into a format that can be stored or transmitted. In the context of Hadoop, data is converted from memory to a format that can be sent between the **Mapper** and **Reducer**.
- **Deserialization**: This is the reverse process, where data is converted from the stored format back into its original form in memory.

Interprocess Communication: Nodes in a Hadoop cluster communicate with each other using **Remote Procedure Calls (RPCs)**, which rely on serialization to convert messages into binary form for efficient transmission.

Persistent Storage: Data is serialized when written to the Hadoop Distributed File System (HDFS) or other persistent storage, making it accessible for future processing and reconstruction.

Integer Types

Java: `int`

Hadoop: `IntWritable`

Text/String Types

Java: `String`

Hadoop: `Text`

Example:
For a file containing the following text:

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

This text would be divided into records interpreted as:

- **(0, On the top of the Crumpetty Tree)**
- **(33, The Quangle Wangle sat,)**
- **(57, But his face you could not see,)**
- **(89, On account of his Beaver Hat.)**

the numbers in parentheses represent the byte offsets of the beginning of each line.

🔑 **Byte Offset**

The byte offset indicates the number of characters from the start of the file to the beginning of each line.

- For instance, the offset for the first line (0) means it starts at the very beginning of the file, while the second line (33) starts after 33 characters have been read.

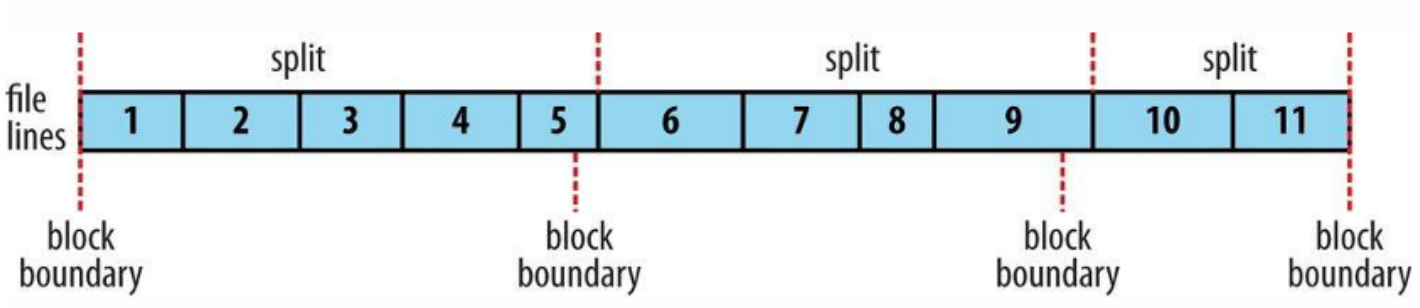
Input Splits and HDFS Blocks

- When a file is processed, it is broken into lines, but these line boundaries may not align with **HDFS block boundaries**.
- Input splits honor logical record boundaries (like lines), which means a split can span multiple HDFS blocks.

💡 **Blocks** are the physical storage units in HDFS,
while
splits are logical divisions used for processing data in the MapReduce framework.

Example Scenario:

- If a file is divided such that the first split contains line 5 (starting at a byte offset), it may span across the first and second blocks of HDFS.
- The second split would then start with line 6, ensuring that each split contains complete lines, regardless of how they align with the underlying block structure.



Note:

- This information is often illustrated with diagrams in Hadoop's definition guides, showing how input splits and HDFS blocks interact during data processing.

---

2. Key-value input format

is an input format used for reading data that represents key-value pairs, where each line in the file can be split into a key and a value based on a specified delimiter (such as a tab character). This format is ideal when the file structure includes one key-value pair per line, separated by a character like a tab.

Example:

Given an input file where `→` represents a horizontal tab character:

```
line1→On the top of the Crumpetty Tree
line2→The Quangle Wangle sat,
line3→But his face you could not see,
line4→On account of his Beaver Hat..
```

How It Works:

For each line:

- The text before the tab (`→`) becomes the **key**.
- The text after the tab becomes the **value**.

So, `KeyValueTextInputFormat` would read the file as follows:

```
(line1, On the top of the Crumpetty Tree)
(line2, The Quangle Wangle sat,)
(line3, But his face you could not see,)
(line4, On account of his Beaver Hat..)
```

3. NLineInputFormat input format

It is another form of TextInputFormat where **the keys are byte offset of the line.** And **values are contents of the line.**

So, each mapper receives a variable number of lines of input with TextInputFormat and KeyValueTextInputFormat.

The number depends on the size of the split. Also, depends on the length of the lines. So, if want our mapper to receive a fixed number of lines of input, then we use NLineInputFormat.

The keys and values are the same as those that TextInputFormat produces.

💡 N- It is the number of lines of input that each mapper receives.

By default (N=1), each mapper receives exactly one line of input.

Suppose N=2, then each split contains two lines. So, one mapper receives the first two Key-Value pairs. Another mapper receives the second two key-value pairs.

the input format allows each mapper to receive a fixed number of lines of input, unlike `TextInputFormat` or `KeyValueTextInputFormat`, where the number of lines per mapper can vary based on split size and line length.

How It Works:

- If `N` is set to 2, then each split will contain two lines.
- For example, with the following input lines:

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

  - The first mapper receives:

```
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
```

  - The second mapper receives:

```
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)
```

💡 **Difference from TextInputFormat**:

`NLineInputFormat` differs in how it constructs splits. Instead of varying based on file size, it uses a specified number of lines per split, which allows for more predictable processing loads per mapper.

one of the advantages of using
`NLineInputFormat` is that it can help ensure a more balanced distribution of data across mappers by setting a fixed number of lines for each mapper.

Binary : input format

4. SequenceFileInputFormat

It is an InputFormat which reads sequence files. Sequence files are binary files. These files also store sequences of binary key-value pairs. These are block-compressed and provide direct serialization and deserialization of several arbitrary data. Hence,

Key & Value both are user-defined.

This format is :

- **Splittable**: Sequence files contain sync points, enabling readers to synchronize with record boundaries from any point in the file, making it easy to split files for parallel processing.
- **Compression Support**: The format supports compression, allowing for efficient storage and faster processing.
- **Arbitrary Data Types**: Sequence files can store arbitrary data types using various serialization frameworks, providing flexibility in how data is structured.

the
Binary Input Format allows for the processing of non-textual data.

What is more difficult for hadoop, a small number of large files or a large number of files with a small file size?

For Hadoop, a large number of files with a small size is the hardest, if you have a large number of files but a small size, this is a bigger challenge on Hadoop compared to a few large files.

When you have a large number of small files in Hadoop, it poses several challenges primarily related to the NameNode and its memory usage

NameNode

The NameNode is referred to as the master node in the Hadoop architecture. It is responsible for managing the overall structure and metadata of the Hadoop file system. It directs the data flow and keeps track of where each file's blocks are stored. If the NameNode goes down, the entire Hadoop system cannot function because applications rely on it to locate files.

DataNode

DataNodes are the slave nodes in the architecture. They are responsible for storing the actual data blocks. They operate under the control of the NameNode and respond to its requests. Each DataNode regularly communicates with the NameNode to report its status and the blocks it stores.

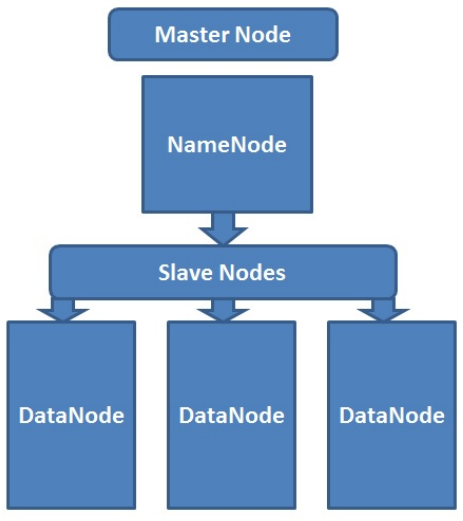
*The **NameNode** manages information about where data is located, while the **DataNode** stores the actual data.*

How Does the **NameNode** Help with **Data Locality**?

Suppose you have a large file distributed across three DataNodes:

- **DataNode A** contains Block 1 and Block 2.
- **DataNode B** contains Block 3.
- **DataNode C** contains Block 4.

When executing a MapReduce operation, if you have a task that requires reading Block 1 and Block 2, the NameNode will assign the task to DataNode A, where the required data is stored locally, thereby reducing processing time.



what problems does the SequenceFile try to solve ?

For HDFS

Small File Problem:

- **Issue**: In Hadoop, small files (files significantly smaller than the HDFS block size, typically 128 MB) create inefficiencies. Each file, directory, and block in HDFS is represented as an object, consuming approximately 150 bytes of memory in the NameNode.
    - **Example**: For instance, having **10 million small files** can consume about **3 GB** of memory on the NameNode. This excessive memory usage leads to scalability issues; handling **a billion files** becomes unfeasible due to limited resources.

For MapReduce

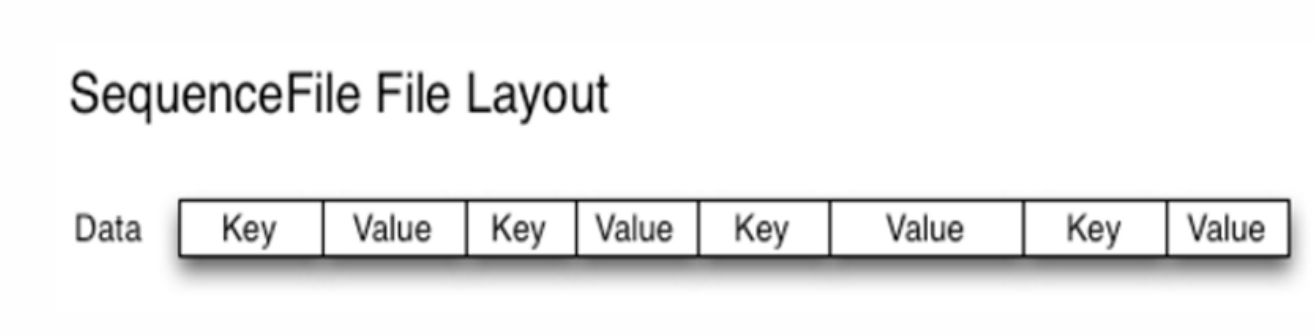
Processing Overhead:

- **Issue**: In a standard MapReduce job, each file processed typically corresponds to a map task. When there are many small files, this results in a large number of map tasks, leading to increased overhead and longer job execution times.
    - **Example**: If there are **10,000 small files**, and each file triggers a separate map task, the job could take significantly longer to execute due to task management overhead.

how can SequenceFile help to solve the problems?

Concept of SequenceFile

- The **SequenceFile** format is designed to aggregate multiple small files into a larger single file. This is beneficial in scenarios where there are numerous small files, as it reduces the overhead associated with file management in Hadoop.



Example

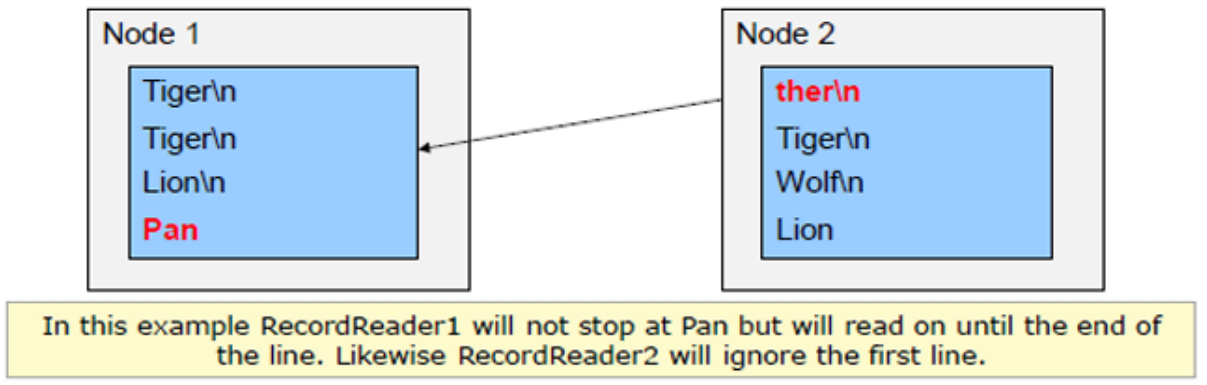
- Consider a scenario with **10,000 files**, each **100 KB** in size. Using SequenceFile, these can be combined into a single larger file, where the **filename** serves as the **key** and the **content** of each file acts as the **value**.

Benefits of Using SequenceFile

1. **Reduced Memory Usage on NameNode**:
    - Combining 10,000 small files into a 1 GB SequenceFile reduces the NameNode's memory requirement from approximately 4.5 MB to about 3.6 KB, alleviating pressure and enhancing cluster performance.
2. **Splittability**:
    - SequenceFiles are splittable, allowing them to be processed in smaller chunks. This makes them ideal for MapReduce jobs, enabling efficient handling of large datasets through parallel processing.
3. **Compression Support**:
    - SequenceFiles support compression, which decreases storage needs and improves I/O performance by minimizing the data transferred between disk and memory, resulting in faster job execution times.

MapReduce : RecordReader

- Most of the time a split will not happen at a block end
- Files are read into Records by the RecordReader class
  - Normally the RecordReader will start and stop at the split points.
- LineRecordReader will read over the end of the split till the line end.
  - HDFS will send the missing piece of the last record over the network
- Likewise LineRecordReader for Block2 will disregard the first incomplete line



MapReduce Word Count Implementation

1. Mapper Class (`wordcountmapper`)

The mapper reads lines of text, splits them into words, and outputs each word with a count of 1.

- **Code Overview**:
  - **Inputs**: Each line from the file is processed as text.
  - **Splitting**: The line is split into words using `split(" ")`.
  - **Outputs**: For each word, it writes `(word, 1)` to the context, meaning that each word is counted once.

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

public class wordcountmapper extends Mapper<Object, Text, Text, IntWritable> {
    @Override
    protected void map(Object key, Text value, Context context) throws IOExceptio
n, InterruptedException {
        String mytext = value.toString();
        String[] allwords = mytext.split(" ");
        for (String x : allwords) {
            context.write(new Text(x), new IntWritable(1));
        }
    }
}
```

- **Generic Types**:
  - Input key: `Object`
  - Input value: `Text` (line of text)
  - Output key: `Text` (word)
  - Output value: `IntWritable` (count)

2. Reducer Class (`wordcountreducer`)

The reducer aggregates the counts for each word emitted by the mapper and outputs the total count for each word.

- **Code Overview**:
  - **Inputs**: It receives each unique key (word) and a collection of values (the counts).
  - **Summing**: It sums the counts using a `for` loop.
  - **Outputs**: It writes `(word, total_count)` to the context.

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class wordcountreducer extends Reducer<Text, IntWritable, Text, IntWritabl
e> {
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context contex
t) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable iw : values) {
            sum += iw.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

- **Generic Types**:
  - Input key: `Text` (word)
  - Input value: `IntWritable` (individual counts)
  - Output key: `Text` (word)
  - Output value: `IntWritable` (total count)

3. Driver Class (`wordcountdriver`)

The driver sets up the MapReduce job, configures input and output paths, and initiates the job execution.

- **Code Overview**:
  - **Configuration**: We create a configuration and initialize a job.
  - **Setting Classes**: We specify the Mapper and Reducer classes to use.
  - **Setting Output Types**: We define the output types for keys and values.
  - **Input and Output Paths**: We set the input path (where the data is located) and the output path (where results will be stored).
  - **Running the Job**: The job is executed, and it returns 0 if successful and 1 if it fails.

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

public class wordcountdriver {
    public static void main(String[] args) throws IOException, InterruptedExc
eption, ClassNotFoundException {
        Configuration c = new Configuration();
        Job j = Job.getInstance(c, "mywordcount");
        j.setMapperClass(wordcountmapper.class);
        j.setReducerClass(wordcountreducer.class);
        j.setJarByClass(wordcountdriver.class);
        j.setOutputKeyClass(Text.class);
        j.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(j, new Path("hdfs://localhost:8020/user/
cloudera/input/data.dat"));
        FileOutputFormat.setOutputPath(j, new Path("hdfs://localhost:8020/use
r/cloudera/2019c"));
        System.exit(j.waitForCompletion(true) ? 0 : 1);
    }
}
```

the **Driver class** is the **starting point** for running a MapReduce program. It sets up everything the program needs to operate and begins the processing task.

A **JAR file** (short for Java ARchive) is a packaged file format used to aggregate many Java class files and associated resources (like images, libraries, metadata) into a single file, usually for distribution and deployment.

Sample Input

Let's say we have a text file called `data.txt` with the following content:

```
Hello World
Hello Hadoop
Hello MapReduce
```

Step 1: Mapper Phase

Input to Mapper: The input to the mapper is read line by line. For our example, the input to the mapper will be:

1. Line 1: `Hello World`
2. Line 2: `Hello Hadoop`
3. Line 3: `Hello MapReduce`

Mapping Process:

- For each line, the mapper splits the line into words and outputs each word with a count of `1`.

Mapper Output:

- For Line 1 (`Hello World`):
  - Output: `("Hello", 1)`
  - Output: `("World", 1)`
- For Line 2 (`Hello Hadoop`):
  - Output: `("Hello", 1)`
  - Output: `("Hadoop", 1)`
- For Line 3 (`Hello MapReduce`):
  - Output: `("Hello", 1)`
  - Output: `("MapReduce", 1)`

Final Mapper Output

The combined output from all the mapper tasks looks like this:

```
("Hello", 1)
("World", 1)
("Hello", 1)
("Hadoop", 1)
("Hello", 1)
("MapReduce", 1)
```

Step 2: Shuffle and Sort Phase

The framework then groups the output of the mappers by key (word). The output after shuffling and sorting would look like:

```
("Hadoop", [1])
("Hello", [1, 1, 1])
("MapReduce", [1])
("World", [1])
```

Step 3: Reducer Phase

Input to Reducer: The reducer takes the grouped output from the shuffle and sort phase.

Reducing Process:

- For each unique key (word), the reducer sums up the values (counts).

Reducer Output:

- For `("Hadoop", [1])`:
  - Output: `("Hadoop", 1)`
- For `("Hello", [1, 1, 1])`:
  - Sum = 3
  - Output: `("Hello", 3)`
- For `("MapReduce", [1])`:
  - Output: `("MapReduce", 1)`
- For `("World", [1])`:
  - Output: `("World", 1)`

Final Output

The final output of the reducer would be:

```
("Hadoop", 1)
("Hello", 3)
("MapReduce", 1)
("World", 1)
```

Why & where Hadoop is used / not used?

What Hadoop is Good For

1. **Massive Amounts of Data Through Parallelism**:
    - Hadoop is designed to process vast amounts of data by distributing the workload across multiple nodes in a cluster. This parallel processing capability allows organizations to analyze large datasets efficiently.
2. **A Variety of Data Types**:
    - Hadoop can handle various data formats, including structured (like relational databases), unstructured (like text files), and semi-structured data (like JSON and XML). This flexibility makes it suitable for diverse applications, from data warehousing to big data analytics.
3. **Inexpensive Commodity Hardware**:
    - Hadoop runs on commodity hardware, making it cost-effective. Organizations can scale their storage and processing power without investing in expensive infrastructure, which lowers the total cost of ownership.

What Hadoop is Not Good For

1. **Not Suitable for Processing Transactions (Random Access)**:
    - Hadoop is designed for batch processing rather than real-time transaction processing. It is not optimized for random access patterns, making it unsuitable for applications that require immediate feedback or updates.
2. **Not Good When Work Cannot Be Parallelized**:
    - If a task is inherently sequential and cannot be divided into smaller parallelizable units, Hadoop's performance benefits diminish. In such cases, traditional processing approaches might be more efficient.
3. **Not Good for Low Latency Data Access**:
    - Hadoop is not ideal for applications that require quick responses, such as real-time data querying. The architecture is built for batch processing, which typically introduces latency.
4. **Not Good for Processing Lots of Small Files**:
    - Hadoop's efficiency decreases when dealing with a large number of small files. Each file introduces overhead in terms of metadata management, which can slow down processing times. As mentioned earlier, formats like **SequenceFile** can help mitigate this issue, but it still remains a challenge.
5. **Not Good for Intensive Calculations with Little Data**:
    - For tasks that involve heavy computation but deal with small amounts of data, Hadoop may not be the best choice. The overhead associated with distributing tasks and managing resources can outweigh the benefits of using a distributed system for smaller datasets.

Lecture 5 MapReduce features

Using *MapReduce* involves handling data distributed across multiple nodes in a cluster.

example, you have two datasets:

1. **Dataset 1 (Customer Details)**: Contains customer names with customer IDs (`fn, customer id`).
2. **Dataset 2 (Transaction Details)**: Contains transaction amounts with customer IDs (`amount, customer id`).

Our goal is to join these datasets to retrieve the customer name along with their transaction amount using *MapReduce*.

> **Side data** can be defined as extra read-only data needed by a job to process the main dataset.
>
> The challenge is to make side data available to all the map or reduce tasks (which are spread across the cluster) in a convenient and efficient fashion.

Side Data: `fn, customer id` is considered side data—data not part of the main dataset but needed to complete the join operation with `amount, customer id`.

In *MapReduce*, it's standard to process a single dataset as input, even if it's divided across multiple files, because all files are treated as part of the same dataset and processed as a single input.

However, when dealing with **two independent datasets** (such as customer details and transaction data in example), it's challenging to load both datasets simultaneously

The optimal solution is to use

one dataset as the primary input in the *Map* phase and treat the other as *Side Data*, loading it through *Distributed Cache or Job Configuration*

This allows each node to look up the side data locally for joining operations without overloading the memory or shuffle phases.

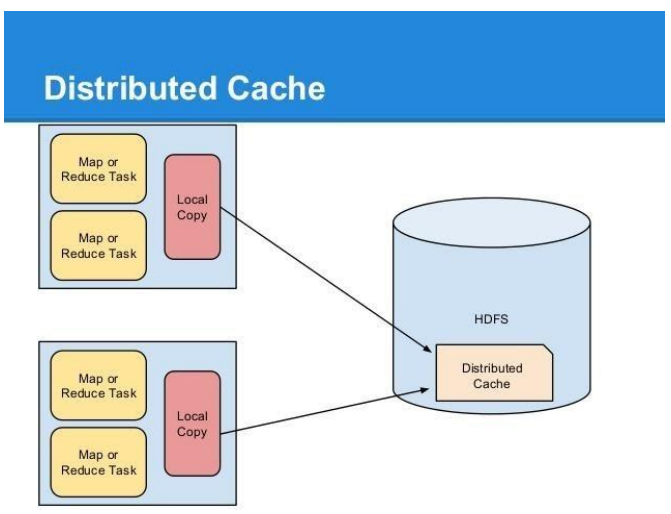
Hadoop MapReduce have two techniques :-

- **Using the Job Configuration**
- **Using Distributed cache**

---

1. Distributed cache

- This provides a service for copying files and archives to the task nodes in time for the tasks to use them when they run. To save network bandwidth, files are normally copied to any particular node once per job.
- Hadoop Distributed Cache is a facility provided by the Hadoop MapReduce framework.
- It cache files when needed by the applications. Once we have cached a file for our job, Hadoop will make it available on each data nodes where map/reduce tasks are running. Thus, we can access files from all the data nodes in our map and reduce job.
- **Lookup** files are usually passed to MapReduce jobs using distributed cache technique .



The process of adding file to distributed cache as follows :-

1. Copy requested file to HDFS.
2. Setup application job configuration .

- MapReduce job will copy the cache file on all the nodes **before** starting of tasks on those nodes.

  The file is said to be **localized** at this point.

The default size of Hadoop distributed cache is **10 g** and it is possible to control size of distributed cache in mapred-site.xml file.

1. Copy the File to HDFS:

First, you copy the file that you want to add to the **distributed cache** into **HDFS** (Hadoop Distributed File System). HDFS is where Hadoop stores data distributed across multiple servers (or nodes).

2. Set Up Job Configuration:

Next, you configure your **MapReduce job** to specify that the file should be included in the **distributed cache**.

- After this configuration, the MapReduce job will automatically distribute the file to all the nodes involved in the job **before** the tasks on those nodes start.
- Once the file is copied to the nodes, it's considered **localized**, meaning it's now available on the local machines that will run the mappers or reducers.

In our example

we need to use the **distributed cache** to send the **Customer Details** dataset (which contains the `customer_id` and `fn` or customer name) to the **reducer**.

Why should the file be sent to the reducer and not the mapper?

- In this case, the **aggregation** (summing the amounts) typically happens in the **reducer** because the **mapper** just processes and passes the data (like extracting `customer_id` and `amount`), while the **reducer** is responsible for grouping the data by `customer_id` and performing the aggregation.

After aggregating the total amount for a given `customer_id`, do we need the `customer_id` or the `fn` (customer name)?

- After aggregating the **total amount** for a particular **customer_id**, we need the **fn** (customer name) to display along with the total amount.
- The **fn** is found in **Dataset 1** (the customer details), and we use the **distributed cache** to ensure this dataset is available to the **reducer**, so it can look up the customer name corresponding to each `customer_id`.

Customer details

Cust ID	First Name	Last Name	Age	Profession
4000001	Kristina	Chung	55	Pilot
4000002	Paige	Chen	74	Teacher
4000003	Sherri	Melton	34	Firefighter
4000004	Gretchen	Hill	66	Engineer
.....	.....	.....	.....	.....

Transactional details

Trans ID	Date	Cust ID	Amount	Game Type	Equipment	City	State	Mode
0000000	06-26-2011	4000001	40.33	Exercise & Fitness	Cardio Machine Accessories	Clarksville	Tennessee	credit
0000001	05-05-2011	4000002	198.44	Exercise & Fitness	Weightlifting Gloves	Long Beach	California	credit
0000002	06-17-2011	4000002	5.58	Exercise & Fitness	Weightlifting Machine Accessories	Anaheim	California	credit
0000003	06-14-2011	4000003	198.19	Gymnastics	Gymnastics Rings	Milwaukee	Wisconsin	credit
0000004	12-28-2011	4000002	98.81	Team Sports	Field Hockey	Nashville	Tennessee	credit
0000005	02-14-2011	4000004	193.63	Outdoor Recreation	Camping & Backpacking & Hiking	Chicago	Illinois	credit
0000006	10-17-2011	4000005	27.89	Puzzles	Jigsaw Puzzles	Charleston	South Carolina	credit
.....	.....	.....	.....	.....	.....	.....	.....	.....

Reducer code : setup method

- Setup method is basically a init function, for a given Mapper/Reducer a implementation can be added to initialize/set resources to be used in map or reduce phase.
    - For example, in the **Mapper** or **Reducer**, if you need to access data from the **distributed cache**, you can do so in the **setup** method to make sure the data is ready before actual processing starts.

```
public class reducecustomer extends Reducer<Text, FloatWritable, Text, FloatWritable>
    public URI[] files;
    public HashMap<String, String> index = new HashMap<>();
    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        try {
            files = DistributedCache.getCacheFiles(context.getConfiguration());
            Path path = new Path(files[0]);
            FileSystem fs = FileSystem.get(context.getConfiguration());
            FSDataInputStream in = fs.open(path);
            BufferedReader br = new BufferedReader(new InputStreamReader(in));
            String line = "";
            while ((line = br.readLine()) != null) {
                String splits[] = line.split(",");
                index.put(splits[0], splits[1]);
            }
            br.close();
            in.close();
        } catch (IOException ex) {
            Logger.getLogger(reducecustomer.class.getName()).log(Level.SEVERE, null,
            ex.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }}
```

Reducer code : reduce method

```
@Override
    protected void reduce(Text key, Iterable<FloatWritable> values, Context context)
        float sum = 0;
        for (FloatWritable iw : values) {
            sum += iw.get();
        }
        String customername = index.get(key.toString());
        context.write(new Text(customername), new FloatWritable(sum));
    }
}
```

File Reference Count:

The **Node Manager** (which manages the nodes where tasks run) tracks the number of tasks that are using a given file from the **distributed cache**.

Each time a **Mapper** or **Reducer** accesses a file in the distributed cache, the **reference count** for that file is **incremented by 1**. This tells the system that the file is in use by a task.

- **Before the Task Runs**:
  - The reference count is incremented as soon as a task is about to use the file.
  - This ensures that the system knows how many tasks are using the file, and the file won't be deleted prematurely while it's still in use.
- **After the Task Runs**:
  - Once the task completes, the reference count for that file is **decreased by 1**.
  - The file remains in the cache as long as its reference count is greater than zero (i.e., it's still in use by other tasks).
- Only when the file is not being used (when the count reaches zero) is it eligible for deletion

```
 * @author cloudera
 */
public class mydriver {

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

        Configuration c = new Configuration();
        c.set("fs.defaultFS", "hdfs://localhost:8020");
        Job j = Job.getInstance(c, "distrib");
        j.addCacheFile(new Path("hdfs://localhost:8020/user/cloudera/distri/customer/cust_details").toUri());
        j.setMapperClass(mapcustomer.class);
        j.setReducerClass(reducecustomer.class);
        j.setJarByClass(mydriver.class);
        j.setOutputKeyClass(Text.class);
        j.setOutputValueClass(FloatWritable.class);
        FileInputFormat.addInputPath(j, new Path("hdfs://localhost:8020/user/cloudera/distri/customer/transaction_details"));
        FileOutputFormat.setOutputPath(j, new Path("hdfs://localhost:8020/user/cloudera/mydistribuetd"));
        Path out = new Path("hdfs://localhost:8020/user/cloudera/mydistribuetd");
        out.getFileSystem(c).delete(out);
        System.exit(j.waitForCompletion(true) ? 0 : 1);

    }
```

---

2. Using the Job Configuration

What can you do with the job configuration?

1. **Setting Key-Value Pairs**:
   - You can set arbitrary key-value pairs (like a small note or setting) in the configuration for your job. Think of it as giving your MapReduce job some instructions or parameters.
   - Example: You might want to pass a setting like `key = "maxAmount"` and `value = "1000"`, telling your job that the maximum amount allowed is 1000.
2. **How to Set Values**:
   - To set these key-value pairs, you use specific **setter methods** provided by Hadoop's **Configuration** class.
   - This is useful when you just need to pass a small piece of information (metadata) to the tasks running in your MapReduce job.
3. **Retrieving Values**:
   - After setting your key-value pairs in the job configuration, the task can later retrieve them using a method like `getConfiguration()`. This allows your tasks to use the information you've provided in the configuration.

Handling Larger Data:

- Typically, you can pass simple **primitive types** (like numbers or strings) as metadata, which is usually enough for small bits of information.
- If you need to pass complex objects (like a custom object or a large dataset), you have two options:
  - You can serialize (convert) the object into a string format yourself (if you have a mechanism for doing that).
  - Or, you can use Hadoop's built-in **Stringifier** class, which handles turning objects into strings (and back again).

⚠ **Don't use this for large amounts of data**. If you try to pass too much data (more than a few kilobytes), it could overload the memory in the MapReduce components.

- This is because the job configuration is read by multiple components (the client, application master, and the task JVM), and every time it's read, all the data is loaded into memory, even if some of it isn't used. This can lead to memory issues.

Technique	What is it?	When to use it?	Main Difference
Job Configuration	A way to pass small data (like settings or key-value pairs) between tasks. It's stored in the job configuration and can be accessed via **getConfiguration()**.	Use it when you need to pass small pieces of metadata or settings to the Mapper or Reducer tasks (e.g., certain values or settings for operations).	Used for passing small data (like numbers or strings) within the job itself, without handling external files.
Distributed Cache	Storing files in distributed memory to be accessible by all nodes in the cluster. Files are loaded from HDFS and distributed to the nodes.	Use it when you need to pass large files or datasets (like big lookup tables) between the Mapper and Reducer tasks.	Used for passing larger files (e.g., customer data or lookup tables) to tasks, handling external files.

Main Difference:

- **Job Configuration** is used for passing small pieces of data (like settings or simple values) as **key-value pairs** within the job configuration.

  For example, when you need to pass a specific value (like a threshold amount or a configuration setting).
- **Distributed Cache** is used for passing entire **files** or **large datasets** to the tasks.

  For example, when you have **large data** that the **Mapper** or **Reducer** needs to access (like a customer database or large lookup tables).

⚠
- **Job Configuration** is for small data like configuration values.
- **Distributed Cache** is for large files or datasets that need to be available across all nodes.

Join in MapReduce

MapReduce can perform joins between large datasets, but writing the code to do joins from scratch is fairly involved. Rather than writing MapReduce programs, you might consider using a higher-level framework such as Pig, Hive, or Spark, in which join operations are a core part of the implementation.

> If the join is performed by the mapper it is called a ***map-side** join*, whereas if it is performed by the reducer it is called a ***reduce-side** join*.

If both datasets are too **large** for either to be copied to each node in the cluster, we can still join them using MapReduce with a map-side or reduce-side join, depending on how the data is structured

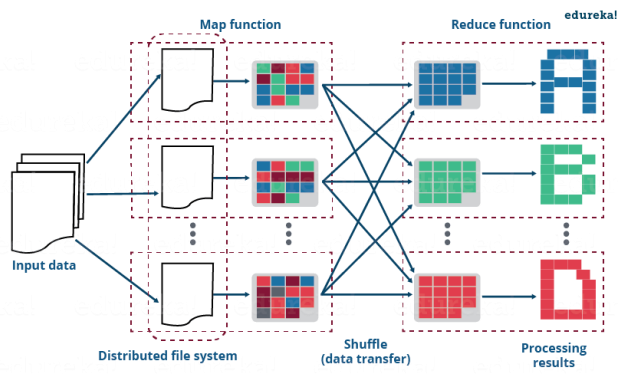
Just like SQL join, we can also perform join operations in MapReduce on different data sets. There are two types of join operations in MapReduce:

1.**Map side join** : the join operation is performed in the map phase itself. Therefore, in the map side join, the mapper performs the join and it is mandatory that the input to each map is partitioned and sorted according to the keys.

2. **Reduce side join** : the join operation is performed in reduce phase. It is easier to implement.

Reduce join

- A reduce-side join is more general than a map-side join, in that the input datasets don't have to be structured in any particular way, but it is **less efficient** because both datasets have to go through the MapReduce shuffle.
- The basic idea is that the mapper tags each record with its source and uses the join key as the map output key, so that the records with the same key are brought together in the reducer.
- We use several ingredients to make this work in practice.



Reduce join works as follows :

1. A reduce-side Mapper reads the input data which are to be combined based on common column or join key.
2. The mapper processes the input and adds a tag to the input to distinguish the input belonging from different sources or data sets or databases.
3. The mapper outputs the intermediate key-value pair where the key is nothing but the join key.
4. After the sorting and shuffling phase, a key and the list of values is generated for the reducer.
5. Now, the reducer joins the values present in the list with the key to give the final aggregated output

in this case, we want to combine customer details (`cust_details`) with transaction details (`transaction_details`) to calculate the total amount spent by each customer. Here's how we can design the job using a **reduce-side join**:

Problem Overview:

- We have two datasets:
  1. **cust_details**: Contains details of customers, including a `customer_id` and `first_name`.
  2. **transaction_details**: Contains transaction records, each with a `customer_id` and a `transaction_amount`.

Our goal is to:

- Retrieve each customer's `first_name`.
- Calculate the `total_amount` spent by the customer (sum of `transaction_amount`).

Step-by-Step Explanation of the Reduce-Side Join:

1. Mapper Phase:

- The mapper will process each record in both datasets (`cust_details` and `transaction_details`).
- For each record, the mapper will emit a key-value pair where the key is the `customer_id`, and the value is either:
  - A tuple that contains the `first_name` from the `cust_details` (with an appropriate tag to distinguish this dataset), or
  - The `transaction_amount` from the `transaction_details` (with a tag for this dataset).

This helps distinguish between data coming from different datasets when they are shuffled and sorted later in the process.

2. Shuffle and Sort:

- The shuffle and sort phase happens automatically in MapReduce. During this phase, all records with the same `customer_id` will be grouped together.
- After the shuffle, we will have a list of values for each `customer_id`. Each list will contain:
  - Some records from `cust_details` (with `first_name`), and
  - Some records from `transaction_details` (with `transaction_amount`).

This step is crucial because it brings together all the relevant data for each `customer_id` before passing it to the reducer.

3. **Reducer Phase**:

- The reducer will receive each `customer_id` along with a list of values (which are either `first_name` from `cust_details` or `transaction_amount` from `transaction_details`).
- The reducer will:
  1. Extract the `first_name` from the list.
  2. Sum all the `transaction_amount` values associated with that `customer_id`.
  3. Output the result as a key-value pair where the key is the `first_name` and the value is the `total_amount` spent by that customer.

Reduce join works

Mapper code :customer details

```
import java.io.IOException;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class mapper1 extends Mapper<Object, Text, Text, Text>{

    @Override
    protected void map(Object key, Text value, Context context) throws IOException, I
      //  super.map(key, value, context); //To change body of generated methods, choo
       String record = value.toString();
        String customer[] = record.split(",");
        context.write(new Text(customer[0]),  new Text("cname "+customer[1]));
    }
}
```

Mapper code : customer transaction

```
package com.mycompany.custreducesidejoin;
import java.io.IOException;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class mapper2 extends Mapper<Object, Text, Text, Text> {

    @Override
    protected void map(Object key, Text value, Context context) throws IOException, I
        String record = value.toString();
        String customer[] = record.split(",");
        context.write(new Text(customer[2]), new Text("camount " + customer[3]));
    }
}
```

- Output from mapper1 as ***Example:*** ・4000001, cname Kristina], [4000002, cname paige].
- Output from mapper2 as ***Example:*** ・4000001, camount 40.33・, ・4000002, camount 198.44・.
- After shuffling and sorting step ・4000001 – **[(cname kristina), (camount 40.33), (camount 47.05),...]**};

Reducer code

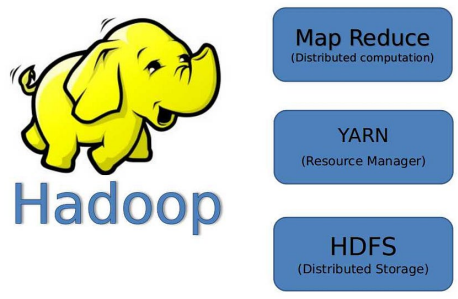
```
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class myreducer extends Reducer<Text, Text, Text, Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IO
        String name = "";
        float amount = 0;
        for (Text t : values) {
            String data[] = t.toString().split(" ");
            if (data[0].equals("cname")) {
               name = data[1];
            } else if (data[0].equals("camount")) {
                amount += Double.parseDouble(data[1]); }
        }
        context.write(new Text(name), new Text(String.valueOf(amount)));
    }
}
```

Driver code

```
public class myrjdriver {
    public static void main(String[] args) throws IOException, InterruptedException,
        Configuration c = new Configuration();
        Job j = Job.getInstance(c, "reduce join job");
       c.set("fs.defaultFS", "hdfs://localhost:8020");
        j.setJarByClass(myrjdriver.class);
        j.setReducerClass(myreducer.class);
        j.setOutputKeyClass(Text.class);
        j.setOutputValueClass(Text.class);
        MultipleInputs.addInputPath(j, new Path("hdfs://localhost:8020/user/cloudera/
        MultipleInputs.addInputPath(j, new Path("hdfs://localhost:8020/user/cloudera/
        FileSystem.get(c).delete(new Path("hdfs://localhost:8020/user/cloudera/crj"),
        FileOutputFormat.setOutputPath(j, new Path("hdfs://localhost:8020/user/cloude
        System.exit(j.waitForCompletion(true) ? 0 : 1);
    }
}
```

Lecture 6 - HDFS and YARN

- There are **three** core components in Hadoop :
  - **HDFS**
  - **YARN**
  - **MapReduce**



HDFS

is a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

Very large files:

HDFS is meant for storing extremely large files, ranging from hundreds of megabytes to petabytes. It's optimized for handling vast amounts of data, often in the range of gigabytes to terabytes.

Streaming data access :

The system is built to handle a ***"write once, read many times"*** approach. Once data is written, it's typically read many times for different analyses or processes.

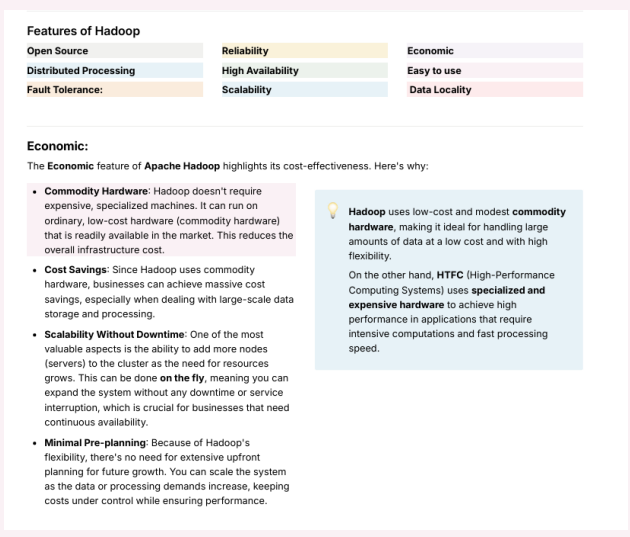
HDFS prioritizes efficiently reading large datasets, rather than quickly accessing individual records.

Commodity hardware

HDFS runs on clusters of inexpensive, commonly available hardware. This makes it cost-effective and scalable. The system is built to tolerate hardware failures without disrupting operations, ensuring high availability even when individual nodes fail.

commodity hardware (commonly available hardware that can be obtained from multiple vendors)

Remember



HDFS is not suitable for :

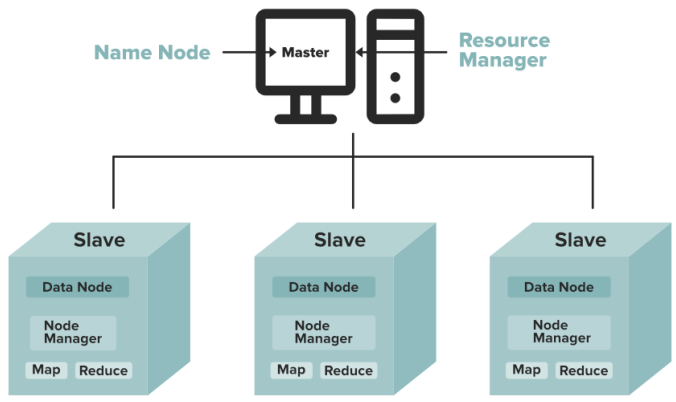
1.Low-latency data access Applications HDFS is designed for high throughput (large-scale data access), but it doesn't provide **low-latency access**. Applications that need very fast data retrieval, in the range of milliseconds, will struggle with HDFS.

For example, **HBase** is better suited for applications that require quick, real-time data access.

2.Lots of small files Because The **NameNode** (which manages the metadata of files) stores this metadata in memory, and the more files you have, the more memory is required. Since each file or block uses about 150 bytes of memory, storing a huge number of small files (like billions) exceeds the memory limits of current hardware. Therefore, HDFS works best with **large files** rather than numerous small ones.

Hadoop cluster contains

1. NameNodes
2. DataNodes.



- The **NameNode** is essential for the operation of HDFS and must always be running.
- A Hadoop cluster contain at least **two NameNodes** (Active and Standby) for redundancy.
- **HDFS** is designed for **sequential data writing** and does not support random writes or updates to files once they're stored.

The Active NameNode is responsible for all client operations in the cluster, while the Standby NameNode acts as a backup.

Data Storage and Replication in HDFS:

Data is Split into Blocks

- Data in HDFS is divided into **blocks** for easier storage and processing. The block size is configurable, typically set to **64MB**, **128MB**, or **512MB**. Larger block sizes are often used to store and process large files efficiently.

Replication of Blocks

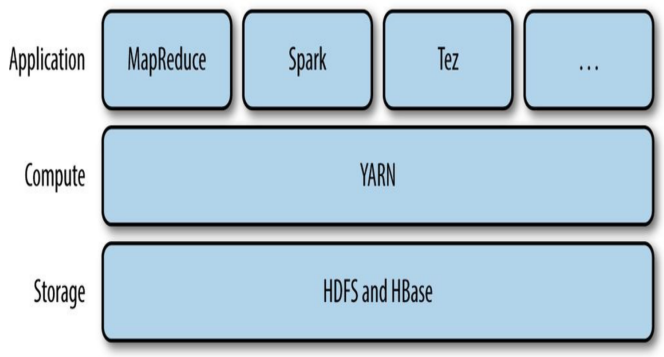
- Each block of data is **replicated** across multiple **DataNodes** in the cluster for redundancy and fault tolerance. The default replication factor is **three**, meaning each block is stored on **three different nodes**. This ensures that if one node fails, the data is still available from the other replicas.

Distributed Across DataNodes and Racks

- The blocks are **distributed across different nodes** in the cluster. To improve data availability and fault tolerance, it's recommended that **two replicas** be placed on nodes within the **same rack**, and the **third replica** be placed on a node in a **different rack**. This setup ensures that even if an entire rack fails, the data can still be retrieved from the replica stored in another rack.

YARN

- **Apache YARN** (Yet Another Resource Negotiator) is Hadoop's cluster resource management system.
- YARN was introduced in **Hadoop 2** to improve the MapReduce implementation, but it is general enough to support other distributed computing paradigms as well..
- YARN provides APIs for requesting and working with cluster resources, but these APIs are not typically used directly by user code. Instead, users write to higher-level APIs provided by distributed computing frameworks, which themselves are built on YARN and hide the resource management details from the user.

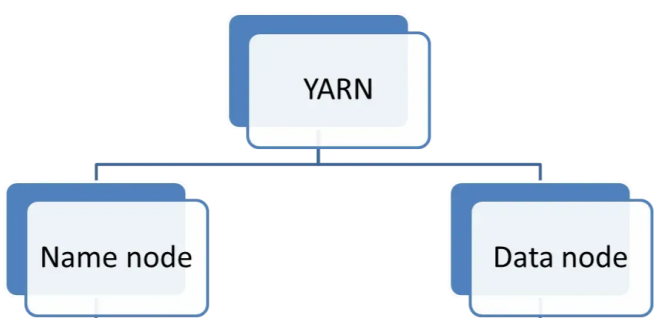


With **YARN**, multiple processing engines like **MapReduce**, **Spark**, and **Flink** can now run **on the same data in HDFS** simultaneously.
This provides
increased efficiency, **better performance**, and **flexible resource allocation** across various engines.

- Cluster Architecture
  - works alongside **HDFS** (Hadoop Distributed File System) and **HBase** (a distributed NoSQL database) in the **cluster storage layer** to provide a complete infrastructure for data processing and storage.
  - The **compute layer** (YARN) interacts with these storage layers to manage jobs and their execution across the cluster.

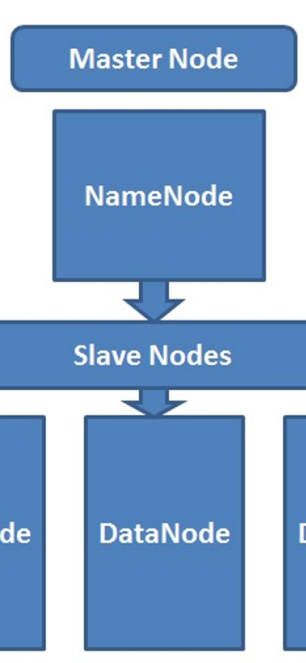
The **NameNode** manages information about where data is located,

while the **DataNode** stores the
actual data.



important

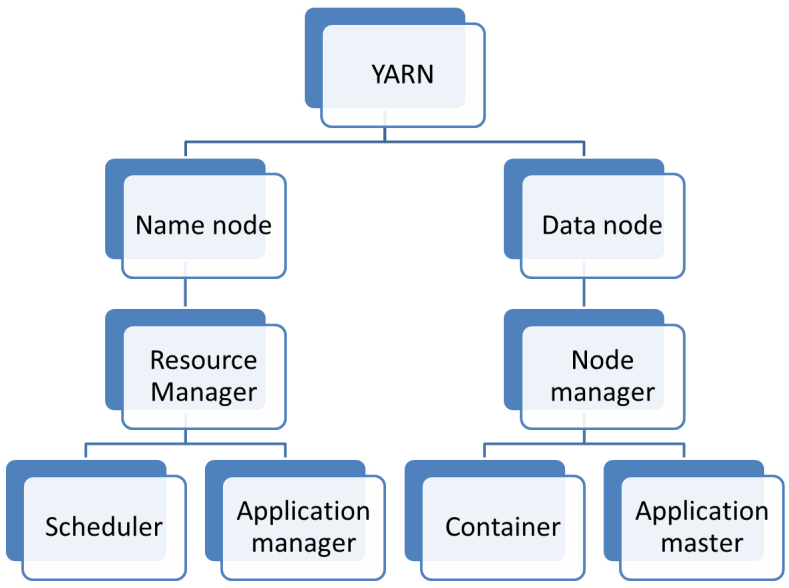
In **Hadoop MapReduce**, **Map** and **Reduce** tasks are **executed on the DataNode**, (where the data is stored), not on the **NameNode**.



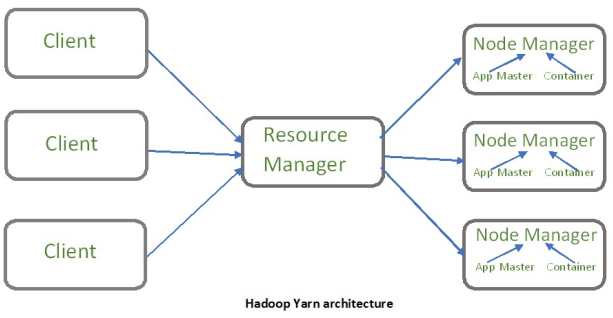
YARN architecture

Resource manager (long running daemon):

It is the master daemon of YARN and is responsible for resource assignment and management among all the applications. Whenever it receives a processing request, it forwards it to the corresponding node manager and allocates resources for the completion of the request accordingly.



- The **Resource Manager** (RM) is a **single instance** per Hadoop cluster, and it is responsible for managing the allocation and use of resources across the entire cluster. This central role ensures that resources are efficiently distributed and utilized among the various applications running on the cluster.



Hadoop Yarn architecture

The **Resource Manager** is the **master daemon** in YARN, responsible for managing and allocating cluster resources to various applications.

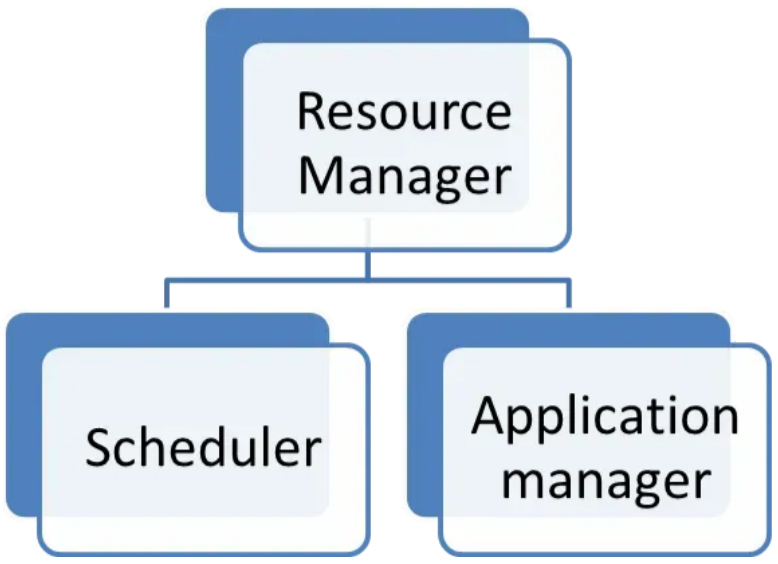
receives resource requests from applications and forwards them to the appropriate **Node Manager** for execution.

plays a critical role in ensuring efficient resource utilization across the cluster.

run on master node

- **Cluster resource tracking:** It maintains a global view of the cluster and tracks the available resources on each node.
- **Cluster health monitoring:** It monitors the health of nodes in the cluster and manages the failover of resources in case of node failures.
- **Cluster resource allocation:** It receives resource requests from application masters and allocates the necessary resources to run the application.

- It has **two** major components:
  1. **Application manager**
  2. **Scheduler**



Resource manager – Application manager :

The **Application Manager** is a critical component of YARN that interacts with the **Resource Manager** to manage the lifecycle of applications submitted to the cluster.

- It is responsible for accepting the application and negotiating the first container (application master) from the resource manager. It also restarts the **Application Master** container if a task fail.

Tasks:

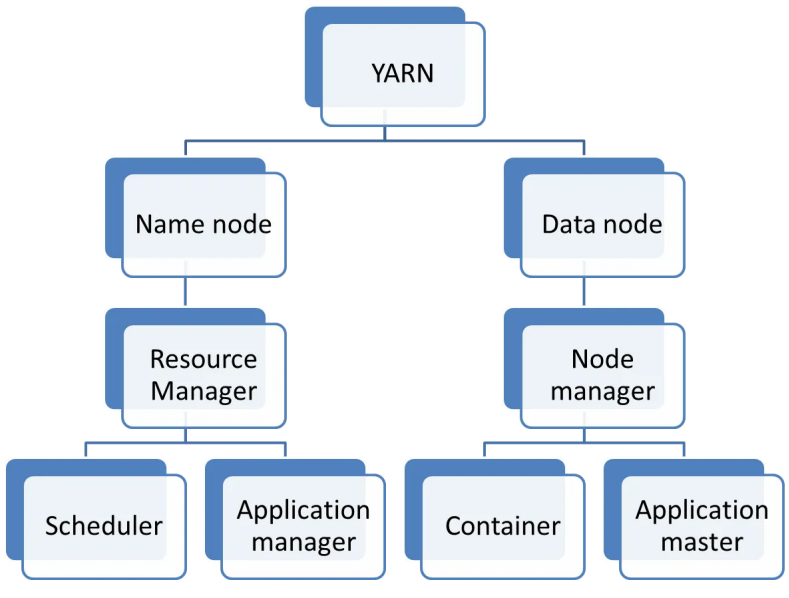
- **Application Manager** is responsible for accepting, validating, and managing the lifecycle of applications.
- It works by **validating resource requests**, ensuring unique application IDs, and forwarding applications to the **Scheduler**.
- It **handles application failures**, logs execution summaries, and keeps a **cache** of completed applications for user access.

Application manager tasks :

1. Application manager validates the application specification
2. It rejects any application that requests unsatisfiable resources for its Application Master (i.e., no node in the cluster has enough resources to run the Application Master itself).
3. It ensures that no other application was already submitted with the same application ID.
4. Forward the application to the scheduler
5. It places an Application Summary in the daemon’s log file after the application finishes.
6. the Applications Manager keeps a cache of completed applications long after applications finish to support users requests for application data

Node manager

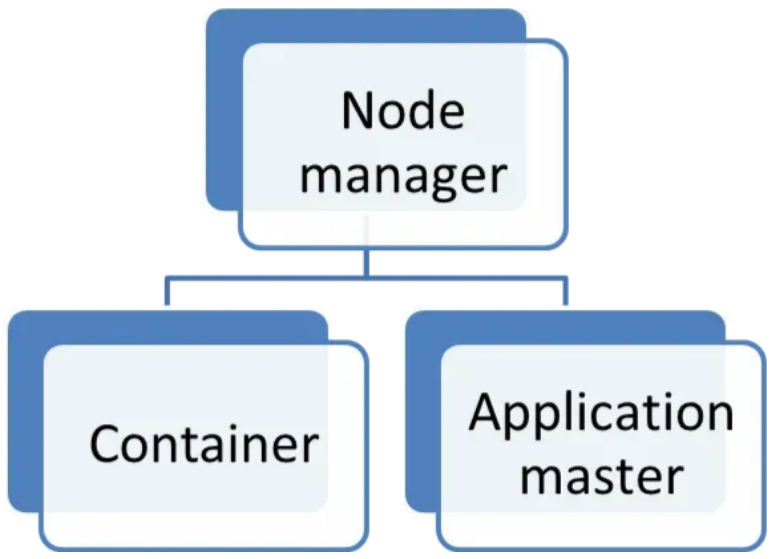
The **Node Manager** (NM) is a crucial component in **YARN** that manages the resources and execution of tasks on each node in the cluster. It communicates with the **Resource Manager** and ensures that applications are running as expected on the node.



Node manager responsibilities :

- **Reporting node health**: Each Node Manager announces itself to the Resource Manager and periodically sends a heartbeat to provide node status and information, including memory and virtual cores. In case of failure, the Node Manager reports any issues to the Resource Manager, diverting resource allocations to healthy nodes.
- **Launching Containers**: Node Managers take instructions from the Resource Manager, launch containers on their nodes, and set up the container environment with the specified resource constraints.
- **Container Management**: Node Managers manage container life cycle, dependencies, leases, resource usage, and log management
- **Log Handler**: keeping the containers' logs, and uploading them onto a file-system.

- **Node manager** has **two** components
  1. **Containers**
  2. **Application master**



Node manager - **container** :

- **A container** is a collection of physical resources such as RAM, CPU cores, and disks on a single node.
- There can be multiple containers on a single node.
- Every node in the system is considered to be composed of multiple containers of minimum size of memory (e.g., 512 MB or 1 GB) and CPU.
- A container executes an application-specific process with a constrained set of resources(memory, CPU, and so on).
- Containers are assigned to execute tasks from applications, such as **MapReduce jobs or Spark tasks**
- RM allocating containers based on the application requirements.
- The application run in one container or more

Node manager - **Application master**:

- The Application Masters (AM) manage the execution of tasks within these containers, monitor their progress, and handle any task failures or reassignments.
- Each application starts out as an Application Master, which is itself a container (often referred to as container 0).
- Once started, the Application Master must negotiate with the Resource Manager for more containers using a protocol based on Remote Procedure Calls (**RPC**) over the network .
- Container requests **(and releases)** can take place in a dynamic fashion at run time. For instance, a MapReduce job may request a certain amount of mapper containers; as they finish their tasks, it may release them and request more reducer containers to be started.
- The Application Master is the process that coordinates an application's execution in the cluster (It runs in container 0).
- Each application has its own unique Application Master (one per application), which is tasked with negotiating resources (containers) from the Resource Manager and working with the Node Manager(s) to execute and monitor the tasks.
- It will periodically send heartbeats to the Resource Manager to affirm its health and to update the record of its resource demands.

---

1. ResourceManager Receives the Job:

- When a job is submitted to **YARN**, the **ResourceManager** (RM) **receives the job**. This job typically consists of a **JAR file** (such as a MapReduce application or a Java program) and the **dataset location** (where the data is stored in HDFS).
- The **ResourceManager** is responsible for managing the overall **cluster resources**, but it does **not execute the job itself**. Instead, it allocates resources across the cluster for different applications.

2. Role of the ApplicationManager:

- Once the **ResourceManager** receives the job, it hands the job off to the **ApplicationManager**. The **ApplicationManager** is a component of the **ResourceManager**.
- The **ApplicationManager** is responsible for **launching a new application**, i.e., the **Application Master** for that job.
- The **ApplicationManager** validates that the job can run in the cluster and then launches the **Application Master** (AM), which will manage the job's execution across the cluster.

3. Role of the Application Master:

- The **Application Master** is responsible for **managing the execution** of the job (application) on the cluster.
- **Application Master** locates the appropriate **NodeManagers** that have enough resources (such as memory and CPU) to run the tasks for the job.
- It communicates with the **ResourceManager** to request resources and assigns **containers** to execute the job's tasks on the available **NodeManagers**.

4. Containers on DataNodes:

- **Containers** are environments where the **Map** and **Reduce** tasks are executed.
- These **containers** are **allocated by NodeManagers** on **DataNodes** in the cluster.
- **NodeManagers** are responsible for **managing containers** and **executing tasks** inside them. This means that tasks are actually executed on the **DataNodes**.
- When the **Application Master** requests resources for tasks, it typically does so on the **DataNodes** where the data resides, ensuring efficient **data locality**.

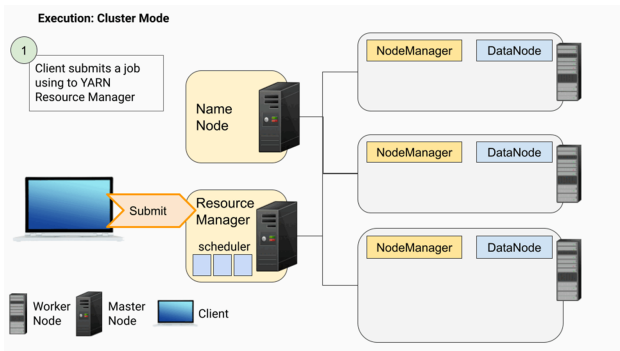
5. Role of the NameNode:

- The **NameNode** in **HDFS** is not involved in resource allocation or task execution. Its role is solely to manage **metadata** for the HDFS system.
- The **NameNode** keeps track of where the data blocks are stored across the cluster (which DataNodes contain which blocks of data).
- While the **Application Master** and **NodeManager** deal with job execution, the **NameNode** provides information on the **location of the data** in HDFS so that the **Application Master** can access it during processing.

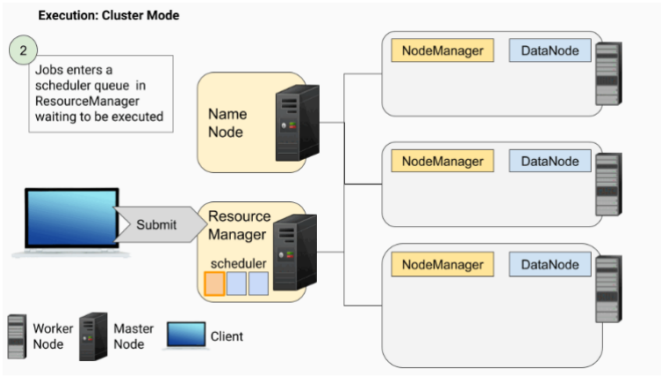
- **In Hadoop YARN**, a **container** is created **per request**, meaning that a container is created for each task that needs to be executed.
- **The NodeManager** is responsible for the **actual creation** of containers in **Hadoop YARN**, based on the **request** coming from the **ResourceManager**.
- The **NodeManager** exists on **every DataNode** in **Hadoop YARN**.
- The **ResourceManager** is present in **every Cluster** in **Hadoop YARN**.
- The **ApplicationMaster** in **Hadoop YARN** needs to communicate with the **ResourceManager** to get information about **where the data is stored**

Running application in YARN

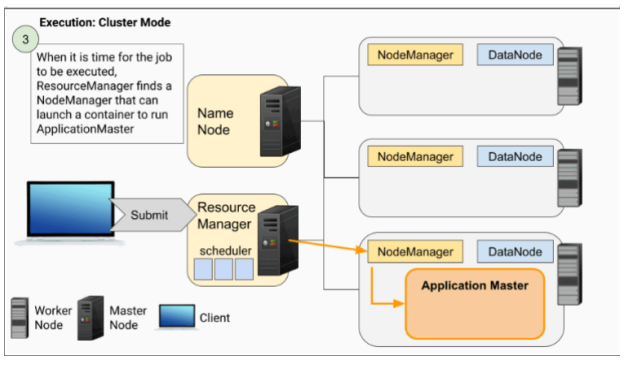
- **Step 1:** A client submits a job using “spark-submit” to the YARN Resource Manager.



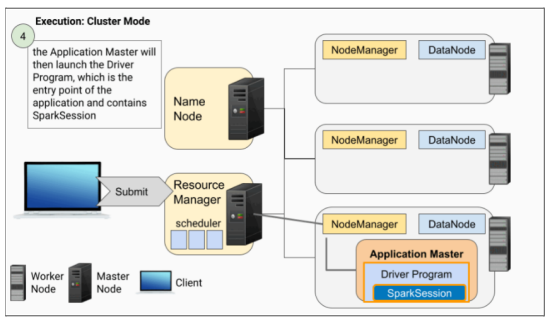
- **Step 2:** The job enters a scheduler queue in the Resource Manager, waiting to be executed.



- **Step 3:** When it is time for the job to be executed, the Resource Manager finds a Node Manager capable of launching a container to run the Application Master.

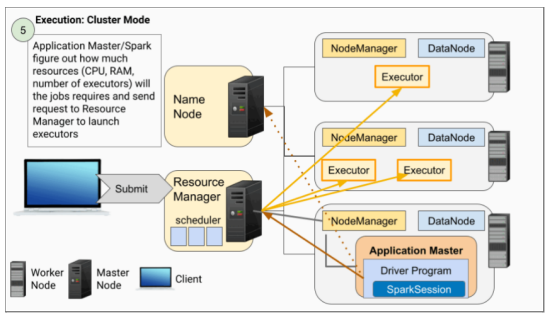


- **Step 4:** The Application Master launches the Driver Program (the entry point of the program that creates the SparkSession/SparkContext).

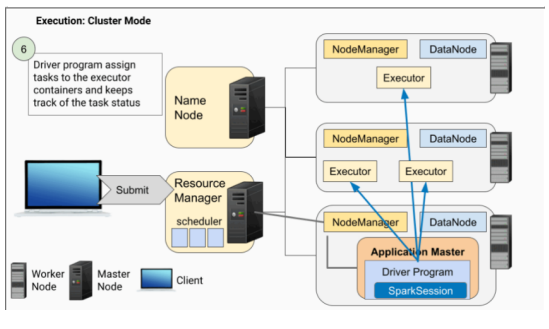


- **Step 5:** The Application Master/Spark calculates the required resources (CPU, RAM, number of executors) for the job and sends a request to the Resource Manager to launch the executors.

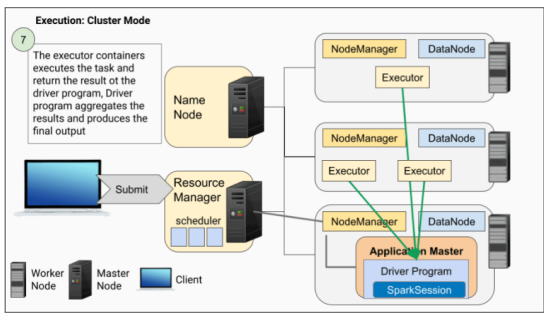
  The Application Master communicates with the NameNode to determine the file (block) locations within the cluster using the HDFS protocol.

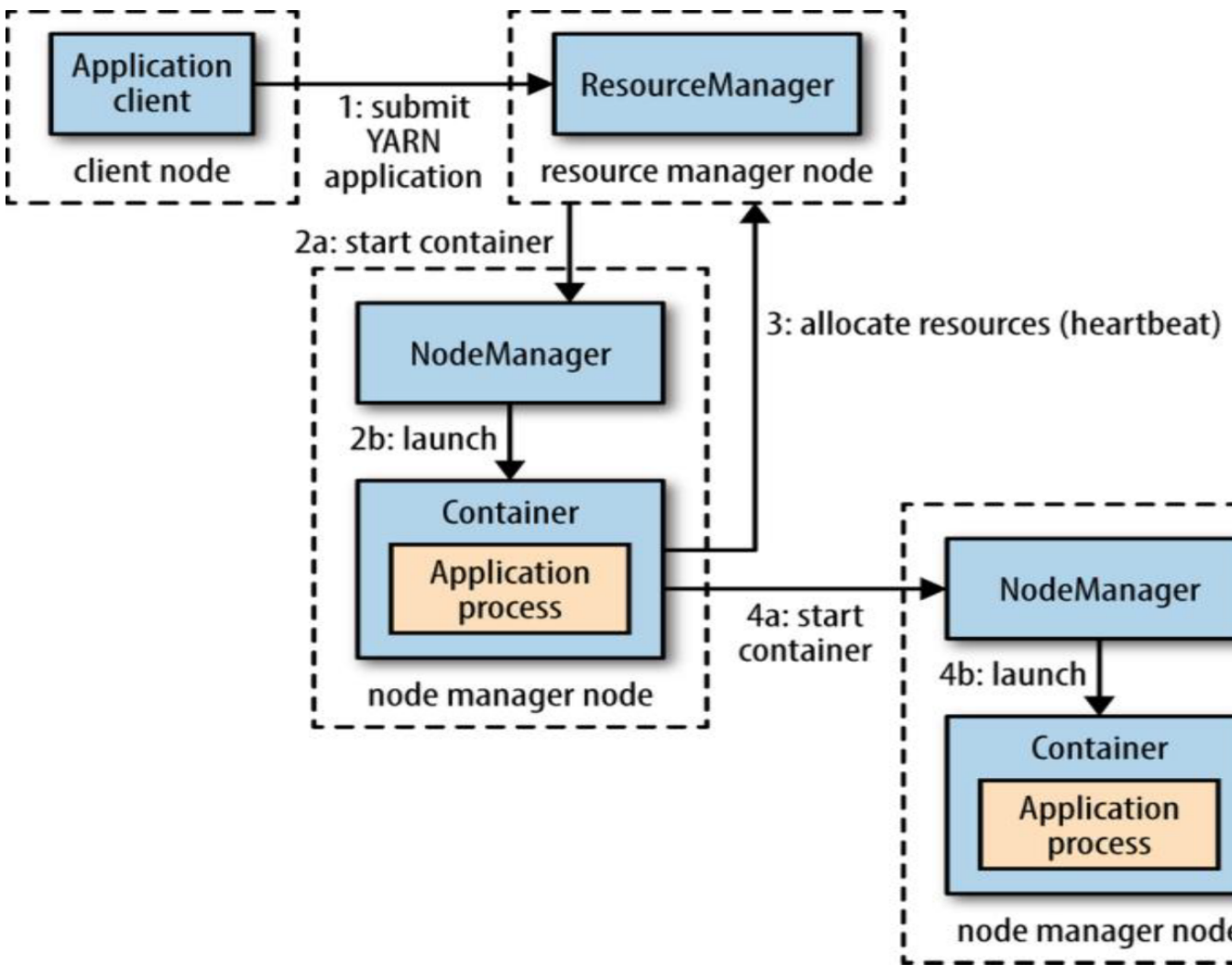


- **Step 6:** The Driver Program assigns tasks to the executor containers and keeps track of the task status.



- **Step 7:** The executor containers execute the tasks and return the results to the Driver Program. The Driver Program aggregates the results and produces the final output.).





YARN benefits

- **Centralized Resource Management**: YARN provides a centralized resource management system that can dynamically allocate and manage resources for different frameworks/applications running on the Hadoop cluster.
- **Flexibility**: The scheduling and resource management capabilities are separated from the data processing component. This allows for running various types of data processing applications on a single cluster.
- **Better Cluster Utilization**: YARN's dynamic resource allocation ensures each application gets the resources it needs to run successfully without affecting other applications. Additionally, resources unutilized by one framework/application can be consumed by another.

- **Better Cluster Utilization**: The Node Manager is a more generic and efficient version of the Tasktracker Instead of having a fixed number of map and reduce slots, the Node Manager has a number of dynamically created resource containers The size of a container depends upon the amount of resources assigned to it, such as memory, CPU, disk, and network IO.
- **Cost-Effective:** With YARN, one "do-it-all" Hadoop cluster can run a diverse set of workloads and support a variety of applications, making it a more efficient and cost-effective platform for big data processing.
- **Reduced Data Movement:** As there is no need to move data between Hadoop, YARN, and systems running on different clusters of computers, data motion is reduced.

YARN vs MapReduce 1

Aspect	MapReduce 1	YARN
Main Daemons	**JobTracker** and **TaskTrackers**	**ResourceManager** and **NodeManagers**
Job Coordination	JobTracker coordinates and schedules jobs	ResourceManager coordinates resources, and each job has its own **ApplicationMaster**
Task Execution	TaskTrackers run tasks and report progress to JobTracker	NodeManagers run tasks (as containers) and report status to ResourceManager
Task Scheduling	JobTracker handles both job scheduling and task progress	ResourceManager schedules resources for tasks, **ApplicationMaster** schedules the job
Task Progress Monitoring	JobTracker monitors task progress, retries failed tasks	**ApplicationMaster** monitors task progress, NodeManager handles task execution
Task Recovery	JobTracker reschedules failed tasks on different TaskTrackers	ApplicationMaster can request new containers from ResourceManager if tasks fail
Job History	JobTracker stores job history, with optional separate JobHistoryServer	Timeline Server stores application history
Task Execution Units	TaskTrackers handle execution of tasks	NodeManagers handle containers which execute tasks (Map/Reduce)
Failure Handling	JobTracker reschedules tasks on other TaskTrackers	ApplicationMaster requests new containers from ResourceManager on failure
Resource Management	Handled by JobTracker (combines task scheduling and resource management)	Handled by ResourceManager (manages cluster-wide resources)

MapReduce 1	YARN
jobtracker	Resource manager, application master, timeline server
Tasktracker	Node manager
slot	container

Lecture 7-Apache Hive

Apache Spark and Apache Flink are not directly tied to Hadoop, which means they do not necessarily have to work on data stored in HDFS.

MapReduce is part of the Hadoop **Ecosystem**, which refers to the set of systems that operate under the Apache Hadoop project.

Tools like Hive, Pig, and Sqoop are essentially interfaces that simplify working with data stored in HDFS. These tools convert the instructions you write (such as SQL in the case of Hive) into MapReduce Jobs or other processing engines (like Spark). The goal is to simplify interacting with the data without the need to write complex code using MapReduce directly.

Among the other **tools** that Hadoop works with:

- **Hive**
- **Sqoop**
  - Used to **import** data from traditional databases such as MySQL and PostgreSQL into Hadoop or **export** data from Hadoop to databases.
- **Flume**
  - A tool for collecting and transferring **real-time** data to HDFS.
  - It is commonly used to capture **logs** or event data.
- **Mahout**
  - A **machine learning** library built on top of Hadoop.
  - It is used for building recommendation systems, classification, and clustering models."

Apache Hive

- **Facebook** built **Apache Hive** on top of the **Hadoop** framework for **data warehousing**.

  Hive was developed to manage and analyze the massive volumes of data Facebook was generating daily from its social network.

  After trying several systems, the team chose **Hadoop** for storage and processing due to its cost-effectiveness and scalability.

The main purpose of **Hive** is to allow you to write **SQL queries** on data stored in **HDFS** in a simplified way. It is used with a **data warehouse** because the data stored in **HDFS** is usually large and complex

- Hive was created to allow analysts with strong **SQL** skills (but limited **Java** experience) to run queries on the large datasets stored in **HDFS**.

  Today, Hive is a successful Apache project used by many organizations as a scalable, general-purpose data processing platform.

  The current version of Hive is **4.0.0-beta-1**, released on **14 August 2023**.

Apache Hive vs Traditional database

Feature	Apache Hive	Traditional DB
Purpose	Big data analytics	Transaction systems and Data warehouse
Data Size	Designed to handle petabytes of data.	Gigabytes to terabytes; some systems handle petabytes at higher cost.
ACID	Limited support	Full ACID support
Speed	Slower, optimized for batch processing.	Faster, optimized for real-time transactional processing.
Storage	Built on top of HDFS	Uses internal storage mechanisms.
Compute Engine	MapReduce, Spark, and TEZ	Built-in engine
Metadata Storage	Stored in Hive Metastore.	Stored in system catalogs within the database
Query Language	HiveQL	SQL

ACID: Atomicity, Consistency, Isolation, Durability .

Schema on Write vs. Schema on Read: Trade-Offs

Schema on Read (Hive):

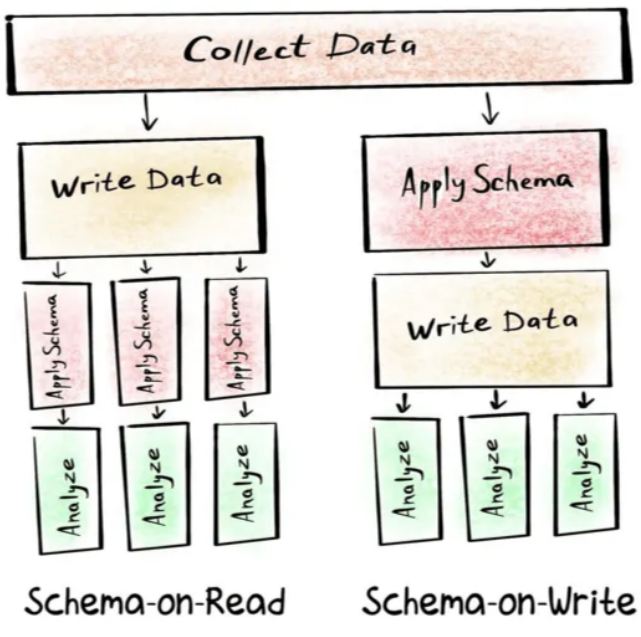
The schema is applied only when a query is executed, not during data loading.

- **Advantages**
  - **Fast Initial Load:** Enables quick data loading as it skips validation or restructuring during this phase.
  - **Flexibility:** Supports multiple schemas for the same data, facilitating diverse analyses.
  - **Adaptability:** Handles scenarios where the schema is unknown at load time or the data is heterogeneous.
- **Disadvantages**
  - **Query Performance:** Slower query execution due to schema enforcement at read time, adding extra processing overhead.
  - **Less Structured Data:** May result in inconsistencies if the raw data is unclean or poorly organized.

Schema on Write (Traditional Databases):

The schema is enforced when data is loaded. Data that doesn't conform to the schema is rejected.

- **Advantages:**
  - **Query Time Performance:** Faster query execution since the database can index columns and compress data during the load process.
  - **Data Integrity:** Ensures consistent and valid data within the database.
- **Disadvantages:**
  - **Load Time Overhead:** Takes longer to load data, as the database must read, parse, and serialize it into its internal format.
  - **Inflexibility:** The schema must be predefined, which can be challenging when the data structure is not yet known or may change.



Updates, Transactions, and Indexes in Hive

- **Updates and Transactions**:
  - HDFS does not support in-place updates, so changes (inserts, updates, and deletes) are stored in small delta files.
  - These delta files are merged into the base table files using background MapReduce jobs handled by the Metastore.
  - Transactions, introduced in Hive 0.13.0, are required to enable these features.
  - Tables must have transactions enabled to support updates.
- **Table and Partition Locking**:
  - Hive supports locking mechanisms at both the table and partition levels for concurrency management.
- **Indexes**:
  - Hive supports indexes to improve query performance in specific cases.
  - Example: Queries like `SELECT * FROM t WHERE x = a` can leverage an index on column `x` to reduce the number of files scanned.

Fine grained vs coarse grained

- **Fine-Grained Update:** Refers to updating a single record or element in a database. This approach is flexible but becomes very costly when dealing with billions of records, as you need to store the details of each individual update.
- **Coarse-Grained Update:** Involves applying functional operations like **map**, **reduce**, **flatMap**, and **join** (commonly used in Spark). Instead of updating each record individually, a single function is applied to a large number of records at once. This reduces storage costs since you only save the function itself rather than the details of each update.

Spark's Advantage: Spark uses a **DAG (Directed Acyclic Graph)** model, which stores a small sequence of operations that can be recomputed as long as the original data is available. Fine-grained updates make recomputation difficult and expensive, while coarse-grained updates make this process more efficient.

Coarse-grained

Updating large chunks of data at once, such as a full row or multiple records using a broader operation.

In **Hive**, updates are generally applied at the row-level rather than the individual cell or column level, making updates more **coarse-grained.**

Coarse-Grained Updates in Hive:

1. **Full Row Rewrites:**
   - When you update a row in Hive (e.g., updating the salary of an employee), it doesn't modify just the specific cell or column.
   - Instead, Hive deletes the old row and writes a new version of the entire row with the updated value.

Hive doesn't natively handle fine-grained updates where individual cells can be updated independently without affecting the entire

- Example: Running `UPDATE employee_table SET salary = 80000 WHERE employee_id = 12345` will result in rewriting the entire row, not just the salary column.

2. **Partitioning and File-Based Storage:**
   - Hive stores data in files on Hadoop Distributed File System (HDFS).
   - When data is updated, Hive typically creates new versions of these files (or partitions) instead of modifying individual cells directly.
   - If the table is partitioned, the update may require rewriting the entire partition, even if only one row is changed.

Key Limitations of Coarse-Grained Updates:

1. **Performance Overhead:**
   - Rewriting entire rows or partitions can be computationally expensive.
   - For large datasets, updates become inefficient, especially when only a small subset of the data needs modification.
2. **Lack of Transactional Support:**
   - Hive's traditional update mechanism is not transactional by default.
   - It doesn't natively support ACID properties like handling concurrent updates, rollbacks, or fine-grained version control.

Fine-grained

In Hive, fine-grained update would involve the ability to update individual cells (specific columns within a row) or even small portions of data without rewriting entire rows or partitions.

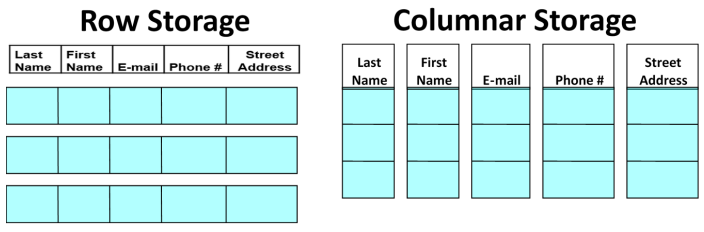
> In Hive's traditional implementation, this is not natively supported, but there have been advancements and newer features to handle more granular updates.

ACID Transactions:

- Starting from Hive 0.14, support for ACID (Atomicity, Consistency, Isolation, Durability) transactions allows more structured updates like `INSERT`, `UPDATE`, and `DELETE`.
- Fine-grained updates are still somewhat limited compared to what you might expect from an RDBMS like MySQL or PostgreSQL.

Key Components of Fine-Grained Updates in Modern Hive:

1. **ACID Tables:**
   - Hive can be configured to use **ACID-compliant tables**, enabling support for `INSERT`, `UPDATE`, and `DELETE` operations with transactional guarantees.
   - Despite ACID support, updates in Hive are **row-level**, meaning the entire row is rewritten when an update is performed, not just the specific column or cell.
2. **Transactional Storage Format:**
   - Hive utilizes the **ORC (Optimized Row Columnar)** format, which is designed for columnar data storage.
   - ORC enhances performance for queries and updates by focusing on columns, but even with this format, updates are handled at the **row** or **partition** level rather than being truly fine-grained.



Example:

```
UPDATE employee_table
SET salary = 85000
WHERE employee_id = 12345;
```

- **What Happens:**
  - Hive rewrites the entire row with `employee_id = 12345`, not just the `salary` column.
  - The system ensures **atomicity**: either the row is fully updated, or no changes occur, even if a failure happens during the operation.

Feature/Aspect	Coarse-Grained Updates	Fine-Grained Updates
Level of Update	Affects entire rows or partitions	Ideally affects specific cells or columns
Performance	Slower for large datasets (due to full row rewrites)	Potentially faster for small changes, but not natively supported in Hive
ACID Support	Limited (pre-Hive 0.14)	Supported with Hive ACID (post-Hive 0.14)
Row-Level Updates	Yes, but with row rewriting	Yes, but limited by transactional model
Column-Level Updates	Not directly supported (entire rows are rewritten)	Possible through columnar formats (like ORC)
Complexity of Setup	Simple to configure	Requires ACID configuration and transactional storage
Use Cases	Large batch processing, non-transactional data systems	Transactional workloads with specific, fine-grained data changes

Apache Hive Alternatives:

Cloudera Impala:

An open-source, interactive SQL engine designed for low-latency queries on large-scale datasets.

- Offers significantly faster performance compared to Hive on MapReduce.
- Uses a dedicated daemon on each **data node** for query execution.
- Compatible with Hive metastore, Hive formats, and most HiveQL constructs (also supports SQL-92).

How It Works:

- A client query contacts an arbitrary node running Impala's daemon, which acts as the coordinator.
- The coordinator distributes tasks to other daemons and combines the results into a final output.

Use Case: Migrating from Hive to Impala or using both systems on the same cluster is straightforward due to shared compatibility.

Presto:

Open-source, distributed SQL query engine originally have similar architectures to Impala

---

Apache Drill:

Open-source, distributed SQL query engine originally have similar architectures to Impala.

- **Key Features:**
  - Targets SQL:2011 standard rather than HiveQL , supporting advanced SQL functionalities.

Spark SQL:

A SQL module built on Apache Spark, enabling SQL queries on top of Spark's distributed computing framework.

⚠ **Important Note:** Spark SQL is different to using the Spark execution engine from within Hive.

Hive, on Spark provides all the features of Hive since it is a part of the Hive project.

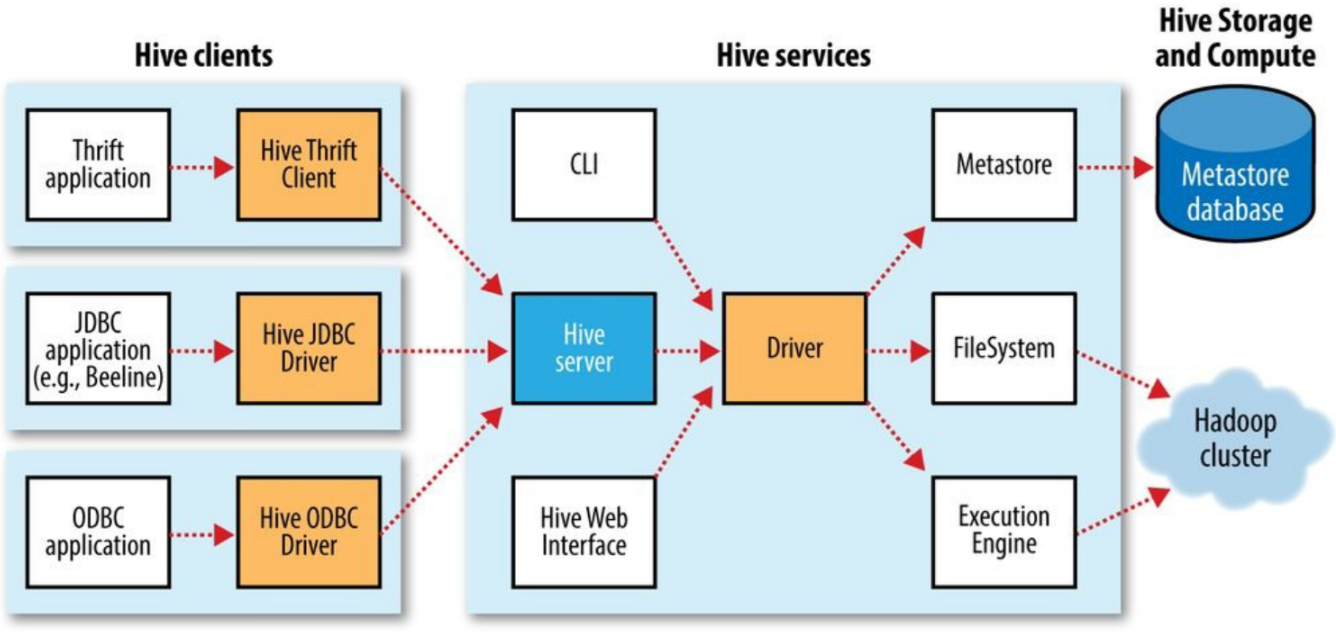
Spark SQL, on the other hand, is a new SQL engine that offers some level of Hive compatibility.

Apache Phoenix:

A SQL layer built on HBase, designed for operational analytics.

- **Key Features:**
  - Provides SQL access via a JDBC driver.
  - Converts SQL queries into HBase scans.
  - Leverages HBase coprocessors for server-side aggregation.

Apache Hive Architecture



Hive Clients

Hive clients connect to the Hive Server and allow various applications or users to interact with Hive. Clients and drivers serve as intermediaries, connecting your applications with Hive Server in the Hive services

Types of Hive Clients:

1. Thrift Client

- **Thrift** is a communication protocol that allows applications written in any language supporting Thrift (e.g., Python, Ruby) to interact with Hive.

  Hive Server is exposed as a **Thrift Service**, enabling cross-language compatibility.

  - When you have applications written in non-Java languages like Python or Ruby, you can use Thrift to connect them to Hive.

  Example:

  - Running a HiveQL query from a Python script using a library that supports Thrift.

---

2. JDBC Driver

- JDBC (**Java** Database Connectivity) is a standard interface in Java for connecting to databases.

  Hive provides a **JDBC Driver**, allowing Java-based applications to connect to Hive Server at a specified host and port.

---

3. ODBC Driver

- ODBC (Open Database Connectivity) is a protocol that allows tools supporting ODBC (e.g., BI tools like Tableau or Power BI) to connect to Hive.

  The ODBC Driver for Hive enables such tools to interact with Hive using Thrift as the underlying communication protocol.

- **Use Case:**
  - Connecting BI tools to Hive for data visualization and analysis.
  - Extracting and analyzing data stored in Hive using tools like Tableau or Power BI.

  Notes:

  - Apache Hive does not ship with an ODBC Driver, but third-party drivers are available.
  - ODBC Driver relies on **Thrift** for communication, similar to the Thrift Client.

Key Differences Between Thrift, JDBC, and ODBC

Feature	Thrift Client	JDBC Driver	ODBC Driver
Target Users	Various programming languages	Java-based applications	BI tools supporting ODBC
Protocol	Thrift	JDBC	ODBC (uses Thrift)
Applications	Python, Ruby	Java applications	Tableau, Power BI

How These Clients Work

1. **Connection:**
   - Each client (Thrift, JDBC, or ODBC) establishes a connection with **Hive Server**.
2. **Query Execution:**
   - Queries are sent via the client to Hive Server, which processes them.
3. **Result Retrieval:**
   - Hive Server executes the query and sends the results back to the client.

Hive Services

Hive services provide interaction points for clients to access Hive functionality. Act as the intermediaries for client interactions with Hive.

Types of Hive Services

1. CLI (Command-Line Interface)

- The default interface for interacting with Hive.

  Requires a local installation of Hive components and configurations.

  Runs in **embedded mode**, meaning it directly accesses Hive's core components.

---

2. Beeline

- A lightweight command-line interface for Hive.
- Can operate in:
  1. **Embedded mode:** Directly interacts with Hive components.
  2. **Remote mode:** Connects to **HiveServer 2** via JDBC for executing queries.

---

3. HWI (Hive Web Interface)

- A web-based interface for Hive, providing an alternative to CLI or Beeline.

  Allows users to run queries and browse the **Hive Metastore** without installing additional client software.

---

4. HiveServer 2

- An advanced version of the original Hive Server.

  Exposes a **Thrift service** for connecting to Hive from various clients (Thrift, JDBC, ODBC).

  Provides a **remote interface** for running queries and managing user sessions.
- **Key Features:**
  - Supports **authentication** and **multiuser concurrency**.
  - Allows programmatic submission of HiveQL queries.

Hive Services

Service	Description
CLI	Default interface; works locally.
Beeline	Command-line tool; supports remote connections.
HWI	Web-based interface for Hive.
HiveServer 2	Advanced server for Thrift, JDBC, and ODBC.

Hive Execution Engine

The **execution engine** is a critical component in Hive responsible for running the query plans generated by the query compiler.

Options:

1. **MapReduce:** Default execution engine, suitable for large batch jobs but slower.
2. **Tez:**
   - A DAG-based execution engine offering better performance than MapReduce.
   - Suitable for interactive queries and ETL jobs.
   - Work is underway to support Spark.
3. **Spark:**
   - Another DAG-based engine with high performance and flexibility.
   - Often chosen for its compatibility with the broader Spark ecosystem.

> Both Tez and Spark are general directed acyclic graph (DAG) engines

Selection of Execution Engine:

```
SET hive.execution.engine=spark;
```

Tasks of the Execution Engine:

1. **Plan Execution:** Executes the logical plan from the query compiler.
2. **Job Generation:** Depending on the chosen execution engine (e.g., MapReduce), it generates the necessary jobs to process data in parallel.
3. **Submission to Hadoop:** It submits these jobs to the Hadoop cluster or other compatible compute environments for execution.
4. **Progress Monitoring:** It continuously monitors the progress of the query execution, providing insights into job completion and overall performance ;

Hive Driver

Hive's execution flow involves several components working together to process HiveQL queries and convert them into executable tasks.

1. Hive Driver

- **Receives Queries:** Accepts queries from various sources like:
  - Web UI
  - Command-Line Interface (CLI)
  - Thrift Clients
  - JDBC/ODBC Drivers
  - Transfers received queries to the **Query Compiler**.
- Tracks query execution progress.

It's responsible for the logical and physical query planning

---

2. Query Compiler

- **Role:** Transforms the HiveQL query into an executable plan.
  1. **Parsing:**
     - Breaks down the query into smaller query blocks and expressions.
     - Checks the syntax of the HiveQL query.
  2. **Semantic Analysis:**
     - Ensures the query is meaningful and valid (e.g., verifying table names, columns, and data types).
     - Checks for access permissions.
  3. **Query Translation:**
     - Converts HiveQL statements into **MapReduce jobs** (or other execution engine tasks like Tez or Spark).

---

3. Query Optimizer

- **Role:** Improves the execution plan to ensure the query runs as efficiently as possible.
  1. **Logical Plan Generation:**
     - Creates a **DAG (Directed Acyclic Graph)** of tasks for execution.
  2. **Optimization Techniques:**
     - **Predicate Pushdown:** Moves filter operations as close to the data source as possible to reduce the amount of data processed.
     - **Join Optimization:** Rearranges and optimizes join operations for faster execution (e.g., using map-side joins when appropriate).

Hive Metastore

The Metastore is the central repository of Hive metadata.

By default, the Metastore service runs in the same JVM as the Hive service and contains an embedded Derby database instance backed by the local disk. This is called the embedded Metastore configuration

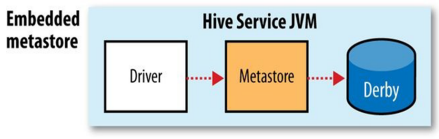
Metastore Components

1. **Service:** A service layer that handles communication with Hive and provides metadata access. Responsible for reading and writing metadata.
2. **Backing Store:** The database where the metadata is actually stored.

   Examples: Derby, MySQL, PostgreSQL.

Configurations of Metastore

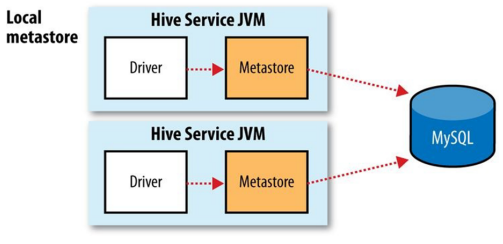
1. **Embedded Metastore:** is a simple way to get started with Hive; however, only one embedded Derby database can access the database files on disk at any one time, which means you can have only one Hive session open at a time that accesses the same Metastore.



Trying to start a second session produces an error when it attempts to open a connection to the Metastore.

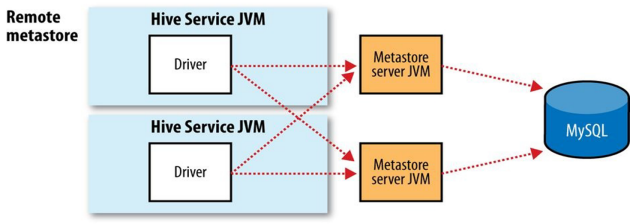
2. **Local Metastore:** Metastore service still runs in the same machine as the Hive service but connects to a database running in a separate process, either on the same machine ~~or on a remote machine.~~

- MySQL is a popular choice for the standalone Metastore.



Runs on the same machine but uses an external database like MySQL.

- **Remote Metastore:** Metastore where one or more Metastore servers run in separate processes to the Hive service. This brings better manageability and security.



Comparison

Feature	Embedded Metastore	Local Metastore	Remote Metastore
Setup Complexity	Simple	Moderate	Advanced
Concurrency	Single session	Multiple sessions	Multiple sessions
Scalability	Low	Moderate	High
Security	Minimal	Moderate	High
Use Case	Testing/Small Apps	Moderate environments	Large-scale systems

1. **Clients:** Allow applications or users to interact with Hive through various interfaces like Thrift, JDBC, and ODBC.
2. **Services:** Provide interaction points with Hive such as CLI, Beeline, and HiveServer 2 for query execution and session management.
3. **Execution Engine:** Executes optimized query plans using engines like MapReduce, Tez, or Spark.
4. **Driver:** Receives queries from clients, transfers them to the query compiler, and executes them.
5. **Metastore:** A central repository for storing metadata about tables, managed through a database like Derby or MySQL.

Hive Table Overview

A **Hive table** consists of two main components:

1. **Data** – The actual content stored in the table, typically on **HDFS** (Hadoop Distributed File System), but it could also reside on other file systems like the **local file system** or **S3**.
2. **Metadata** – The structure and description of the data, such as column names, data types, partitioning information, etc. This metadata is stored in the **Hive Metastore**, not in HDFS.

Types of Hive Tables

When you create a table in Hive, by default Hive will **manage** the data, which means that Hive moves the data into its **warehouse directory.**

> When you load data into a managed table, it is moved into Hive's warehouse directory.

Alternatively, you may create an ***external table*** , which tells Hive to refer to the data that is at an existing location outside the warehouse directory.

Example:

```
CREATE TABLE managed_table (dummy STRING);
LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE managed_table;
```

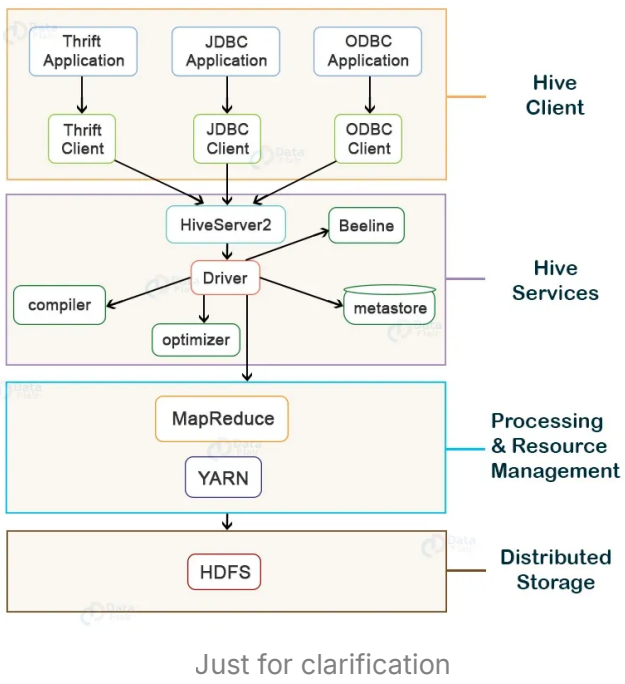
- In this example, Hive will move the file `data.txt` from `/user/tom/data.txt` into the Hive warehouse directory for the `managed_table`, which would be something like `/user/hive/warehouse/managed_table/` on HDFS.

Example:

```
CREATE EXTERNAL TABLE external_table (dummy STRING)
STORED AS TEXTFILE
LOCATION '/user/tom/external_data';
```

- In this case, Hive will use the data at the path `/user/tom/external_data`, but it won't move it to its warehouse directory. If you drop the `external_table`, the data in `/user/tom/external_data` will remain.

Lecture_8-Apache Hive II



Just for clarification

Clients:

- **JDBC Driver**, **Thrift Client**, or **ODBC Driver** connect to Hive.

Hive Services:

- Requests are sent via **CLI**, **Beeline**, or **HWI**.

HiveServer 2:

- Receives the request and forwards it to the **Driver**.

Driver:

- **Compiler** performs **Parsing** and converts SQL into **MapReduce/Tez/Spark java Jobs**.
- Executes **Optimization** such as **Predicate Pushdown** or **Join Optimization**.

Execution Engine:

- Executes the plan on **HDFS** using an execution engine like **MapReduce**.
- Relies on **Metastore** for metadata retrieval.

Execution results are returned to the client.

Tables in Hive

A Hive table is logically made up of the **data being stored** and the **associated metadata describing the layout of the data in the table**.

Data Storage:

- The data typically resides in **HDFS**, although it may reside in any Hadoop file system, including the **local file system** or **S3**.

Metadata Storage:

- Hive stores the **metadata in a relational database** and **not** in, say, **HDFS**.

Hive has two types of tables :

1. **Managed tables .**
2. **External tables**

Note : Hive has its own warehouse directory

Managed table

Managed table : When hive move data to its warehouse directory.

When you load data into a **managed table**, Hive moves the data into its **warehouse directory.**

- For example:

```
CREATE TABLE managed_table (dummy STRING);
LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table;
```

- This will move the file from `hdfs://user/tom/data.txt` into `hdfs://user/hive/warehouse/managed_table`.

> The **load operation** is fast because it is essentially a **move or rename** within the file system.

> Hive uses **Schema on Read**, meaning
>
> - Hive doesn't verify that the data matches the table's declared schema when loading. even for managed tables.
> - If there's a mismatch (e.g., missing fields), it will show up when you query the data (often as **NULL** for missing fields).
> - You can verify correct data parsing by using a simple `SELECT` statement to check a few rows.

Dropping Managed Tables:

- When you drop a managed table using `DROP TABLE`, both the **table** and its **data** (along with metadata) are deleted.

External Table

- In an **external table**, you control the creation and deletion of the data.
- The **location of the external data** is specified when creating the table.

```
CREATE EXTERNAL TABLE external_table (dummy STRING)
LOCATION '/user/tom/external_table';
LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE external_table;
```

In external table, Hive does not managing the data , so hive does not move data to its warehouse directory.

Hive **does not check** if the external location exists when the table is defined. This feature allows you to **lazily load** data into the table after it's created.

Lazily load means **loading data only when needed**, not immediately when the table is created. In **external tables** in Hive, the data location is specified at table creation, but the data is **loaded** only when queried or used.

Dropping External Tables:

- When you **drop an external table**, Hive will delete only the **metadata** and will **leave the data intact**.

Hive uses its **metastore/metadata** to map raw HDFS data to named columns and types.

When to Use:

- Use **managed tables** if you're doing all your processing in Hive.
- Use **external tables** if you need to use Hive alongside other tools on the same dataset.
  - **External tables** are useful when you want to associate **multiple schemas with the same dataset.**

Creating Hive table commands

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
[(col_name data_type [column_constraint_specification] [COMMENT col_comment], ...
    [constraint_specification])]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets
    BUCKETS]
[SKEWED BY (col_name, col_name, ...)
ON ((col_value, col_value, ...) , (col_value, col_value, ...) , ...)
[STORED AS DIRECTORIES]
[[ROW FORMAT row_format]
[STORED AS file_format]
|STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]]
[LOCATION hdfs_path]
[TBLPROPERTIES (property_name=property_value, ...)]
[AS select_statement];
```

Storage formats in Hive:

Text Format:

- **Default** format in Hive.
- Example:

```
CREATE TABLE text_table (id INT, na
me STRING, age INT)
ROW FORMAT DELIMITED FIELDS TERMINA
TED BY ',';
```

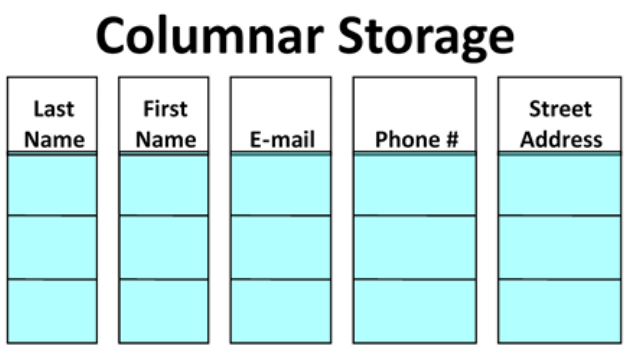
Binary Formats:

- Include Sequence files, Avro, Parquet, RC Files, ORC Files.
- Using a binary format is as simple as changing the STORED AS clause in the CREATE TABLE statement. In this case, the ROW FORMAT is not specified, since the format is controlled by the underlying binary file format.
- Example:

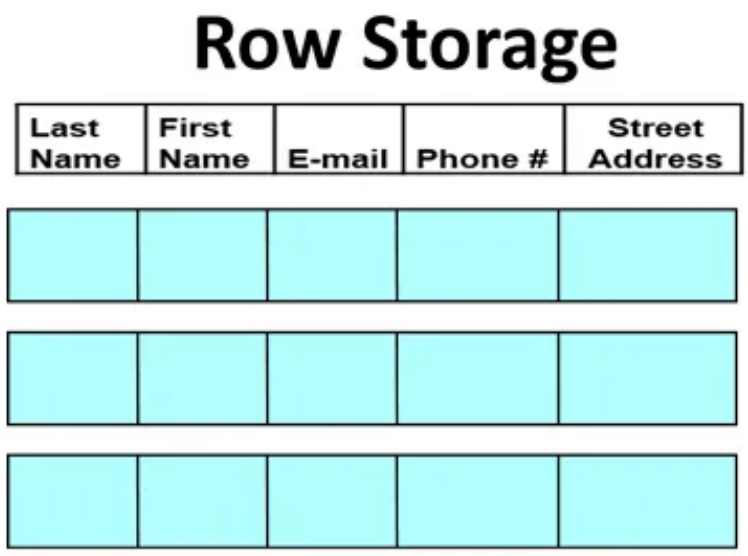
```
CREATE TABLE binary_table (id INT,
name STRING, age INT)
STORED AS ORC;
```

Binary formats can be divided into two categories:

1. **Column-oriented formats** work well when queries access only a small number of columns in the table.



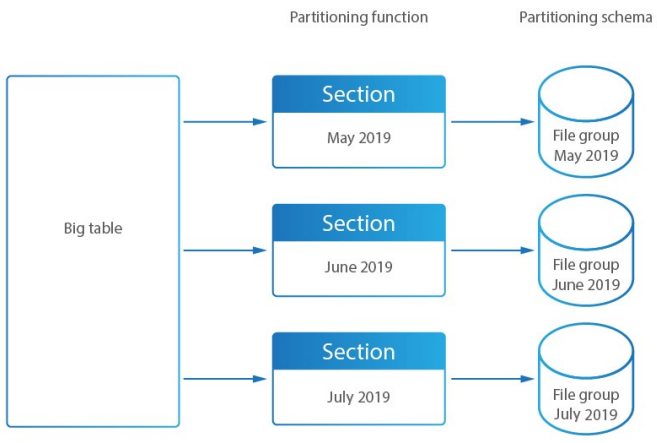
2. **Row-oriented formats** are appropriate when a large number of columns of a single row are needed for processing at the same time.



Criteria	Text Format	Binary Format
Suitable for	Small databases	Large databases
Readability	Human-readable	Not human-readable
Speed	Slower	Faster
Storage size	Larger	Smaller (due to compression)
Schema evolution	Harder	Easier
Parsing	Needs conversion	No conversion needed

Partitioning

Partitioning :Apache Hive organizes tables into partitions for grouping same
type of data together based on a column or partition key. Each table in the hive
can have one or more partition keys to identify a particular partition. Using
partition we can make it faster to do queries on slices of the data.



1. **Non-Partitioned Table**:
    - A standard Hive table where data is stored without any logical separation.

```
CREATE TABLE loadtemp (
    EMPLOYEE_ID INT,
    FIRST_NAME STRING,
    LAST_NAME STRING,
    EMAIL STRING,
    PHONE_NUMBER STRING,
    HIRE_DATE STRING,
    JOB_ID STRING,
    SALARY FLOAT,
    COMMISSION_PCT STRING,
    MANAGER_ID INT,
    DEPARTMENT_ID INT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
```

2. **Partitioned Table**:
    - A Hive table logically divided into partitions based on a specific column(s), improving query performance by scanning only relevant data.

```
CREATE TABLE EMPDATA (
    EMPLOYEE_ID INT,
    FIRST_NAME STRING,
    LAST_NAME STRING,
    EMAIL STRING,
    PHONE_NUMBER STRING,
    HIRE_DATE STRING,
    JOB_ID STRING,
    SALARY FLOAT,
    COMMISSION_PCT STRING,
    MANAGER_ID INT
)
PARTITIONED BY (DEPARTMENT_ID INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
```

3. **Load Data into Non-Partitioned Table**:
    - Load data into the non-partitioned table `loadtemp`.

```
LOAD DATA LOCAL INPATH '/home/cloudera/Desktop/employees.csv'
OVERWRITE INTO TABLE loadtemp;
```

4. **Enable Dynamic Partitioning**:
    - Allow Hive to dynamically create partitions during data insertion by setting the partition mode to nonstrict.

```
SET hive.exec.dynamic.partition.mode=nonstrict;
```

5. **Load Data into Partitioned Table**:

- Transfer data from the non-partitioned table `loadtemp` to the partitioned table `EMPDATA`.

```
INSERT OVERWRITE TABLE EMPDATA PARTITION(DEPARTMENT_ID)
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER,
       HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, DEPARTMENT_ID
FROM loadtemp;
```

Query Optimization:

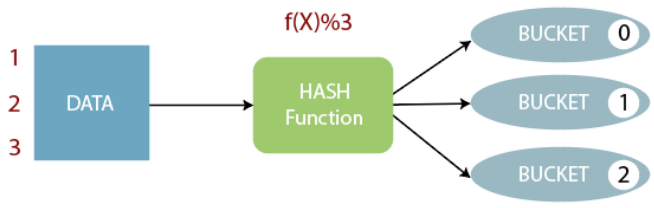
- When querying a **non-partitioned table**, Hive performs a **full table scan** and applies the filter afterward.

- When querying a **partitioned table**, Hive directly scans only the relevant partition, skipping unnecessary data.

Query	Non-Partitioned Table	Partitioned Table
`SELECT * FROM EMPDATA WHERE DEPARTMENT_ID=50`	Full table scan	Identify partition (DEPARTMENT_ID=50)
	Apply `WHERE` filter after scan	Scan only the relevant partition (DEPARTMENT_ID=50)

Hive Bucketing

Divides tables or partitions into smaller, fixed buckets using a hash function on a specific column for better query performance.



- Bucketing can be created on just one column, you can also create bucketing on a partitioned table to further split the data to improve the query performance of the partitioned table.
- Each bucket is stored as a separate file in the table or partition directory on HDFS.
- Records with the same hash value for the bucketed column are placed in the same bucket.

Advantages:

Efficient Query Performance:

- Helps avoid too many directories (as in partitions) by allowing the user to control the number of buckets.
- Improves performance by 2–3x on large tables compared to non-bucketed tables.

with the partition you cannot control the number of partitions as it creates a partition for every distinct value of the partitioned column; which ideally creates a subdirectory for each partition inside the table directory on HDFS .

Controlled Distribution:

- Unlike partitions, you control the number of buckets, avoiding excessive subdirectories on HDFS, which can overwhelm the NameNode.

Improved Sampling:

- Enables efficient testing or refinement of queries on a subset of data.

```
CREATE TABLE users (
    user_id INT,
    name STRING,
    age INT,
    gender STRING
)
CLUSTERED BY (user_id) INTO 4
BUCKETS;
```

```
INSERT INTO TABLE users
VALUES
(1, 'Alice', 25, 'Female'),
(2, 'Bob', 30, 'Male'),
(3, 'Charlie', 28, 'Male'),
(4, 'David', 35, 'Male'),
(5, 'Eva', 22, 'Female'),
(6, 'Frank', 40, 'Male'),
(7, 'Grace', 27, 'Female'),
(8, 'Henry', 32, 'Male'),
(9, 'Ivy', 29, 'Female'),
(10, 'Jack', 34, 'Male'),
(11, 'Karen', 31, 'Female'),
(12, 'Leo', 26, 'Male');
```

The data is automatically distributed into the specified bucket based on hash function. To populate the bucketed table, we need to set the hive.enforce.bucketing property to true

- **Configuration**: Set `hive.enforce.bucketing=true` to populate a bucketed table correctly.

- Bucket 0: (1, 'Alice', 25, 'Female'), (5, 'Eva', 22, 'Female'), (9, 'Ivy', 29, 'Female')
- Bucket 1: (2, 'Bob', 30, 'Male'), (6, 'Frank', 40, 'Male'), (10, 'Jack', 34, 'Male')
- Bucket 2: (3, 'Charlie', 28, 'Male'), (7, 'Grace', 27, 'Female'), (11, 'Karen', 31, 'Female')
- Bucket 3: (4, 'David', 35, 'Male'), (8, 'Henry', 32, 'Male'), (12, 'Leo', 26, 'Male')

Example Query:

```
SELECT * FROM users WHERE user_id=5;
```

Hive identifies the bucket for user_id=5 (e.g., bucket 0) and scans only that bucket, boosting performance.

The reasons for using bucketing are :

1. enable more efficient queries (filtering and joining data based on bucketed column)
2. make sampling more efficient. When working with large datasets, it is very convenient to try out queries on a fraction of your dataset while you are in the process of developing or refining them

Lecture 9-Big Data processing Engine (Spark)

What is Apache Spark ?

Apache Spark is a cluster computing platform designed to be fast and general purpose

Apache Spark is a separate project from Hadoop and can process data stored in various sources, not just HDFS (Hadoop Distributed File System).

- **Spark** improves the **MapReduce** model to support a variety of computations, such as interactive queries and stream processing. This enhances **Spark**'s ability to handle large datasets more quickly and flexibly.

Spark's in-memory processing is well-suited for iterative algorithms because it keeps data in

- Speed is crucial because it allows interactive data exploration rather than waiting for minutes or hours for results.
- One of **Spark**'s key features is the ability to perform computations in memory, significantly boosting performance. Even when tasks require disk storage, **Spark** remains more efficient than traditional **MapReduce**, especially for complex applications.

- **Spark** is designed to handle a wide range of workloads that previously required separate distributed systems, such as:
  - **Batch applications** (processing large datasets in bulk).
  - **Iterative algorithms** (repeated computations, useful in machine learning).
  - **Interactive queries** (queries that require immediate feedback).
  - **Streaming data** (real-time processing of continuous data).
- By supporting all these workloads within a single system, Spark enables the combination of different types of data processing. This reduces the management burden of maintaining separate tools, which is often required in real-world data analysis workflows.

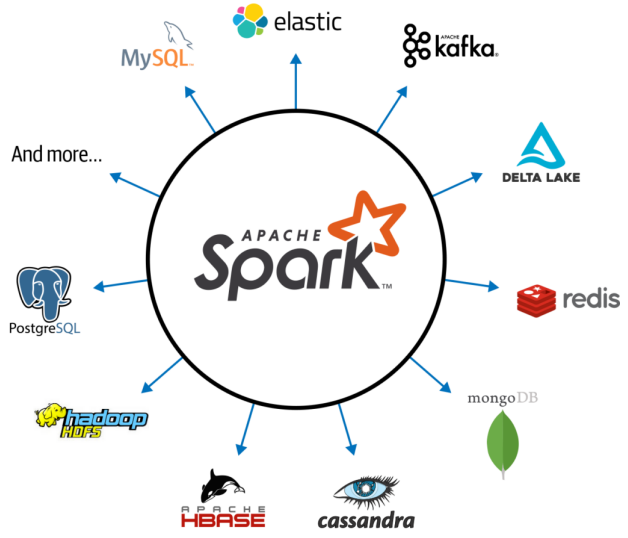
- Rather than needing separate tools for each type of workload, **Spark** provides a unified solution, reducing the complexity of managing multiple systems.
- **Spark** offers easy-to-use APIs in popular programming languages such as **Python**, **Java**, **Scala**, and **SQL**, making it accessible to a broad range of users, whether beginners or experienced programmers.
- **Spark** integrates well with other big data tools. It can run on **Hadoop clusters** and work with any **Hadoop data source**, like **Cassandra** (a distributed database).

- Faster results from analytics has become increasingly important
- Apache Spark is a computing platform designed to be fast and general-purpose, and easy to use

Speed	• In-memory computations • Faster than MapReduce for complex applications on disk
Generality	• Covers a wide range of workloads on one system • Batch applications (e.g. MapReduce) • Iterative algorithms • Interactive queries and streaming
Ease of use	• APIs for Scala, Python, Java, R • Libraries for SQL, machine learning, streaming, and graph processing • Runs on Hadoop clusters or as a standalone • including the popular MapReduce model

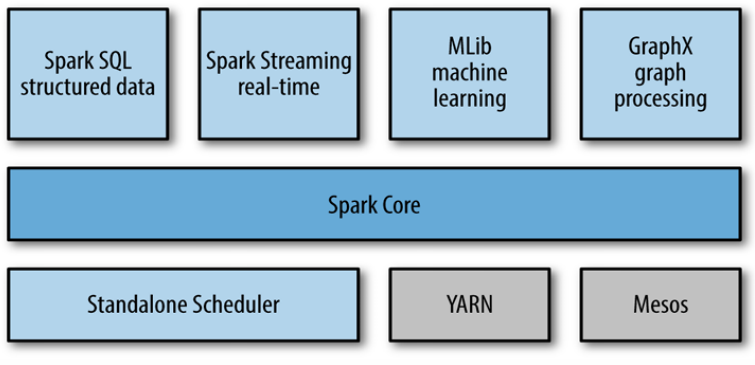
What uses Spark and why?

- Data scientist
  - Analyze and model the data to obtain insight using ad-hoc analysis
  - Transforming the data into a useable format
  - Statistics, machine learning, SQL
- Data engineers
  - Develop a data processing system or application
  - Inspect and tune their applications
  - Programming with the Spark's API
- Everyone else
  - Ease of use
  - Wide variety of functionality
  - Mature and reliable



Spark unified stack

- Spark is a **computational engine** that handles scheduling, distributing, and monitoring applications across a computing cluster. Its core engine is fast and general-purpose, enabling Spark to power specialized components like SQL and machine learning.
- These components are tightly integrated, allowing them to work together seamlessly, similar to libraries in a software project.



Deployment Modes for Apache Spark

1. **Local Mode**:
   - **local[n]**: Runs Spark on a single machine using `n` cores.
   - **local[*]**: Runs Spark on a single machine utilizing all available cores.
   - Spark is installed and run on your local machine without the need for a distributed environment.

1. **Standalone Mode**:
   - Spark operates as a standalone cluster with a master node and multiple worker nodes.
   - Example: If you have 10 machines with Hadoop installed and 10 machines without Hadoop, Spark can run across both types in a multi-node setup. The key difference here is that in **Standalone mode**, Spark is fully responsible for distributing the jobs.

> In this mode, Spark itself handles task distribution and resource management, without relying on YARN or other cluster managers.

1. **Cluster Managers**:
   - **Apache Mesos**: A general-purpose cluster manager that allocates resources across multiple applications and supports Spark.
   - **Hadoop YARN**: A resource manager in Hadoop that allows Spark to run alongside other Hadoop ecosystem tools.
     - In this mode, YARN is responsible for distributing tasks and managing resources within the cluster.

Benefits of Tight Integration:

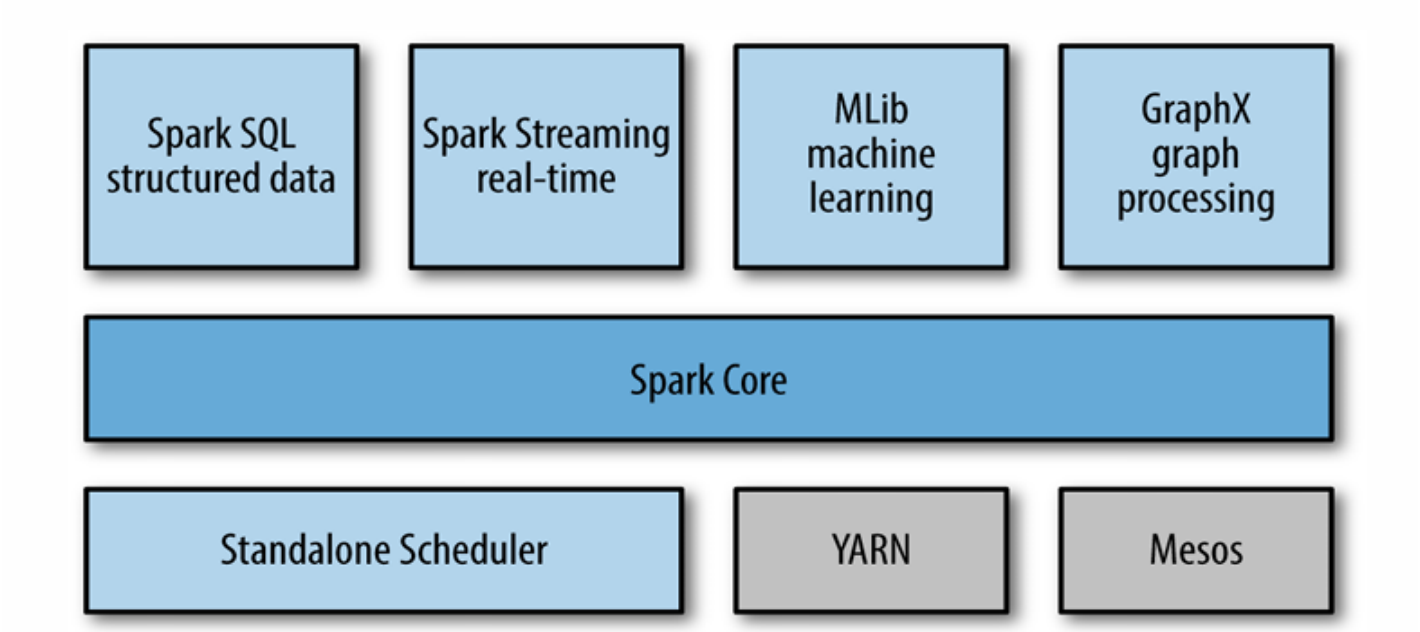
1. **Improved Efficiency Across Components**: All components, like SQL and machine learning, benefit from improvements in the core engine.

   For example, when Spark's core engine is optimized, libraries like SQL or machine learning also automatically speed up.

2. **Cost Savings**: Instead of managing multiple independent software systems (5-10 systems), organizations only need to run one system. This reduces costs related to deployment, maintenance, testing, and support. Furthermore, when new components are added to Spark, organizations can immediately try them without the need for deploying new software.

3. **Seamless Combination of Processing Models**: Spark's integration allows for the creation of applications that combine different processing models.

   For example:
   - A real-time machine learning application can classify data as it’s ingested from streaming sources.
   - Analysts can query the data in real time via SQL.
   - Data engineers and scientists can perform ad-hoc analysis via the Python shell.
   - At the same time, standalone batch applications can access the data.
   - IT teams only need to manage one system, simplifying maintenance.



Spark Core:

Basic functionality: Spark Core provides fundamental features like task scheduling, memory management, fault recovery, and interacting with storage systems.

RDDs (Resilient Distributed Datasets): RDDs are the main programming abstraction in Spark. They represent collections of items distributed across many compute nodes, and these collections can be manipulated in parallel.

- **APIs**: Spark Core provides APIs to build and manipulate these RDDs, making it a key component for distributed data processing.

Spark SQL:

SQL Interface: Spark SQL allows working with structured data, providing a SQL interface to Spark. It supports various data formats, such as Hive tables, Parquet, and JSON.

Hive Query Language (HQL): You can query data using SQL or HQL (Apache Hive's version of SQL).

- **Programmatic Data Manipulation**: Spark SQL integrates SQL queries with programmatic data manipulations using **RDDs** in languages like Python, Java, and Scala, enabling complex analytics within a single application.

Spark SQL:

A tool for working with structured data using **SQL**, supporting various formats like JSON and Parquet, and combining SQL queries with programming.

Spark Streaming:

Real-Time Data Processing: Spark Streaming allows processing of live data streams, such as log files or message queues.

Fault Tolerance: Like Spark Core, Spark Streaming is designed to provide fault tolerance, throughput, and scalability for stream processing.

Streaming API: It provides an API similar to the RDD API for stream data, making it easy for programmers to manipulate real-time data streams, as well as data stored in memory or on disk.

Spark Streaming:

A tool for real-time processing of **streaming** data, with support for fault tolerance and scalability.

Spark MLlib:

Machine Learning Library: Spark MLlib is a library for machine learning (ML) algorithms, including classification, regression, clustering, and collaborative filtering.

- **Additional Functionality**: It also includes features like model evaluation and data import, along with low-level ML primitives such as a generic gradient descent optimization algorithm.
- **Scalability**: MLlib is designed to scale out across a cluster, supporting large-scale machine learning tasks.

> **Spark MLlib**:
>
> A **machine learning** library offering algorithms for classification, clustering, regression, and recommendations, designed to scale for big data.

Spark GraphX:

Graph Processing: GraphX is a library for working with graphs, such as a social network's friend graph, and performing graph-parallel computations.

- **RDD API Extension**: Like Spark Streaming and Spark SQL, GraphX extends the RDD API, enabling users to create directed graphs with properties attached to vertices and edges.
- **Graph Algorithms**: GraphX provides operators for graph manipulation (e.g., subgraph, mapVertices) and algorithms like PageRank and triangle counting.

> **Spark GraphX**:
>
> A tool for analyzing and processing **graphs** using algorithms like PageRank, allowing creation of directed graphs with customizable vertex and edge properties.

Cluster Manager:

Spark scales efficiently from a single node to thousands of compute nodes.

- It supports various cluster managers:
  - **Standalone Scheduler**: A simple built-in cluster manager for quick setups.
  - **Hadoop YARN**: Integrates with Hadoop clusters for resource management.
  - **Apache Mesos**: A general-purpose cluster manager for running distributed applications.

For new installations, the Standalone Scheduler is a good starting point. However, if a Hadoop YARN or Mesos cluster is already set up, Spark can run on these systems.

Storage Layers for Spark:

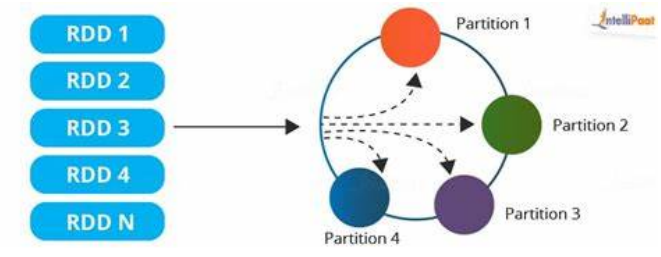
- Spark can create distributed datasets from files stored in:
  - **HDFS** (Hadoop Distributed File System).
  - Other storage systems supported by Hadoop APIs, such as **Amazon S3**, **Cassandra**, **Hive**, **HBase**, or even the local filesystem.

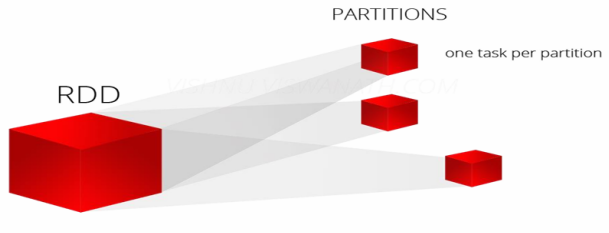
> It's important to remember that Spark does not require Hadoop; it simply has support for storage systems implementing the Hadoop APIs.

- Supported file formats include **text files**, **SequenceFiles**, **Avro**, **Parquet**, and any **Hadoop InputFormat**.

Spark Core and RDDs (Resilient Distributed Datasets)

RDD is a distributed collection of elements in Spark. These elements can be any type of objects in **Python**, **Java**, or **Scala**, including user-defined classes.



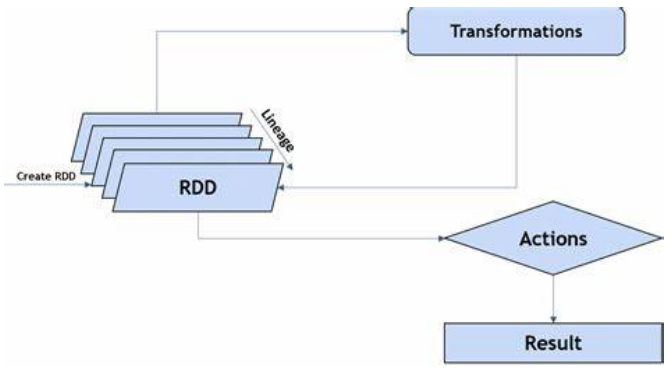


- **RDDs are immutable**: Once created, an RDD **cannot be changed.** If you want to modify data, you need to perform transformations that result in a new RDD.

Partitions and Distribution:

An RDD is split into **multiple partitions**, which may be computed in parallel on different nodes in the cluster. This allows Spark to process large datasets efficiently by utilizing the resources of multiple machines.

All work in Spark is represented as creating new RDDs, transforming existing RDDs, or performing operations on RDDs to compute results.



Immutability of RDDs:

- RDDs are **immutable**, meaning once you create an RDD, you cannot change it directly. To make changes, you need to use **transformations**.
- When you apply transformations to an RDD, the resulting data is saved in a **new RDD**, preserving the original data.

Lineage Graph:

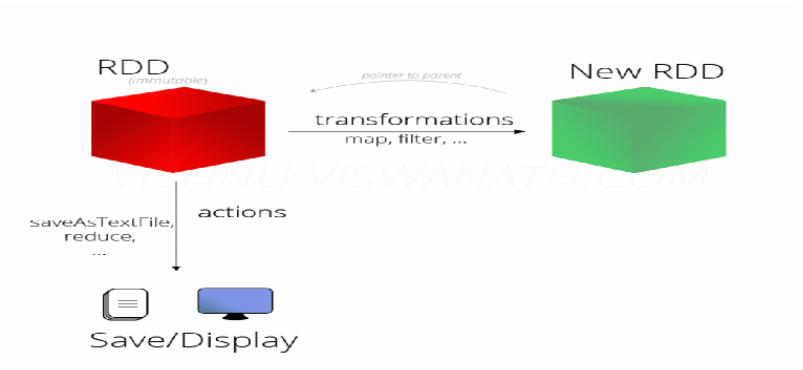
- Spark tracks the sequence of transformations applied to RDDs using a **lineage graph**. This allows Spark to rebuild lost data from its original dataset in case of failure, providing fault tolerance.

Operations on RDDs:

There are two types of operations in Spark for RDDs:

Transformations:

Transformations create a new RDD from an existing one. Examples include operations like **map**, **filter**, **flatMap**, etc.



Actions:

Actions are operations that compute a result from an RDD and return it to the driver program or save it to an external storage system (e.g., **HDFS**, **S3**).

- Examples of actions include **collect**, **count**, **reduce**, **saveAsTextFile**, etc.

Key Difference:

- **Transformations** modify data and generate new RDDs.
- **Actions** trigger the execution of transformations and provide output (either by returning it to the driver program or saving it to storage).

When you apply transformations like `map` or `filter`, Spark doesn't start processing until an **action** (like `collect` or `count`) is called.

> **Lazy Evaluation in Spark** means that **transformations** on **RDDs** are not executed immediately.

How It Works:

- Spark gathers all transformations into a **DAG (Directed Acyclic Graph)** and only starts executing when you ask for a result.
- This allows Spark to optimize and execute operations more efficiently.

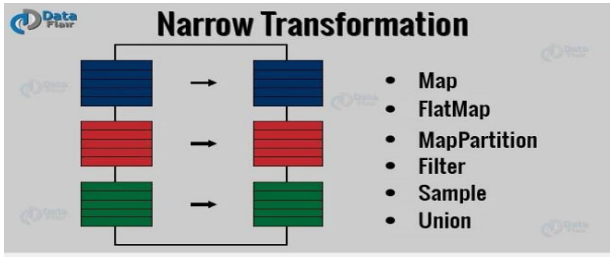
This can be somewhat counterintuitive for new users, but may be familiar for those who have used functional languages such as Haskell or LINQ-like data processing frameworks

---

Transformations

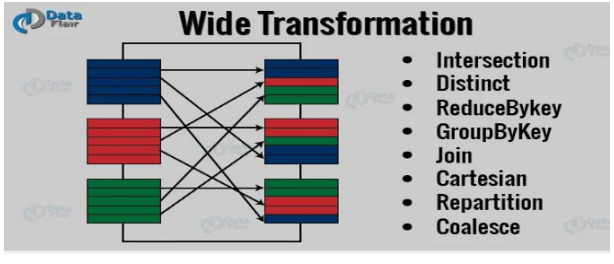
Types of Transformations in Spark

1. **Narrow Transformations**:
    - In narrow transformations, the data required to compute the results from a partition is contained within a single partition of the parent RDD.
    - **No data movement** between partitions is required for narrow transformations.
    - Examples of narrow transformations include:
        - `map()`
        - `filter()`
        - `flatMap()`
    - These transformations only operate within a single partition, so there's no shuffle or movement of data.



2. **Wide Transformations**:
    - In wide transformations, the data required for computation may span multiple partitions of the parent RDD.
    - These transformations involve **data movement between partitions** to calculate the result.
    - Examples of wide transformations include:
        - `groupByKey()`
        - `reduceByKey()`
        - `join()`

    These transformations result in **shuffling of data**, which is why they are called **shuffle transformations**.



Narrow transformations: No data movement between partitions (e.g., `map`, `filter`).

Wide transformations: Data movement is required between partitions (e.g., `groupByKey`, `reduceByKey`), resulting in **shuffling**.

Transformation	Meaning
map(func)	Applies a function to each element and returns a new dataset.
filter(func)	Returns a new dataset with elements that satisfy the given condition.
flatMap(func)	Similar to `map`, but each input item can map to 0 or more output items.
mapPartitions(func**)**	Applies a function on each partition (block) of the RDD.
mapPartitionsWithIndex(func)	Similar to `mapPartitions`, but also provides partition index.
sample(withReplacement, fraction, seed)	Samples a fraction of the data with or without replacement.
union(otherDataset)	Returns a dataset with the union of elements from both datasets.
intersection(otherDataset)	Returns the intersection of elements from both datasets.
distinct([numPartitions**])**	Returns a dataset with distinct elements.
groupByKey([numPartitions**])**	Groups data by key, returning (K, Iterable) pairs.
reduceByKey(func, [numPartitions])	Aggregates values for each key using a reduce function.

aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions])	Aggregates values by key using combine functions.
sortByKey([ascending], [numPartitions])	Sorts key-value pairs by key.
join(otherDataset, [numPartitions])	Joins two datasets on the key, returning (K, (V, W)) pairs.
cogroup(otherDataset, [numPartitions])	Joins datasets and groups values for each key into iterables.
cartesian(otherDataset**)**	Returns a dataset of all pairs of elements from two datasets.
coalesce(numPartitions**)**	Reduces the number of partitions for more efficient operations.
repartition(numPartitions**)**	Reshuffles data randomly to create more or fewer partitions.
repartitionAndSortWithinPartitions(partitioner**)**	Reshuffles and sorts data within each partition for more efficient processing.

Actions

Action	Meaning
reduce(func**)**	Aggregates dataset elements using a function that combines two elements into one.
collect()	Returns all elements of the dataset as an array to the driver program.
count()	Returns the number of elements in the dataset.
first()	Returns the first element of the dataset (similar to `take(1)`).
take(n)	Returns an array of the first `n` elements of the dataset.
takeSample(withReplacement, num, [seed])	Returns a random sample of `num` elements from the dataset, with or without replacement.
takeOrdered(n, [ordering**])**	Returns the first `n` elements, sorted by natural order or custom comparator.
saveAsTextFile(path**)**	Writes the dataset to a text file in the specified path in HDFS or local filesystem.
saveAsSequenceFile(path**)**	Writes the dataset as a Hadoop SequenceFile, available on RDDs of key-value pairs.
saveAsObjectFile(path**)**	Writes the dataset using Java serialization, retrievable via `SparkContext.objectFile()`.
countByKey()	Available on RDDs of (K, V), counts occurrences of each key and returns a hashmap of counts.
foreach(func)	Applies a function on each element for side effects (e.g., updating an accumulator).

Lecture 10 Big Data processing Engine (Spark) II

Broadcast Variables

Broadcast variables are shared variables used to distribute data efficiently across all nodes in the cluster. They are similar to the ***distributed cache*** used in Hadoop MapReduce.

- **Data Distribution**: Spark uses a BitTorrent-like protocol to broadcast variables across the cluster:
  - The driver splits the data into blocks and sends them.
  - Each receiver (or "leecher") fetches the blocks to its local directory.
  - Once a block is fully received, the "leecher" becomes a source for that block for other receivers, reducing the load on the driver.

- **Creation**: Broadcast variables are created using `SparkContext.broadcast(v)`:

```
Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3});
```

- **Accessing**: To get the value of a broadcast variable, use `broadcastVar.value()`:

```
broadcastVar.value();
```

RDD persistence

Persisting an RDD means storing a dataset in memory across operations. Each node in the cluster keeps the parts of the dataset it computes in memory, so it can reuse them in future actions without needing to recompute them.

> In Spark, **RDD persistence** (or caching) is an important feature that helps speed up operations.

- This persistence makes future operations faster, sometimes up to **10 times faster**. It's especially useful for iterative algorithms and when you need quick access to data in interactive applications.
- You can use **persist()** or **cache()** methods to store RDDs in memory.
- **Fault tolerance**: If any part of the RDD is lost (e.g., due to a failure), Spark can automatically rebuild it using the transformations that created it.

Storage Level	Meaning
MEMORY_ONLY	Stores RDD as deserialized Java objects in the JVM. If the RDD doesn’t fit in memory, some partitions won’t be cached and will be recomputed when needed.
MEMORY_AND_DISK	Stores RDD as deserialized Java objects in the JVM. If the RDD doesn’t fit in memory, partitions not fitting in memory are stored on disk.
MEMORY_ONLY_SER (Java, Scala)	Stores RDD as serialized Java objects (one byte array per partition). More space-efficient but CPU-intensive to read.
MEMORY_AND_DISK_SER (Java, Scala)	Similar to MEMORY_ONLY_SER, but partitions not fitting in memory are stored on disk instead of being recomputed.
DISK_ONLY	Stores the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicates each partition on two cluster nodes for fault tolerance.

RDD, DataFrame, and Dataset in Apache Spark:

1. RDD (Resilient Distributed Datasets)

RDD is the fundamental data structure in Spark that represents a collection of elements partitioned across the nodes of a cluster.

RDDs can be created from various data sources like HDFS, local file systems, or relational databases.

You would use **RDD** if you need more control over how the data is processed or if the data is unstructured, but if the data is already well-structured and you need better performance and an easier interface, you would use **DataFrame**.

2. DataFrame

A **DataFrame** is a distributed collection of data organized into named columns, **similar to a table** in a relational database.

It can be created from sources like RDDs, CSV/JSON/Parquet files, Hive tables, or external databases.

DataFrames have a schema (i.e., column names and types), and each row represents one record or observation.

DataFrames support higher-level operations such as filtering, aggregating, joining, and grouping data.

DataFrames benefit from Spark's optimizations (like Catalyst query optimization) and are faster for most operations than RDDs.

3. Dataset

A **Dataset** is a distributed collection of data with strong typing, compile-time type safety, and object-oriented programming features. It is a **strongly-typed version of a DataFrame.**

Datasets in spark are created from case classes (in Scala) or Java classes (in Java).

Encoding: Datasets use a specialized **Encoder** to serialize objects, unlike RDDs which rely on Java serialization or Kryo.

Similar to DataFrames but provide the benefits of strong typing, so you can access the fields directly like any object in your programming language.

Like DataFrames, Datasets leverage Spark's optimizations and can be more efficient than RDDs.

API level refers to the degree of abstraction and interface a library or framework provides for developers to interact with data, determining the balance between control and ease of use.

- **Low-level API** provides more control over execution details but requires more effort from the developer.
- **High-level API** simplifies data manipulation and provides optimized performance through automatic improvements.

Context	RDD	DataFrame	Dataset
Interoperability	**Can be easily converted to DataFrames** and vice versa using `toDF()` and `rdd()`	**Can be easily converted to RDDs and Datasets** using `rdd()` and `as[]`	**Can be easily converted to DataFrames** using `toDF()` and to RDDs using `rdd()`
Type safety	Not type-safe	Not type-safe, errors accessing columns are **detected at runtime**	**Type-safe,** provides **compile-time** type checking to catch errors early
Performance	**Low-level API** with more control over data, but fewer optimizations compared to DataFrame and Dataset	Optimized for performance with **high-level API**, Catalyst optimizer, and code generation	**Faster than DataFrame** because it uses JVM bytecode generation for data operations
Memory Management	**Full control** over memory management, data can be cached in memory or disk as per the user's choice	Optimized memory management with Spark SQL **optimizer to reduce memory usage**	Supports most available data types
Use Cases	Suitable for **low-level data processing** and batch jobs that require fine-grained control over data	**Suitable for structured and semi-structured** data processing with a higher level of abstraction	**Suitable for high-performance batch and stream processing** with strong typing and functional programming
Serialization	Uses **Java serialization** or Kryo when distributing data or writing to disk	DataFrames use a generic **encoder** that can handle any object type	Datasets use specialized **encoders** that are optimized for performance
APIs	Provides a **low-level API** that requires more code to perform transformations and actions on data	Provides a **high-level API** that makes it easier to perform transformations and actions on data	Provides a **richer set of APIs,** supporting both functional and object-oriented programming paradigms
Schema Enforcement	**Does not have an explicit schema**, often used for unstructured data	**Enforces schema at runtime** with explicit schema defining data and types	**Enforces schema at compile-time** with strong typing and checks for errors in data types or structures
Programming Language Support	APIs are available in **Java**, **Scala**, Python, and R, providing flexibility for developers	Available in 4 languages (**Java**, Python, **Scala**, R)	Available only in **Scala** and **Java**
Optimization	No built-in optimization engine in RDD	Uses Catalyst optimizer for query optimization	Includes Catalyst optimizer for query plan optimization
Data Types	**Suitable for structured and semi-structured** data processing with higher-level abstraction	DataFrames support most available data types	Datasets support **all data types** available in DataFrames, plus user-defined types

Low-level data processing means directly manipulating raw data with basic operations, giving the developer full control but requiring more effort and manual performance management.

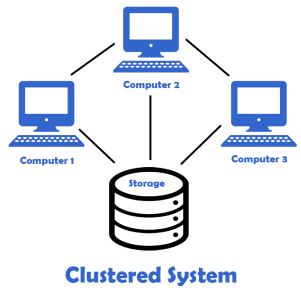
Serialization is the process of converting objects in memory into a format that can be stored or transmitted over a network.

- **RDDs**: You need to manually control how data is serialized, making it more complex to work with.
- **DataFrames**: Simplifies working with structured data (like tables) and handles serialization automatically.
- **Datasets**: Similar to **DataFrames**, but with **performance optimizations** through **type safety** and specialized encoders.

Spark Architecture

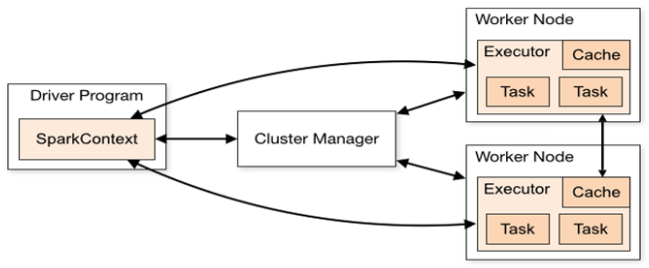
Single machines do not have enough power and resources to perform computations on huge amounts of information (or the user probably does not have the time to wait for the computation to finish).

A cluster, which is a group of computers, pools resources from many machines, enabling you to treat them as a single powerful system. However, simply having multiple machines is not enough; a framework is required to manage and coordinate tasks across the cluster. **Apache Spark** does this by managing and coordinating task execution on a cluster.



Cluster Manager:

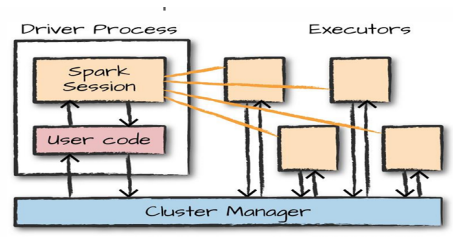
- Spark relies on a cluster manager to allocate resources across the machines. It could be Spark's **standalone cluster manager**, **YARN**, or **Mesos**.
- The **cluster manager** controls physical machines and allocates resources to Spark applications, ensuring the cluster's resources are efficiently utilized.



Spark Applications:

- A **Spark Application** is composed of two main components: the **driver** and the **executors**.

1. **Driver**:
    - The **driver** process **runs the main function** of the application. It sits on one of the nodes in the cluster and handles several important tasks:
        - **Maintaining Information**: It keeps track of the application's state and responds to user inputs or requests.
        - **Task Distribution**: The driver analyzes, distributes, and schedules work across the **executors**.
        - **Central Control**: The driver is critical to the Spark application as it **manages the overall execution** and information flow throughout the application lifecycle.
2. **Executors**:
    - **Executors** are the processes that **actually perform the computation**. Each executor is responsible for:
        - **Executing Code**: Running the code assigned by the driver.
        - **Reporting Status**: Sending back information about the execution status to the driver.



The executors are where the core computation of the application happens.

The **cluster manager** controls the allocation of resources across the physical machines in the cluster.

It can support multiple Spark Applications running simultaneously, ensuring resource allocation and task execution are handled efficiently across all applications.

Deployment Modes for Apache Spark

1. **Local Mode**:
   - **local[n]**: Runs Spark on a single machine using `n` cores.
   - **local[*]**: Runs Spark on a single machine utilizing all available cores.
   - Spark is installed and run on your local machine without the need for a distributed environment.
2. **Standalone Mode**:
   - Spark operates as a standalone cluster with a master node and multiple worker nodes.
   - Example: If you have 10 machines with Hadoop installed and 10 machines without Hadoop, Spark can run across both types in a multi-node setup. The key difference here is that in **Standalone mode**, Spark is fully responsible for distributing the jobs.

   In this mode, Spark itself handles task distribution and resource management, without relying on YARN or other cluster managers.
3. **Cluster Managers**:
   - **Apache Mesos**: A general-purpose cluster manager that allocates resources across multiple applications and supports Spark.
   - **Hadoop YARN**: A resource manager in Hadoop that allows Spark to run alongside other Hadoop ecosystem tools.
     - In this mode, YARN is responsible for distributing tasks and managing resources within the cluster.

Here are the key points to understand about Spark Applications at this point:

Cluster Manager:

Spark employs a **cluster manager** that is responsible for managing resources and keeping track of the available computing resources (memory, CPUs, etc.) in the cluster.

Driver Process:

The **driver process** is responsible for executing the Spark application. It runs the main program and coordinates the execution of tasks across the **executors** (worker processes) in the cluster.

- The driver schedules work (tasks) and distributes them to the executors.
- It also maintains the application state and monitors the progress of the computations.

Spark Driver:

- It is the program that **declares the operations(transformations and actions) on RDDs** of data and submits to the master
- It is the program that creates the SparkContext, connecting to a given Spark Master "we can named as session "
- It prepares the context and declares the operations on the data using RDD transformations and actions
  It submits the serialized RDD graph to the master.
- The master creates tasks out of it and submits them to the workers for execution. It coordinates the different job stages
- The workers is where the tasks are actually executed. They should have the resources and network connectivity required to execute the operations requested on the RDDs

---

Spark Session:

- Introduced in Spark 2.0, the **Spark Session** is the main entry point for running Spark code.
- It is used to interact with data and create **DataFrames**.

> **Spark Session** replaces the old **SQL Context** and **Hive Context**, providing a unified entry point to work with structured and semi-structured data.
>
> It supports reading from and writing to various data sources like **Avro**, **Parquet**, **JSON**, and **JDBC**.

Creating Spark Session:

- **SparkSession.builder()** method is used to create a **SparkSession.Builder** object.
- **appName()** sets the name of the Spark application, useful for monitoring and debugging.
- **config()** method is used to set configuration options (e.g., memory allocation, number of cores).

Data Operations:

- Once the Spark Session is created, it allows reading data from sources like **CSV**, **JSON**, and **Parquet** files, and creating a **DataFrame**.
- The **DataFrame API** can be used for operations like filtering, aggregation, and joining data.

Spark Context:

Spark Context is the entry point to Spark, available since **Spark 1.x**, and is used to create **RDDs**, **accumulators**, and **broadcast variables** on the cluster.

- Since **Spark 2.0**, most methods in **Spark Context** are also available in **Spark Session**.
- The default object **sc** is available in **spark-shell** and can be created programmatically using the **SparkContext** class.

Spark Context vs Spark Session:

Feature	Spark Session	Spark Context
Preferred for	Working with **DataFrames** and **Datasets**, as it provides a consistent and simple interface.	Lower-level operations like creating **RDDs**, **accumulators**, and broadcasting variables.
Creation	Can be created multiple times within an application.	Is a singleton and can only be created once per application.
Configuration	Does not have a separate configuration object; configurations are set using `.config` method.	Created using **SparkConf**, which allows setting various configurations for Spark.
Methods	Provides methods for creating **DataFrames** and **Datasets**, as well as reading and writing data.	Provides methods for creating **RDDs**, **accumulators**, and broadcasting variables.
Purpose	Used to access data in Spark and perform operations on it (e.g., filtering, aggregation).	Used to access the underlying Spark environment and perform lower-level operations like task scheduling.

- **Spark Session** is used for higher-level operations like working with **DataFrames** and **Datasets**, while **Spark Context** is used for lower-level operations, mainly with **RDDs**.
- **Spark Session** is more flexible and preferred in modern Spark applications (since Spark 2.0), while **Spark Context** is limited to managing the Spark environment and lower-level tasks.

Spark SQL

- **Spark SQL** is a Spark module designed for processing **structured data.**
- Unlike the basic **RDD** API, **Spark SQL** leverages additional information about the data's structure and the computation being performed, enabling optimization at the execution level.
- Spark SQL allows interaction through multiple APIs, including **SQL** and **Dataset** API, providing flexibility.
- The same execution engine is used regardless of whether you're using **SQL** or the **Dataset API**, allowing easy switching between APIs based on convenience and clarity for expressing transformations.

Save Modes in Spark SQL:

When saving data, Spark SQL provides several **Save Modes** that control how existing data is handled if it already exists:

1. **SaveMode.ErrorIfExists**: Raises an error if data already exists.
2. **SaveMode.Append**: Appends the new data to the existing dataset without removing the old data.
3. **SaveMode.Overwrite**: Deletes the existing data and writes the new data.
4. **SaveMode.Ignore**: Ignores the save operation if data already exists, preventing any changes.

Important Notes:

- These save modes do not support locking, and **Overwrite** will delete existing data before writing the new data.
- They are not atomic operations, so caution is needed when overwriting or appending data.

Spark SQL Save Modes:

Save Mode Type	Meaning
SaveMode.ErrorIfExists (default)	When saving a DataFrame to a data source, if data already exists, an exception will be thrown.
SaveMode.Append	When saving a DataFrame to a data source, if data/table already exists, the contents of the DataFrame will be appended to the existing data.
SaveMode.Overwrite	When saving a DataFrame to a data source, if data/table already exists, the existing data will be overwritten by the contents of the DataFrame.
SaveMode.Ignore	When saving a DataFrame to a data source, if data already exists, the save operation will not save the contents of the DataFrame and will not alter the existing data. This is similar to a `CREATE TABLE IF NOT EXISTS` in SQL.