

BIG DATA ANALYTICS

Part -1

BY:

DR.RASHMI L MALGHAN

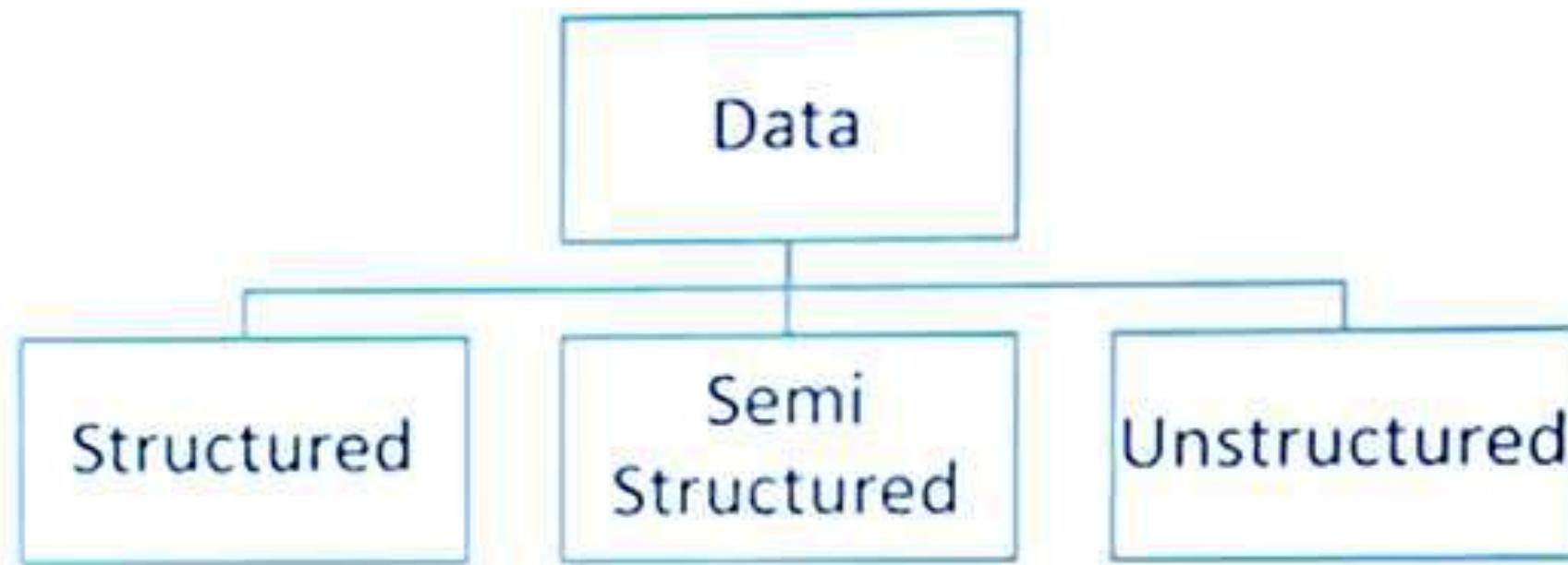
Contents

- ❖ Introduction to Big Data
- ❖ Types of Big Data
- ❖ Big Data characteristics
- ❖ Challenges
- ❖ Data Generators (Fields of Big Data)
- ❖ Traditional Vs Big Data approach
- ❖ Life Cycle
- ❖ Case Study.

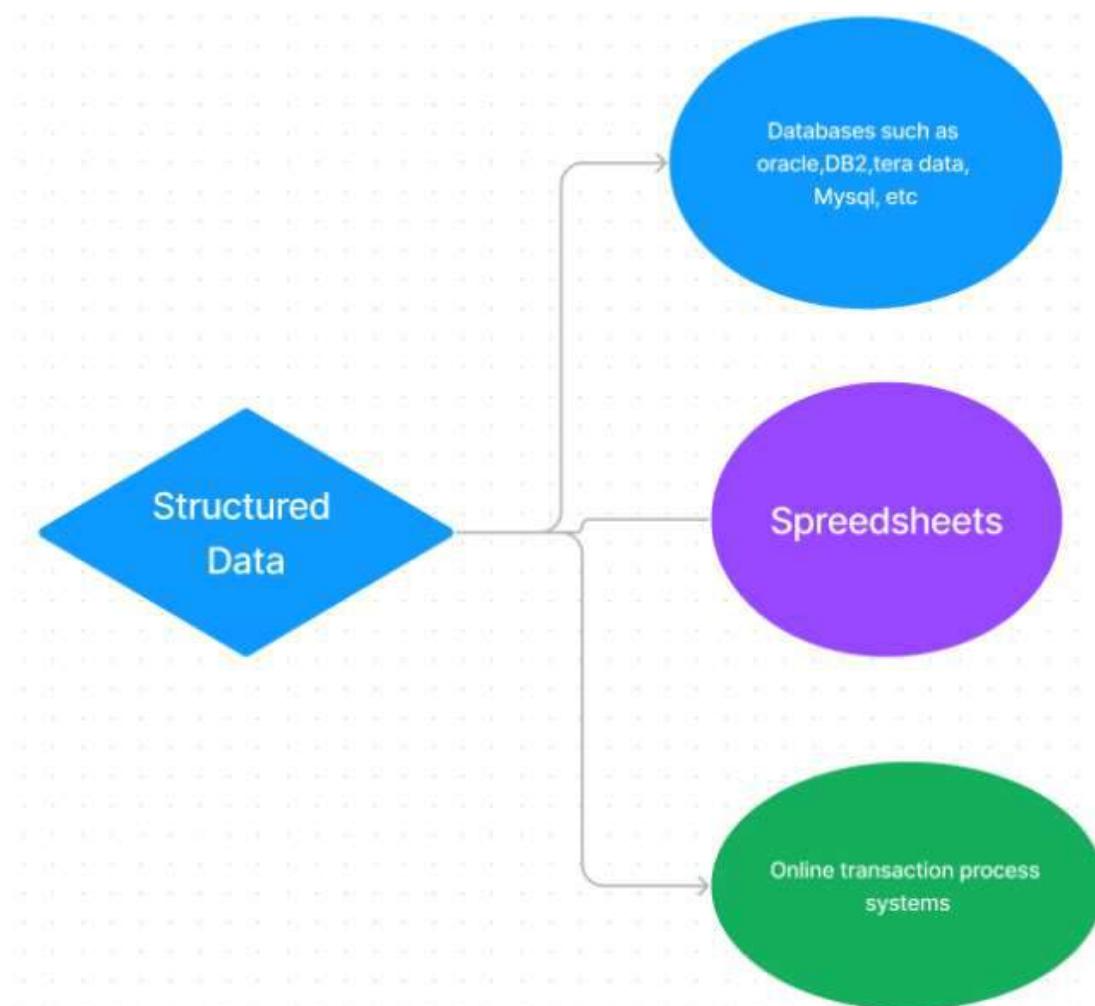
Small Data VS Big Data

SL.NO	Small Data	Big Data
1	Structured	Unstructured
2	Storage = MB, GB, TB	Storage = PB, EB, ZB, YB
3	Gradually Increases (Slow)	Exponentially/ Rapidly Increases
4	Locally Present	Globally Present
5	Centralized	Distributed
6	Oracle, SQL Server	Hadoop, Spark, Cassandra
7	Single Node	Multiple Node

TYPES OF DATA

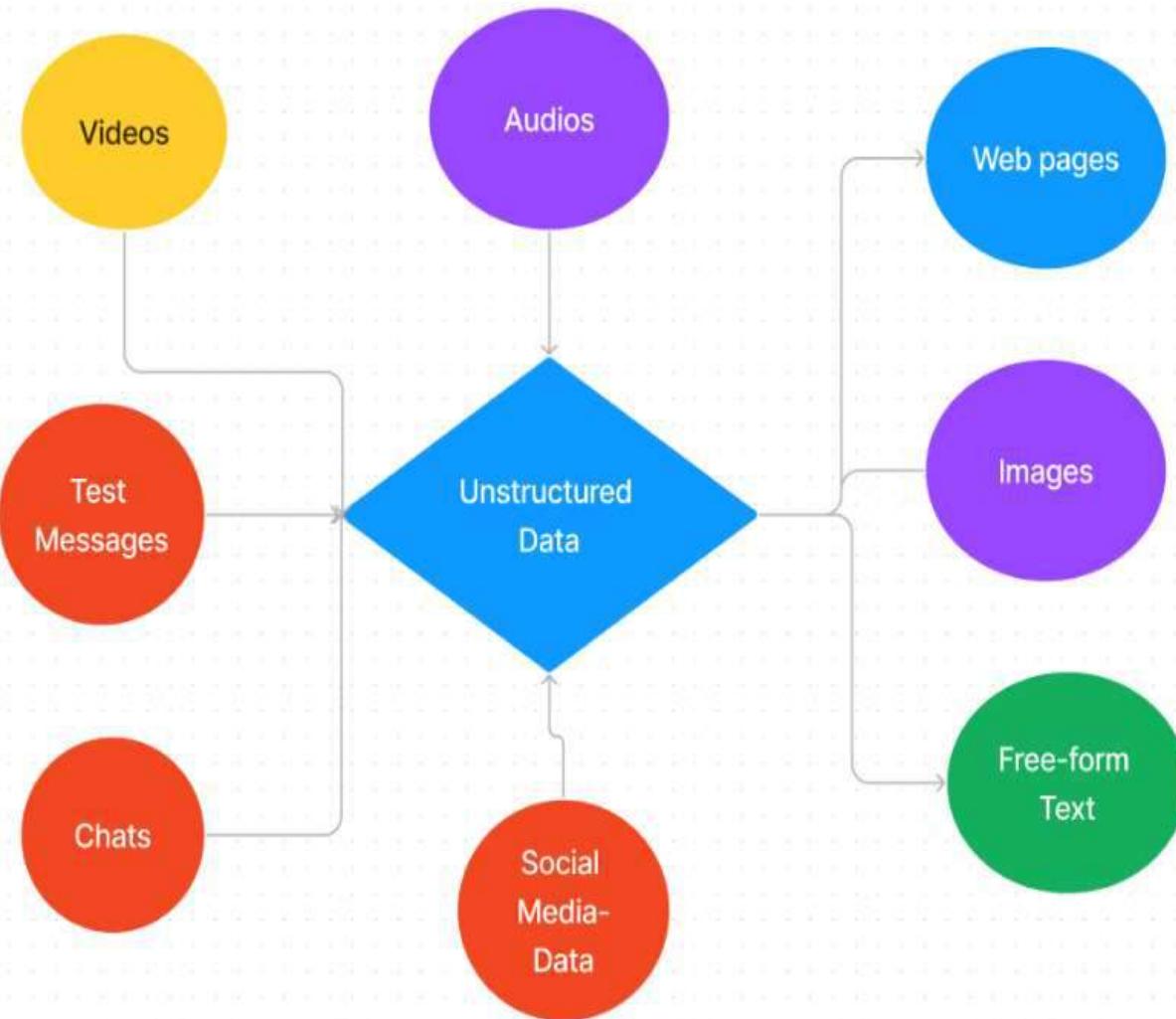


Structured



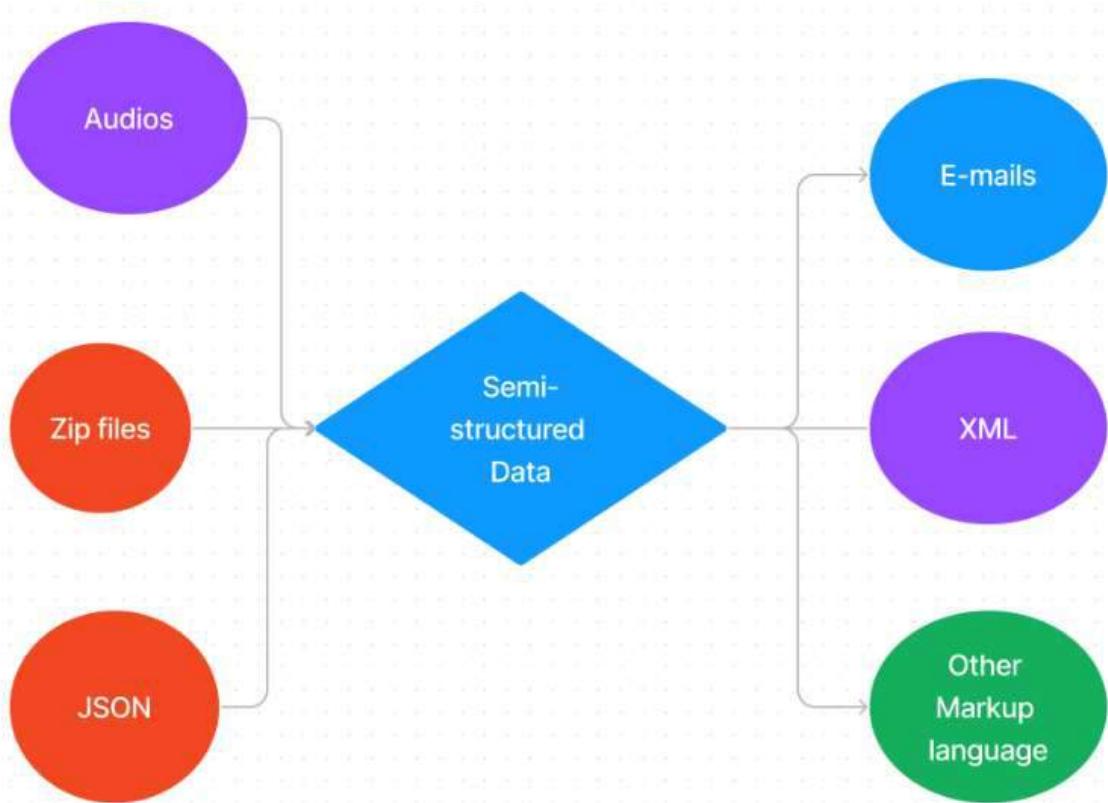
- Data that **resides in a fixed field** within a record.
- It is type of data most **familiar to our everyday lives**. Ex: birthday, address
 - A certain schema binds it, so all the data has the same set of properties. Structured data is also called relational data.
- It is split into multiple tables to **enhance the integrity of the data** by creating a single record to depict an entity.
- **Relationships** are enforced by the application of **table constraints**.

Unstructured



- Unstructured data is the kind of data that doesn't **adhere to any definite schema or set of rules**.
- Its arrangement is **unplanned and haphazard**.
- Photos, videos, text documents, and log files can be generally considered unstructured data.
- Even though the **metadata accompanying an image or a video may be semi-structured**, the actual **data being dealt with is unstructured**.
- Additionally, Unstructured data is also known as "**dark data**" because it cannot be **analyzed without the proper software tools**.

Semi- Structured



- Semi-structured data is not bound by **any rigid schema** for data storage and handling.
- The data is **not in the relational format** and is **not neatly organized** into rows and columns like that in a spreadsheet.
- However, there are some features **like key-value pairs** that help in discerning the different entities from each other.
- Since semi-structured data **doesn't need a structured query language**, it is commonly called *NoSQL data*.
- This type of information typically comes from **external sources** such as social media platforms or other web-based data feeds.

Data Generated On internet – Per minute



"2.1Million"



"3.8Million"



"1.0Million"



"4.5Million"



"188Million"

Examples of Big Data

- Big data is a **clustered management of different forms of data**
- Generated by **various devices** (Android, iOS, etc.),
- **Applications** (music apps, web apps, game apps, etc.),
- **Actions** (searching through SE, navigating through similar types of web pages, etc.).

Challenges of Big Data

- **Rapid Data Growth:** The growth velocity at such a high rate creates a problem to look for insights using it. There is no 100% efficient way to filter out relevant data.
- **Storage:** The generation of such a massive amount of data needs space for storage, and organizations face challenges to handle such extensive data without suitable tools and technologies.
- **Unreliable Data:** It cannot be guaranteed that the big data collected and analyzed are totally (100%) accurate. Redundant data, contradicting data, or incomplete data are challenges that remain within it.
- **Data Security:** Firms and organizations storing such massive data (of users) can be a target of cybercriminals, and there is a risk of data getting stolen. Hence, encrypting such colossal data is also a challenge for firms and organizations. © 2009 — 2023 W3schools® of Technology.

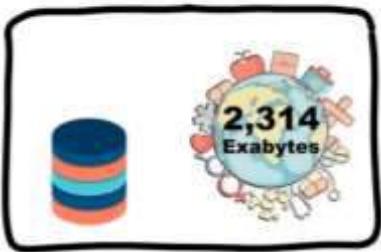
Fields of data that come under the umbrella of Big Data: (Generators of Data)

- **Black Box Data:** Black box data is a type of data that is collected from private and government helicopters, airplanes, and jets. This data includes the capture of Flight Crew Sounds, separate recording of the microphone as well as earphones, etc.
- **Stock Exchange Data:** Stock exchange data includes various data prepared about 'purchase' and 'selling' of different raw and well-made decisions.
- **Social Media Data:** This type of data contains information about social media activities that include posts submitted by millions of people worldwide.
- **Transport Data:** Transport data includes vehicle models, capacity, distance (from source to destination), and the availability of different vehicles.
- **Search Engine Data:** Retrieve a wide variety of unprocessed information that is stored in SE databases.

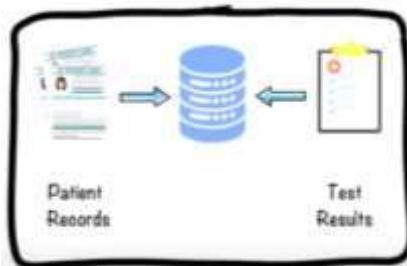
Classify any data as Big Data ?

- Based on 5V's.

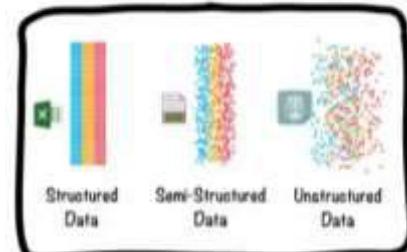
Volume



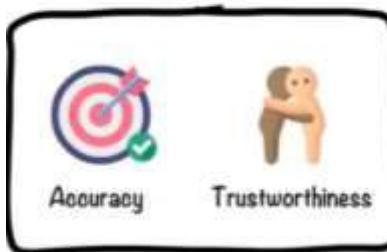
Velocity



Variety



Veracity

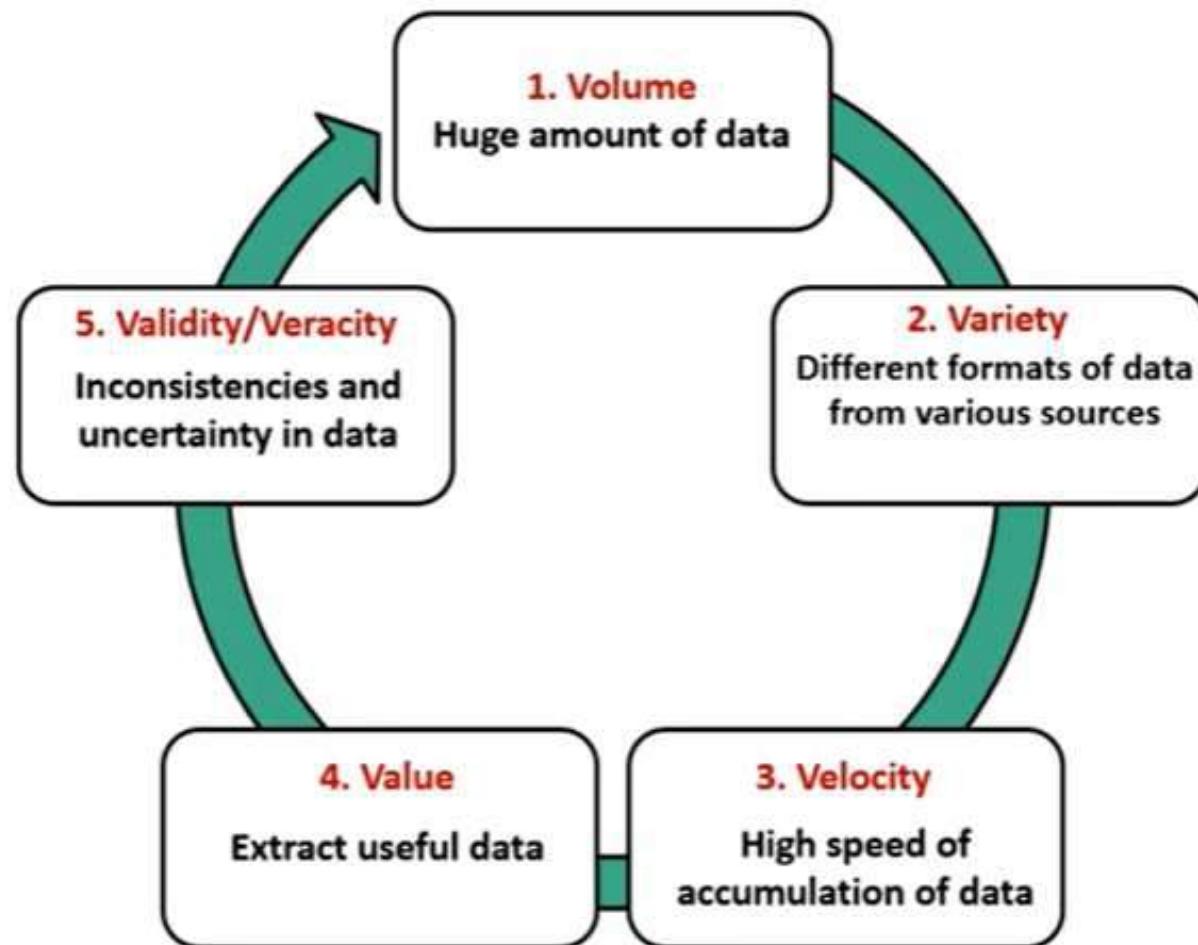


Value

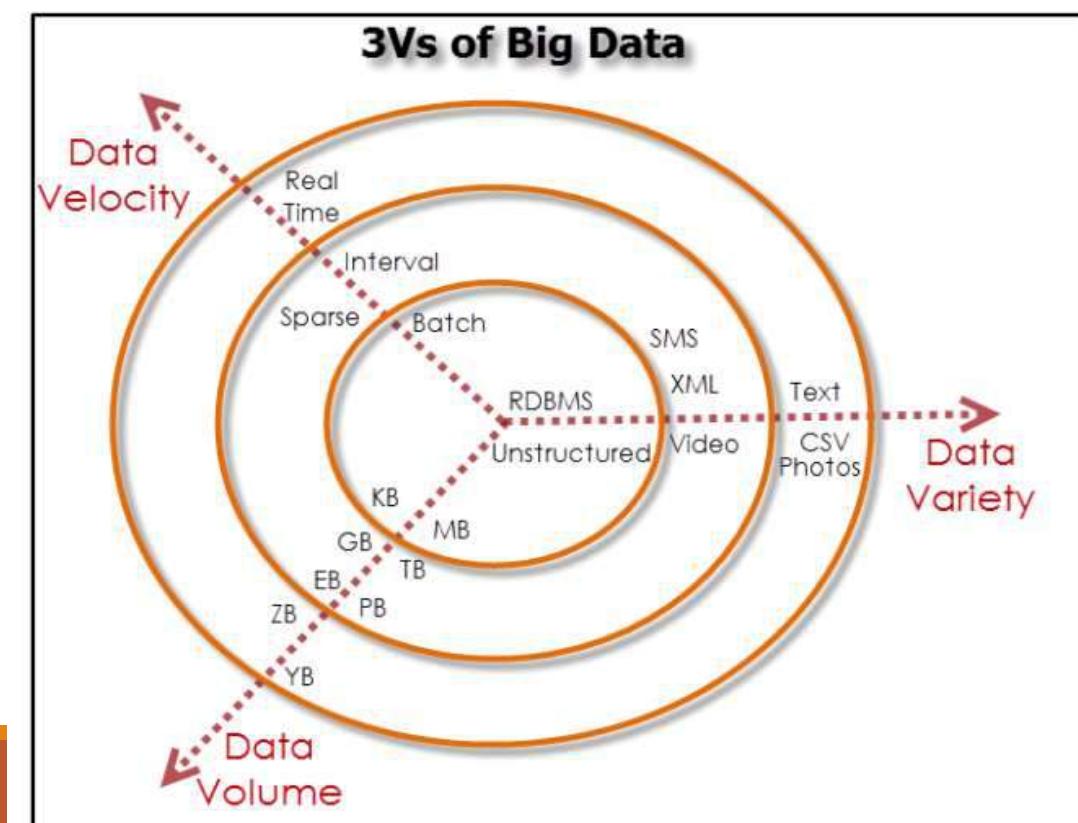


Big Data Characteristics

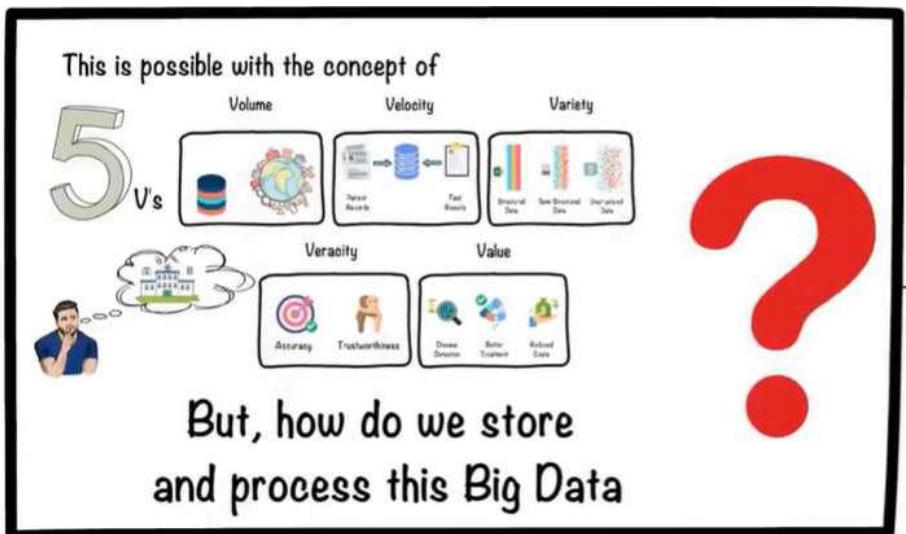
Five V's in Big Data



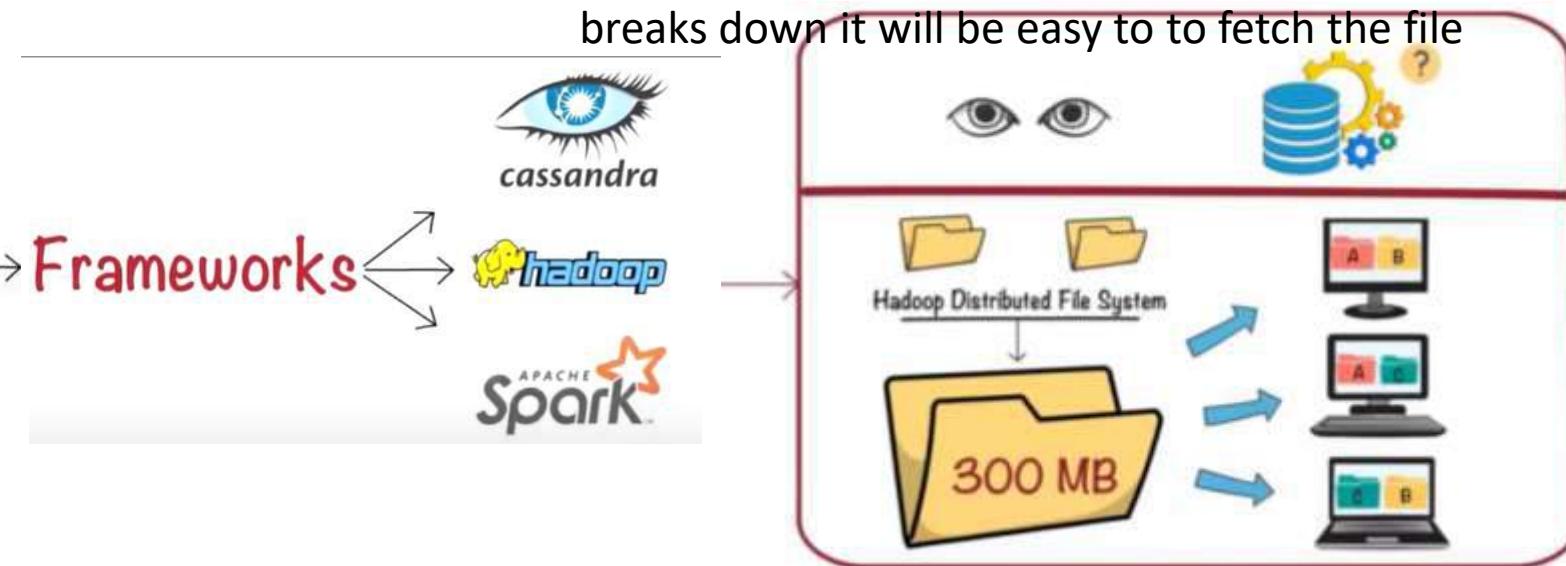
- Volume: Related to size of the data
- Variety: Comprises of a variety of data
- Velocity: Refers to the speed of generation of data.
- Veracity: Quality of data captured, which can vary greatly, affecting its accurate analysis



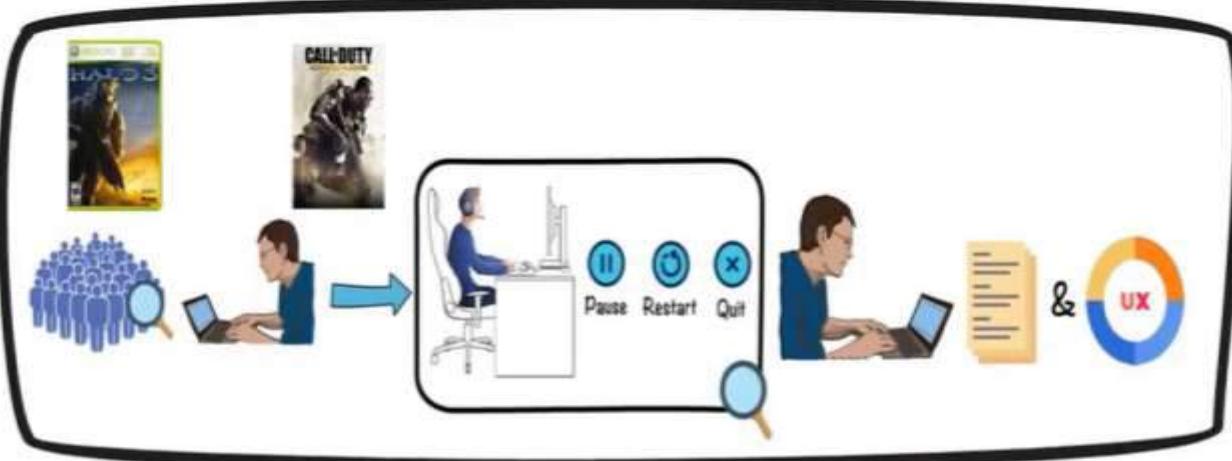
How we store and process this Big Data?



Store – Break file in to small sizes – 300 MB = 128Mb, 128 MB, 44MB
Store them with different nodes – when node breaks down it will be easy to fetch the file



Data Analysis – Gaming



- Designers Analyze gaming data at which stage customers pause , restart , quit etc.
- This analysis will help designers to work on enhancement of story line of games.
- Improve “user Experience”
- Reduce Churn rate.

Predict Hurricane's landfall



Big Data

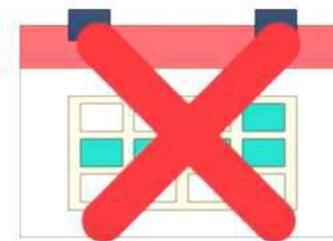


Hurricane Sandy
in 2012



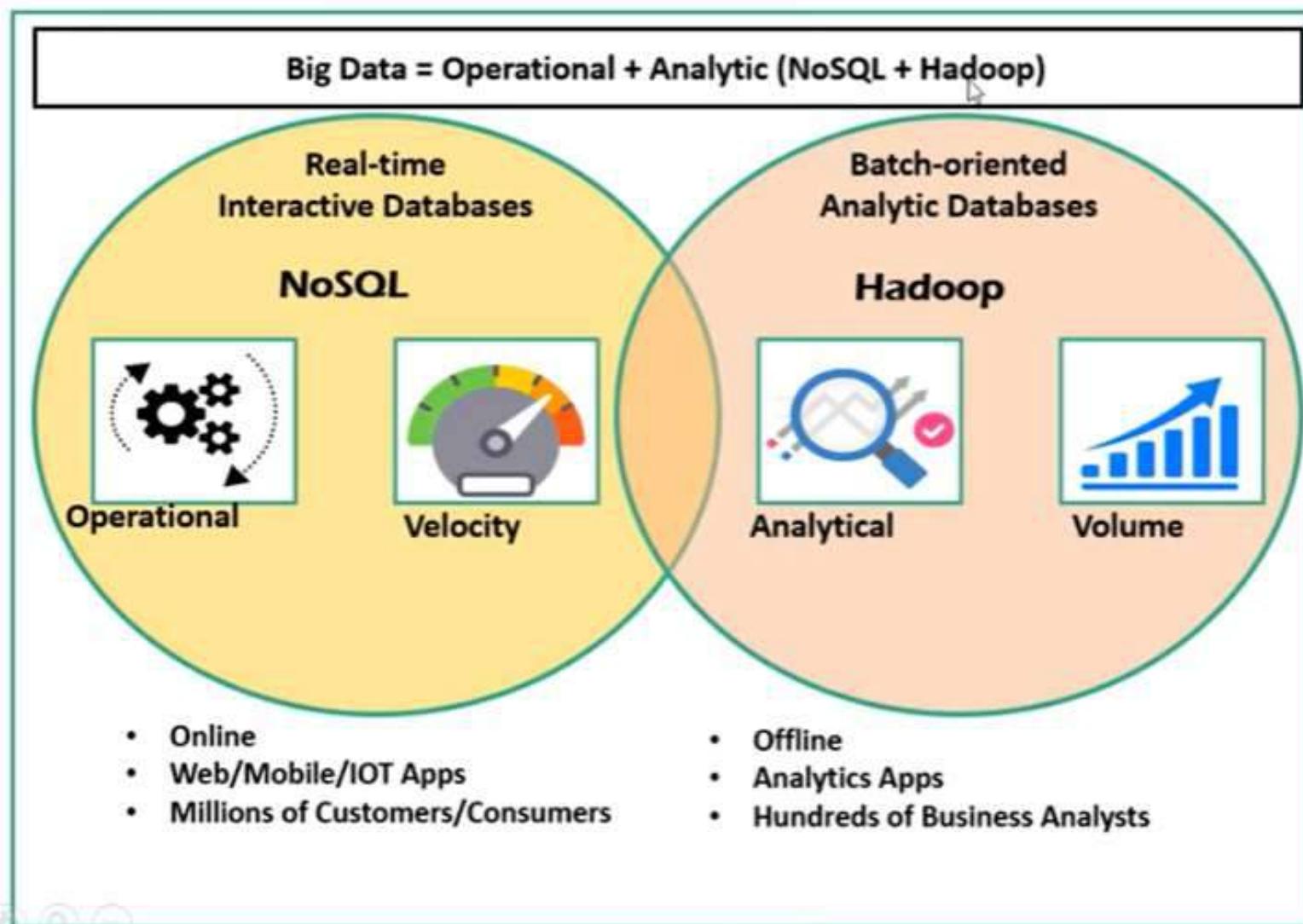
Necessary measures
were taken

It could predict the hurricane's landfall
five days in advance which wasn't possible earlier.



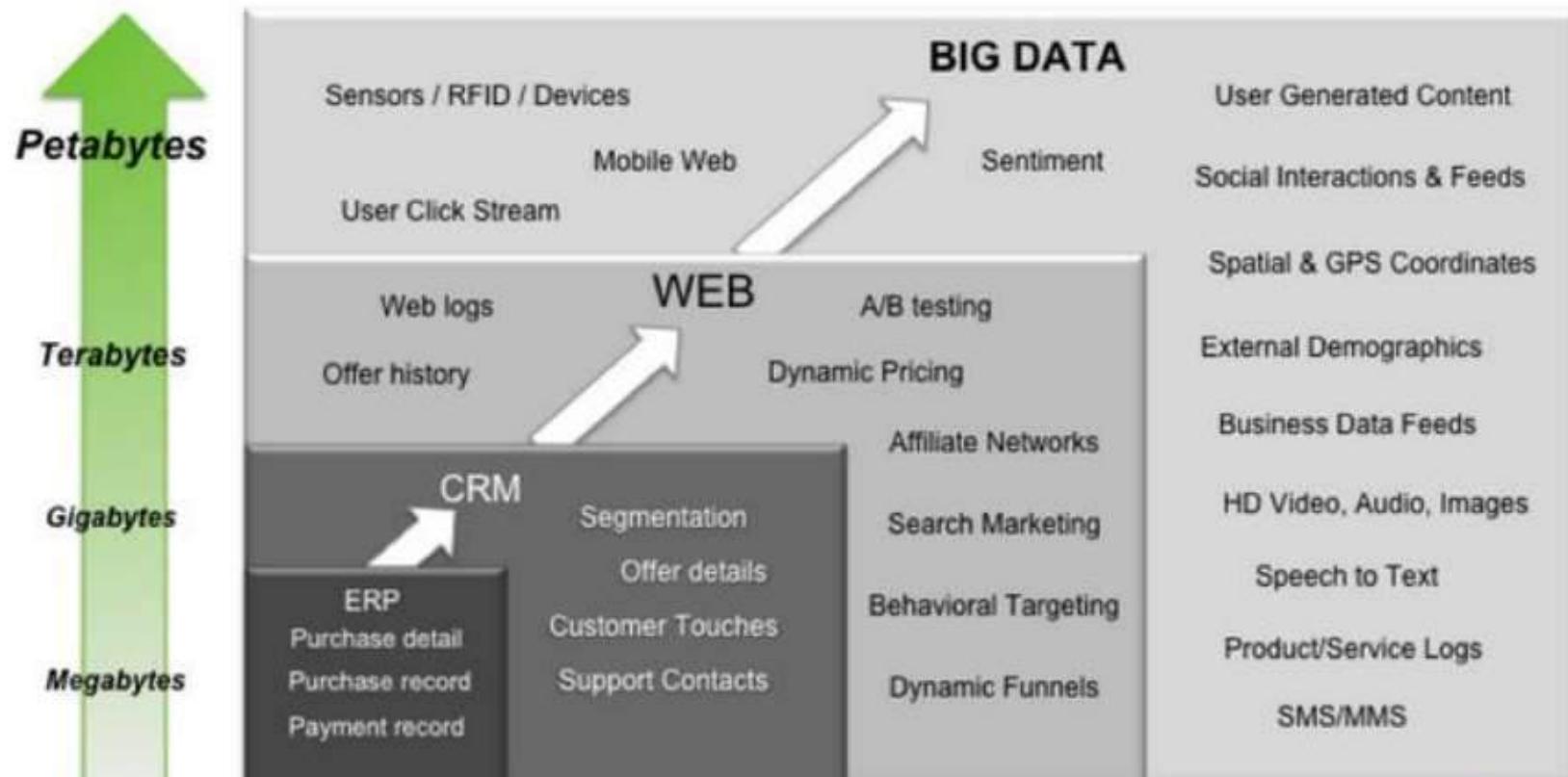
- Processed
- Analyzed Accurately

Big data Technologies/ Classes



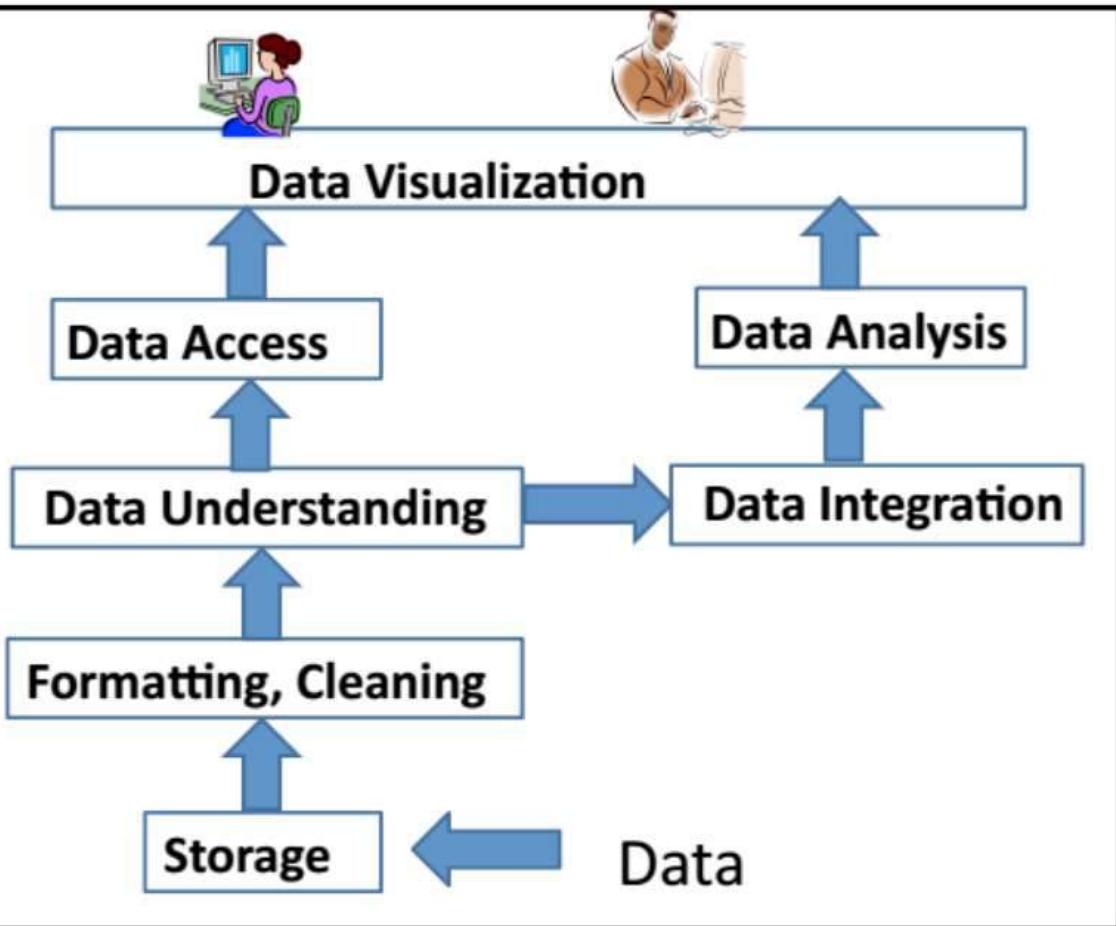
Big Data?

Big Data = Transactions + Interactions + Observations



Increasing Data Variety and Complexity

Big Data Layout



- Apache Hadoop- Apache
- MapReduce – Google
- HDFS(Hadoop distributed file system) – Apache
- Hive – Facebook
- Pig - Yahoo

Figure : Big Data layout

Apache Hadoop

- Apache Hadoop is one of the main supportive element in Big Data technologies.
- It simplifies the processing of large amount of structured or unstructured data in a cheap manner.
- Hadoop is an open source project from Apache that is continuously improving over the years.
- Hadoop is basically a set of software libraries and frameworks to manage and process big amount of data from a single server to thousands of machines.
- It provides an efficient and powerful error detection mechanism based on application layer rather than relying upon hardware."
- In December 2012 Apache releases Hadoop 1.0.0, more information and installation guide can be found at Apache Hadoop Documentation.
- Hadoop is not a single project but includes a number of other technologies in it.

MapReduce

- MapReduce was introduced by google to create large amount of web search indexes.
- It is basically a framework to write applications that processes a large amount of structured or unstructured data over the web.
- MapReduce takes the query and breaks it into parts to run it on multiple nodes.
 - By distributed query processing it makes it easy to maintain large amount of data by dividing the data into several different machines.
- Hadoop MapReduce is a software framework for easily writing applications to manage large amount of data sets with a highly fault tolerant manner.
- More tutorials and getting started guide can be found at Apache Documentation.

HDFS(Hadoop distributed file system)

- HDFS is a **java based file system** that is used to store structured or unstructured data over **large clusters of distributed servers**.
- The data stored in HDFS has **no restriction or rule to be applied**, the data can be either fully unstructured or purely structured.
- In HDFS the work to **make data senseful is done by developer's code only**.
- Hadoop distributed file system provides a highly fault tolerant atmosphere with a deployment on **low cost hardware machines**.
- HDFS is now a part of **Apache Hadoop** project, more information and installation guide can be found at Apache HDFS documentation

HIVE

- Hive was originally developed by Facebook, now it is made open source for some time.
- Hive works something like a bridge in between SQL and Hadoop, it is basically used to make SQL queries on Hadoop clusters.
- Apache Hive is basically a data warehouse that provides ad-hoc queries, data summarization and analysis of huge data sets stored in Hadoop compatible file systems.
- Hive provides a SQL like called HiveQL query based implementation of huge amount of data stored in Hadoop clusters.
- In January 2013 apache releases Hive 0.10.0, more information and installation guide can be found at Apache Hive Documentation.

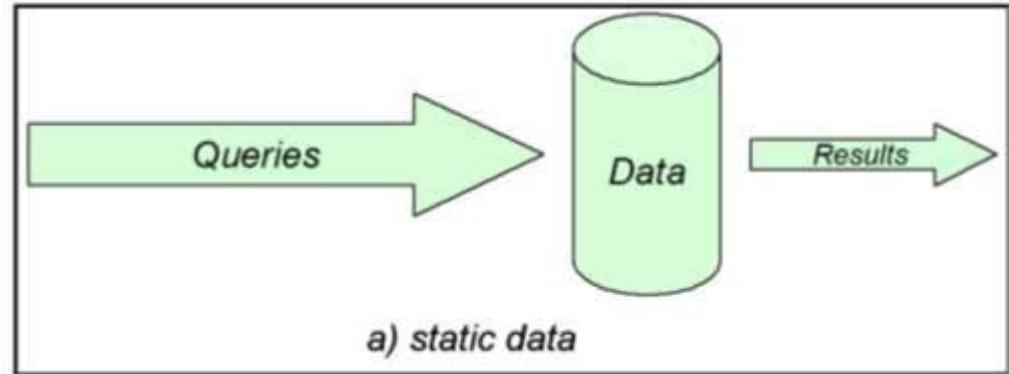
Pig

- Pig was introduced by yahoo and later on it was made fully open source.
- It also provides a bridge to query data over Hadoop clusters but unlike hive, it implements a script implementation to make Hadoop data accessible by developers and business persons.
- Apache pig provides a high level programming platform for developers to process and analyses Big Data using user defined functions and programming efforts.
- In January 2013 Apache released Pig 0.10.1 which is defined for use with Hadoop 0.10.1 or later releases. More information and installation guide can be found at Apache Pig Getting Started Documentation.

Traditional vs Big Data

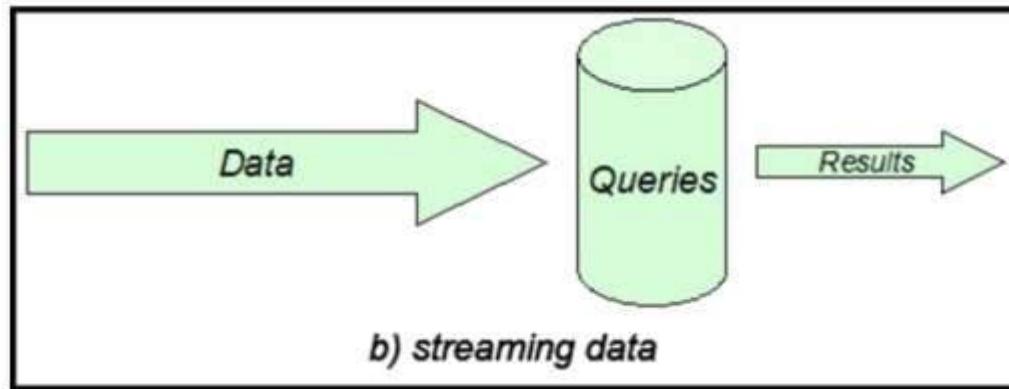
Schema Hard Code/SQL/Small Data

- online transactions and quick updates.
- Schema Based DB (hard code schema attachment)
- Structured
- Uses SQL for Data processing
- Maintains relationship between elements



Schema Less /NoSQL/Big Data

- Migration Easy
- Schema Less Based DB
- store unstructured, semi structured or even fully structured data
- Store a huge amount of data and not to maintain relationship between elements.





Business determines what questions to ask

Classic BI

Structured & Repeatable Analysis



IT structures the data to answer those questions

"Capture only what's needed"

- Working on the live coming data, which can be an input from the ever-changing scenario cannot be dealt in the traditional approach.



IT delivers a platform for storing, refining, and analyzing **all data sources**

"Capture only what's needed"

Big Data Analytics

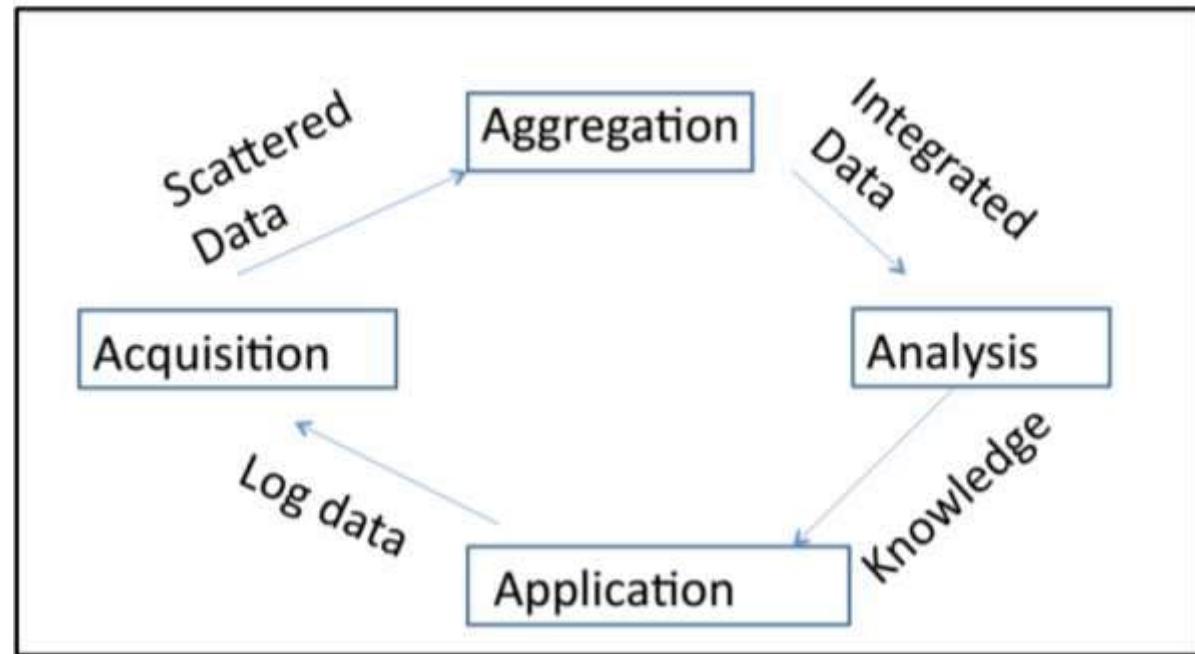
Multi-structured & Iterative Analysis



Business explores data for questions worth answering

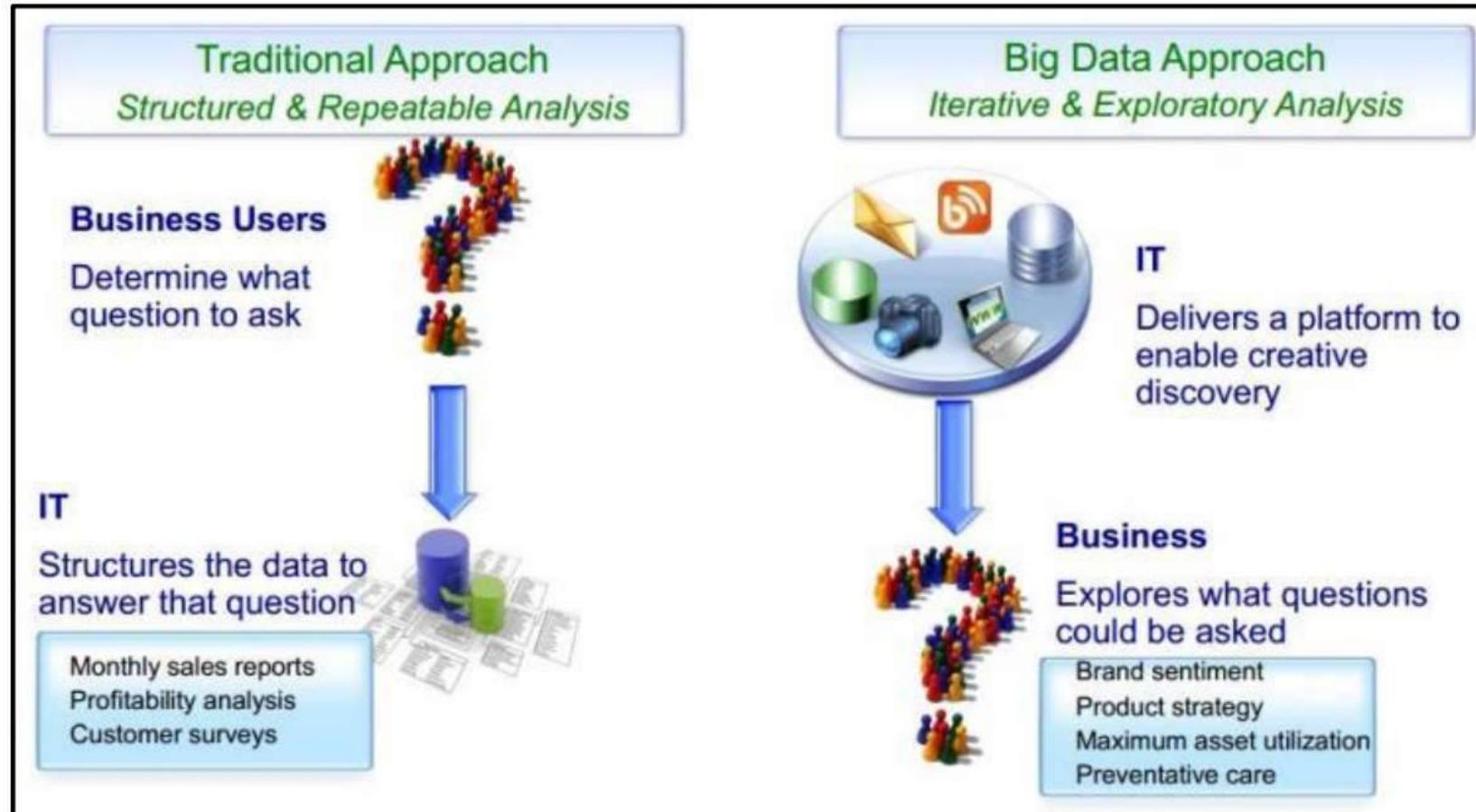
- The live flow of data is captured and the analysis is done on it.
- Efficiency increases when the data to be analyzed is large

Life Cycle of Data

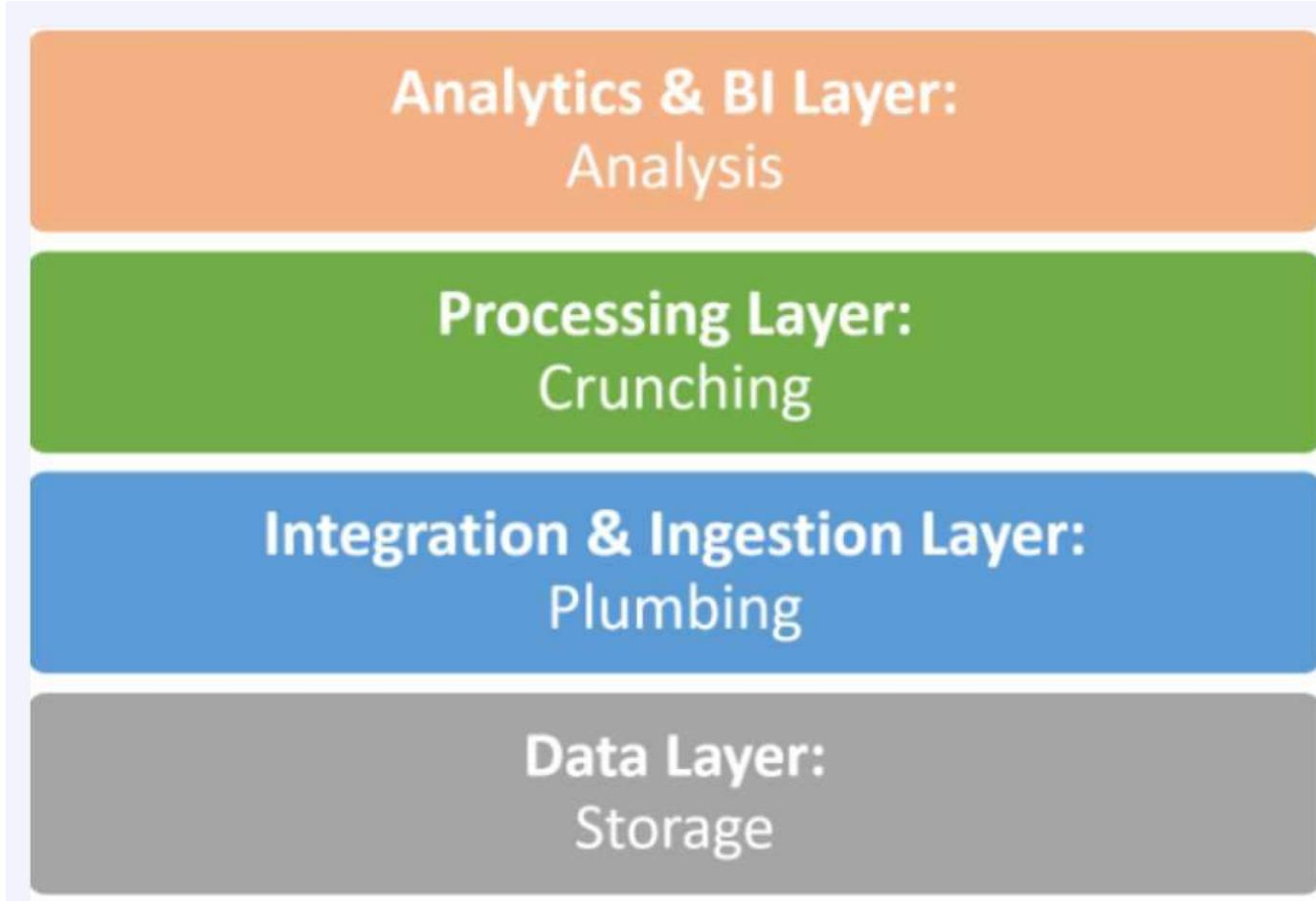


- 1) The analysis of data is done from the knowledge experts and the expertise is applied for the development of an application.
- 2) The streaming of data after the analysis and the application, the data log is created for the acquisition of data.
- 3) The data is mapped and clustered together on the data log.
- 4) The clustered data from the data acquisition is then aggregated by applying various aggregation algorithms.
- 5) The integrated data again goes for an analysis.
- 6) The complete steps are repeated till the desired, and expected output is produced

Traditional vs Big data Approaches



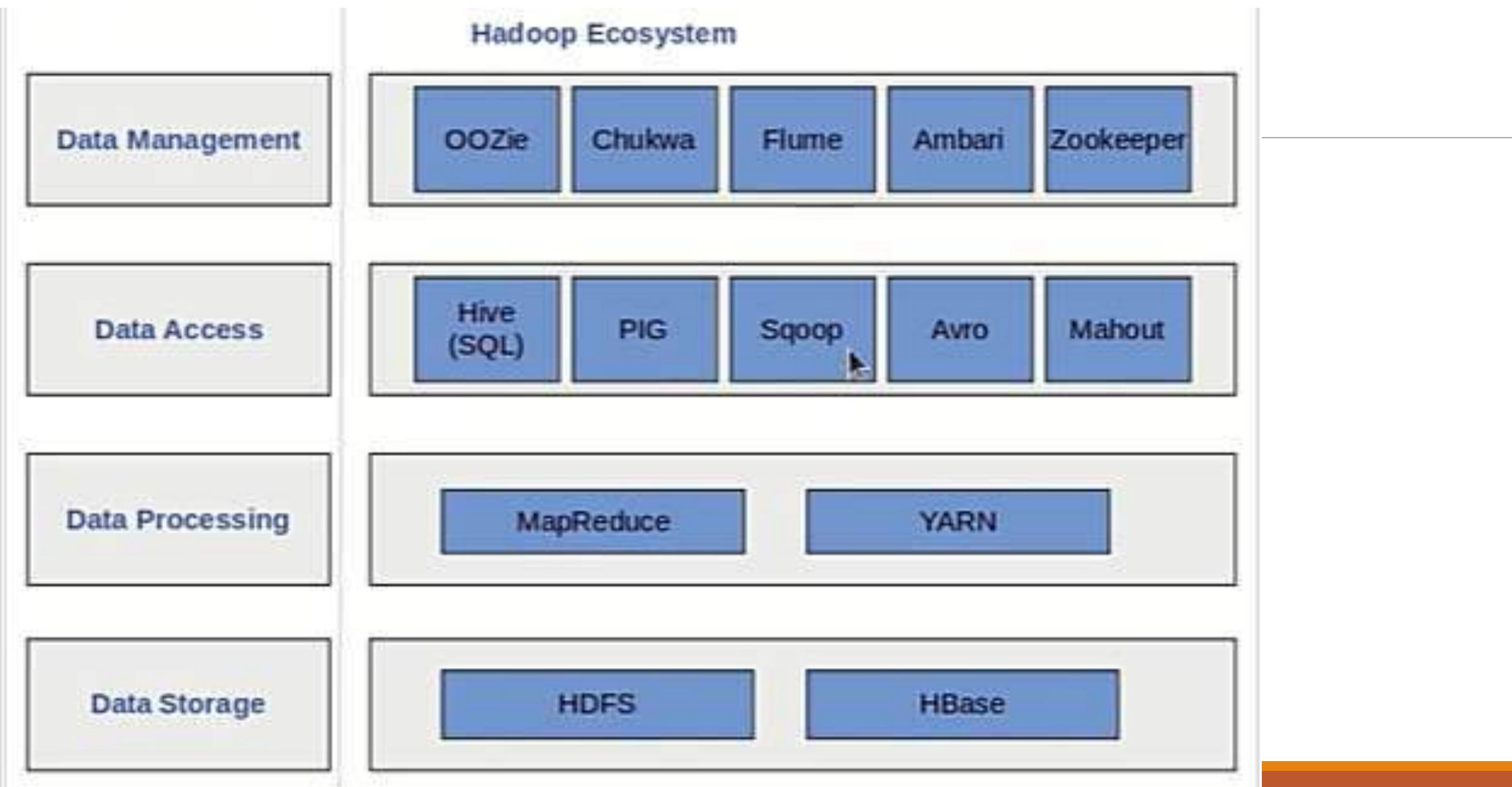
Big Data Technology Stack



Design of logical layers in a data processing

Layer 5 Data consumption	Export of datasets to cloud, web etc.	Datasets usages: BPs, BIs, knowledge discovery	Analytics (real-time, near real-time, scheduled batches), reporting, visualization
Layer 4 Data processing	Processing technology: MapReduce, Hive, Pig, Spark	Processing in real-time, scheduled batches or hybrid	Synchronous or asynchronous processing
Layer 3 Data storage	Considerations of types (historical or incremental), formats, compression, frequency of incoming data, patterns of querying and data consumption	Hadoop distributed file system (scaling, self-managing and self-healing), Spark, Mesos or S3	NoSQL data stores – Hbase, MongoDB, Cassandra, Graph database
Layer 2 Data ingestion and acquisition	Ingestion using Extract Load and Transform (ELT)	Data semantics (such as replace, append, aggregate, compact, fuse)	Pre-processing (validation, transformation or transcoding) requirement
Layer 1 Identification of internal and external sources of data	Sources for ingestion of data	Push or pull of data from the sources for ingestion	Data types for database, files, web or service
			Data formats: structured, semi- or unstructured for ingestion

Hadoop Ecosystem : Components Classification



Components Classification

Mainly components are classified into 4 categories:

- Data Storage

HDFS: Hadoop Distributed File System

HBase: A columnar database that uses HDFS for its storage

- Data Processing

YARN: Operating system/scheduler

MapReduce: Data processing programming model

Components Classification

- Data Access

Hive(SQL): A distributed data warehouse for HDFS data that provides a SQL-like layer to this data

Pig: Framework for analyzing large data sets that let you create data pipelines

Mahout: Machine learning algorithm libraries

Sqoop: Data movement tool that moves data b.w HDFS and relational db

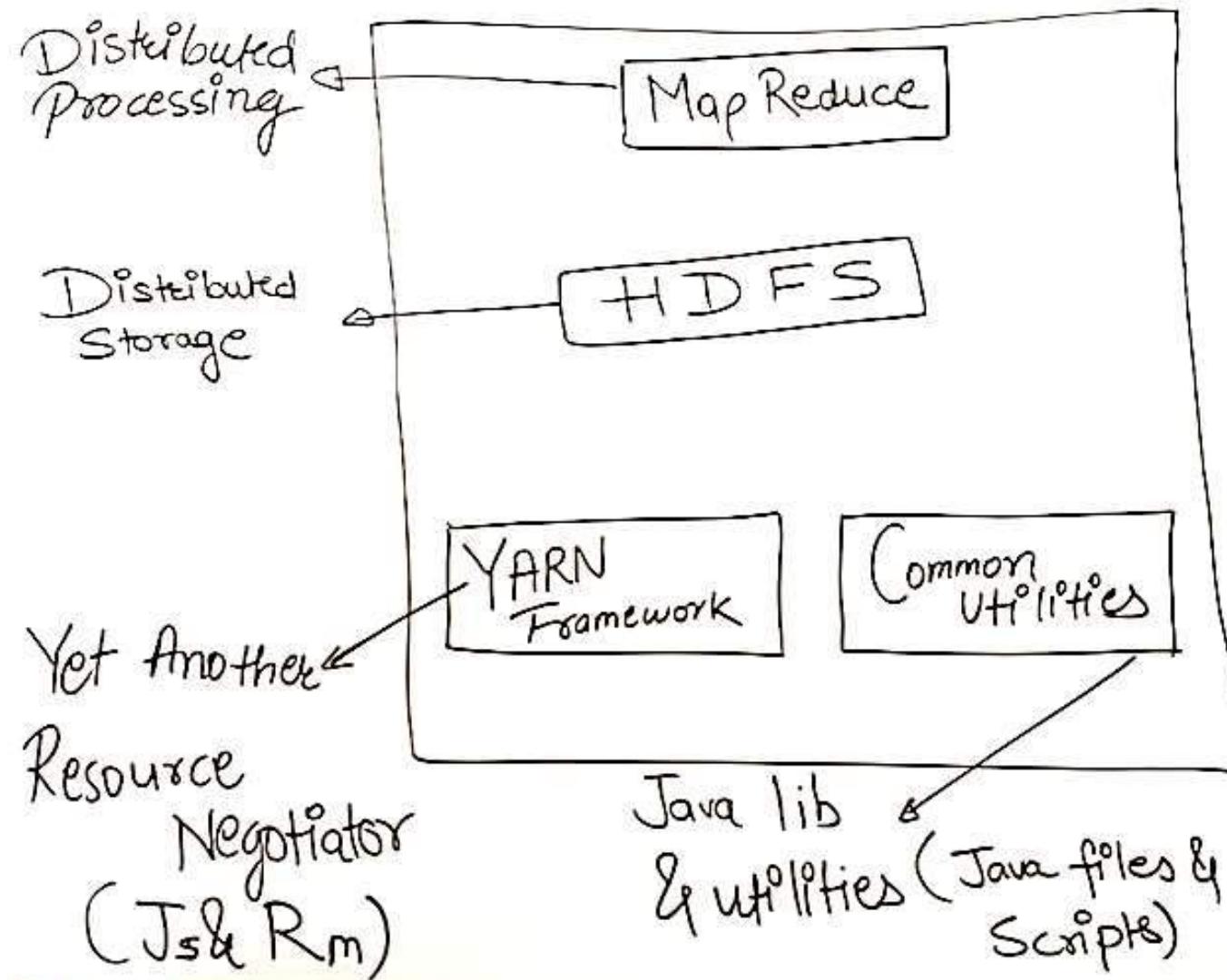
Avro : Framework for transforming data into a compact binary form

- Data Management

Zookeeper: Managing clusters/coordination service

Oozie: Job scheduling tool

HADOOP ARCHITECTURE:



- **Common Utilities (Hadoop Common)** – Provide all the utilities, Java Files, Scripts required by other modules (YARN, HDFS, MapReduce)
- **YARN**: Yet Another Resource Negotiator – Job Scheduling & Resource Management
- **HDFS**: Storage (Master & Slave Nodes)
- **MapReduce**: Data Processing - Parallelism



Hadoop Ecosystem


oozie
(Work flow)


HCatalog
Table & schema Management

 
Pig
(Scripting) **Hive**
(Sql Query)

 
Mahout
(Machine Learning) **Drill**
(Interactive Analysis)


Thrift
AVRO (Cross Language Service)

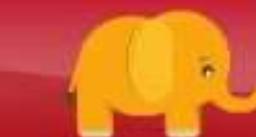

APACHE HBASE
HBASE (Columnar Store)


Sqoop
(Data Collection)


Zookeeper
(Coordination)


Ambari
Apache Ambari
(Management & Monitoring)


Mapreduce
(Data Processing)




Yarn
(Cluster Resource Management)




HDFS
(Hadoop Distributed File system)




FLUME
Flume
(Data Collection)

Various Data Storage and Usage, Tools

Data Source	Examples of Usages	Example of Tools
Relational databases	Managing business applications involving structured data	Microsoft Access, Oracle, IBM DB2, SQL Server, MySQL, PostgreSQL Composite, SQL on Hadoop [HPE (Hewlett Packard Enterprise) Vertica, IBM BigSQL, Microsoft Polybase, Oracle Big Data SQL]
Analysis databases (MPP, columnar, In-memory)	High performance queries and analytics	Sybase IQ, Kognitio, Terradata, Netezza, Vertica, ParAccel, ParStream, Infobright, Vectorwise,
NoSQL databases (Key-value pairs, Columnar format, documents,	Key-value pairs, fast read/write using collections of name-value pairs for storing any type of data; Columnar format, documents,	Key-value pair databases: Riak DS (Data Store), OrientDB, Column format databases (HBase, Cassandra), Document oriented databases: CouchDB, MongoDB; Graph

Various Data Storage and Usage, Tools

Objects, graph)	objects, graph DBs and DSs	databases (Neo4j, Tetan)
Hadoop clusters	Ability to process large data sets across a distributed computing environment	Cloudera, Apache HDFS
Web applications	Access to data generated from web applications	Google Analytics, Twitter
Cloud data	Elastic scalable outsourced databases, and data administration services	Amazon Web Services, Rackspace, GoogleSQL
Individual data	Individual productivity	MS Excel, CSV, TLV, JSON, MIME type
Multidimensional	Well-defined bounded exploration especially popular for financial applications	Microsoft SQL Server Analysis Services
Social media data	Text data, images, videos	Twitter, LinkedIn

Hadoop 1 vs Hadoop 2 [MRV1 vs MRV2]

1) Components

Hadoop1

- HDFS
- MapReduce

Hadoop2

- HDFS
- YARN / MapReduce

2) Services/ Daemons

- Namenode
- Datanode
- Job tracker
- Task tracker

- Namenode/Secondary Namenode
- Datanode
- Resource manager
- Node manager

3) Working

- HDFS: Data storage
- MapReduce:
Data processing +
Resource management

- HDFS: Data storage
- YARN: Resource management
- MapReduce: Data processing

Hadoop 1 vs Hadoop 2 [MRV1 vs MRV2]

4) Limitation

- Single master multiple slaves
- Multiple masters and slaves

5) Cluster capability

- Around 5000 nodes
- Around 10000 nodes

6) Ecosystem

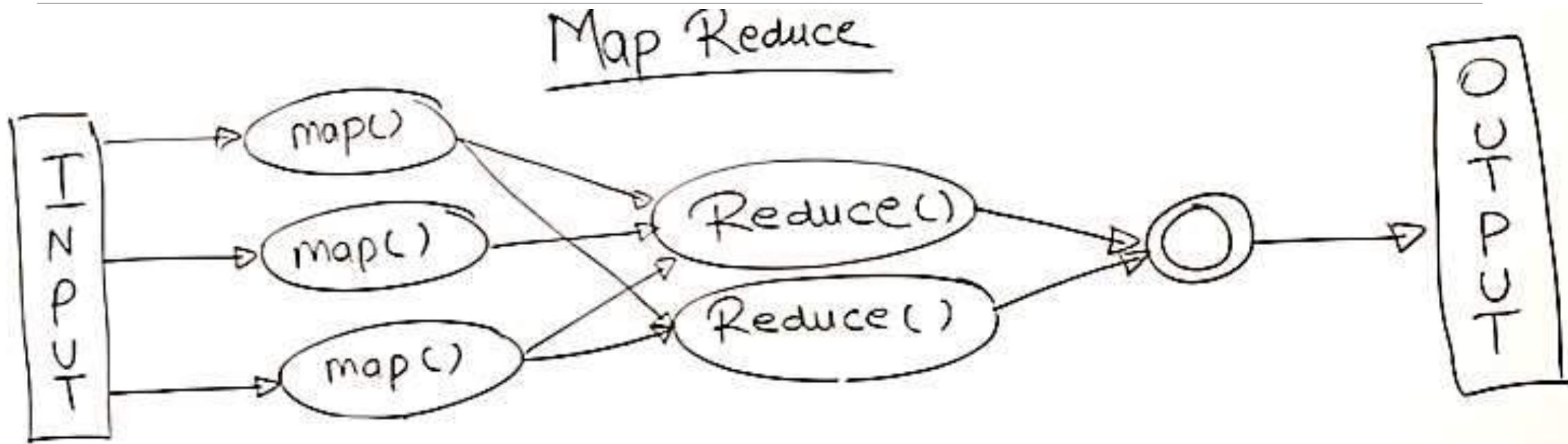
- Less services and tools
- More services and tools

I

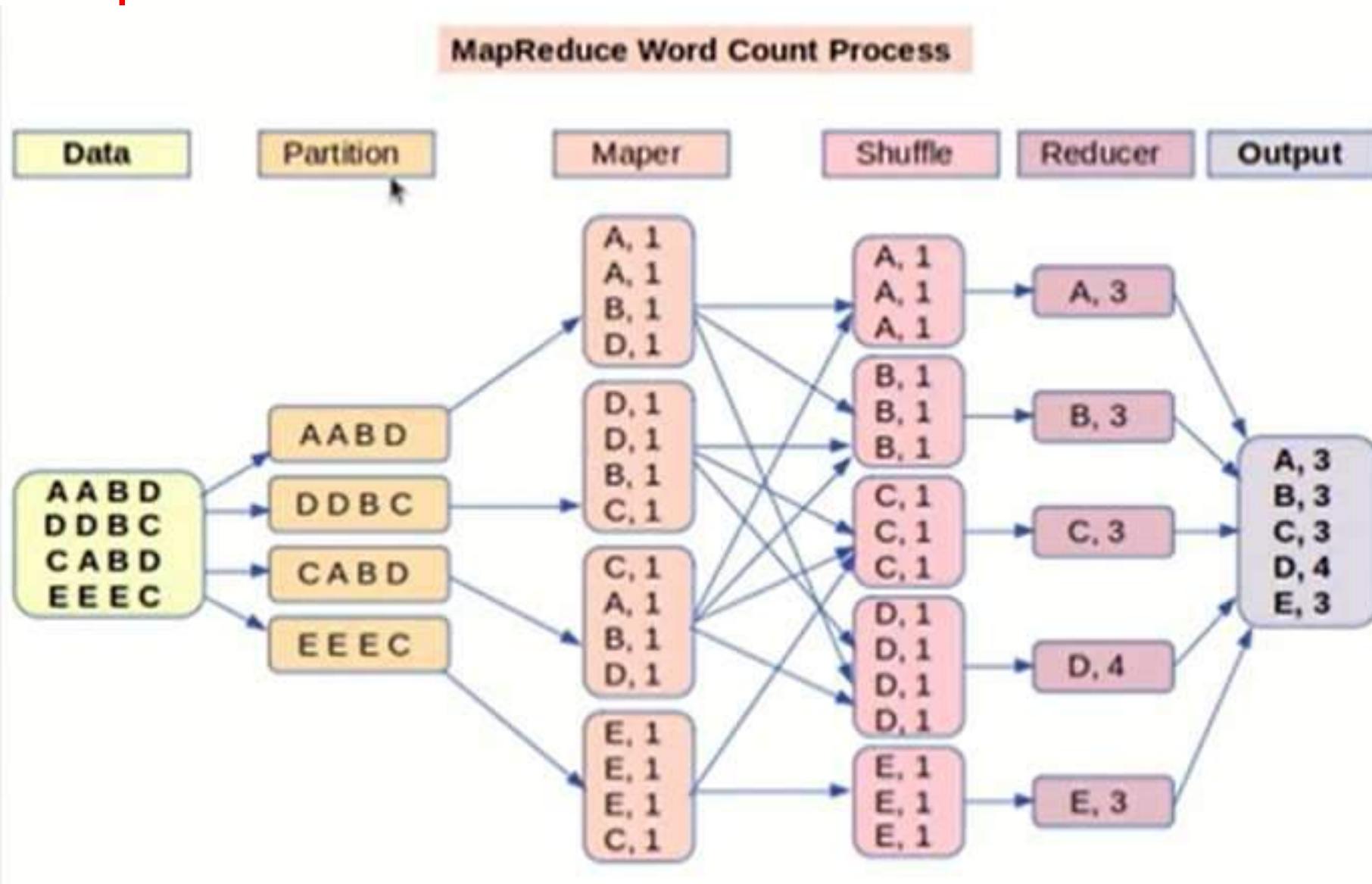
Hadoop V1 vs Hadoop V2



MapReduce:



MapReduce



MapReduce : Case Study (Paper Correction and Identifying Topper)

- Map : Parallelism , Reduce : Grouping

- Total = 20 Papers

- Time = 1 min/Paper

Without (MPP):

- 20 Papers Corrections = 20 mins

With (MPP/MapReduce)

- Divide Task – 4 groups / 4 districts (D1, D2, D3, D4) [Divide Tasks – Mapper Class]
 - Assign 5 papers- Each district
 - To fetch “District Topper” = (D1 = T1, D2= T2, D3 = T3, D4 = T4) = 4 Toppers ; 5 minutes
 - To fetch “State Topper” = T1, T2, T3, T4 = Topper; 4 minutes [Aggregation: Reducer Class]
 - Total job “To Fetch Topper” = 9 Minutes
-
- Summary/ Technicality of MapReduce : Time is reduced rapidly in case of MPP application.



Hadoop Technology

- Hadoop is Open Source tool from the Apache Software Foundation. As the open source project, we can even change the source codes of the Hadoop system. Most of the Hadoop codes are written by Yahoo, IBM, Cloudera etc.
- Hadoop provides parallel processing through different commodity hardware simultaneously.
- As it works on Commodity hardware so the cost is very low. Commodity hardware is low-end and very cheap hardware. So the Hadoop Solution is also economic.

Why TO Use Hadoop?

- The Hadoop solution is very popular. It has captured at least 90% of Big data market.
- Hadoop has some unique features that make this solution very popular.
- Hadoop is Scalable. So we can increase the number of commodity hardware easily.
- It is a fault tolerant solution. When one node goes down other nodes can process the data.
- Data can be stored as a Structured, Unstructured and semi-structured mode. So it is more flexible.

A close-up photograph of a pencil lying diagonally across a sheet of graph paper. The graph paper features a grid pattern and a line chart with data points labeled '100' and '50'. The background is slightly blurred.

Introduction To Hadoop

BY:

DR. RASHMI L MALGHAN

Common Types of Architecture- Multiprocessor

- 1) Shared Memory (SM) : Common Central Memory – Shared by multiple processors
- 2) Shared Disk (SD) : Multiple Processors - Common Collection of Disks – Own Private Memory
- 3) Shared Nothing (SN) : Neither memory nor Disk – Shared among multiple processors.

Parallel Computing vs Distributed Computing

S.NO	Parallel Computing	Distributed Computing
1.	Many operations are performed simultaneously	System components are located at different locations
2.	Single computer is required	Uses multiple computers
3.	Multiple processors perform multiple operations	Multiple computers perform multiple operations
4.	It may have shared or distributed memory	It have only distributed memory
5.	Processors communicate with each other through bus	Computer communicate with each other through message passing.
6.	Improves the system performance	Improves system scalability, fault tolerance and resource sharing capabilities

Introduction:

1. Hadoop cluster

Cluster is collection of machines that uses the Hadoop software on the foundation of a distributed file system HDFS and a cluster resource manager YARN.

The nodes in Hadoop cluster are classified into two types:

i) Master nodes

These nodes run the services that coordinate the cluster's work. Clients contact the master nodes in order to perform computations.

In each cluster master nodes ranging from 3 to 6.

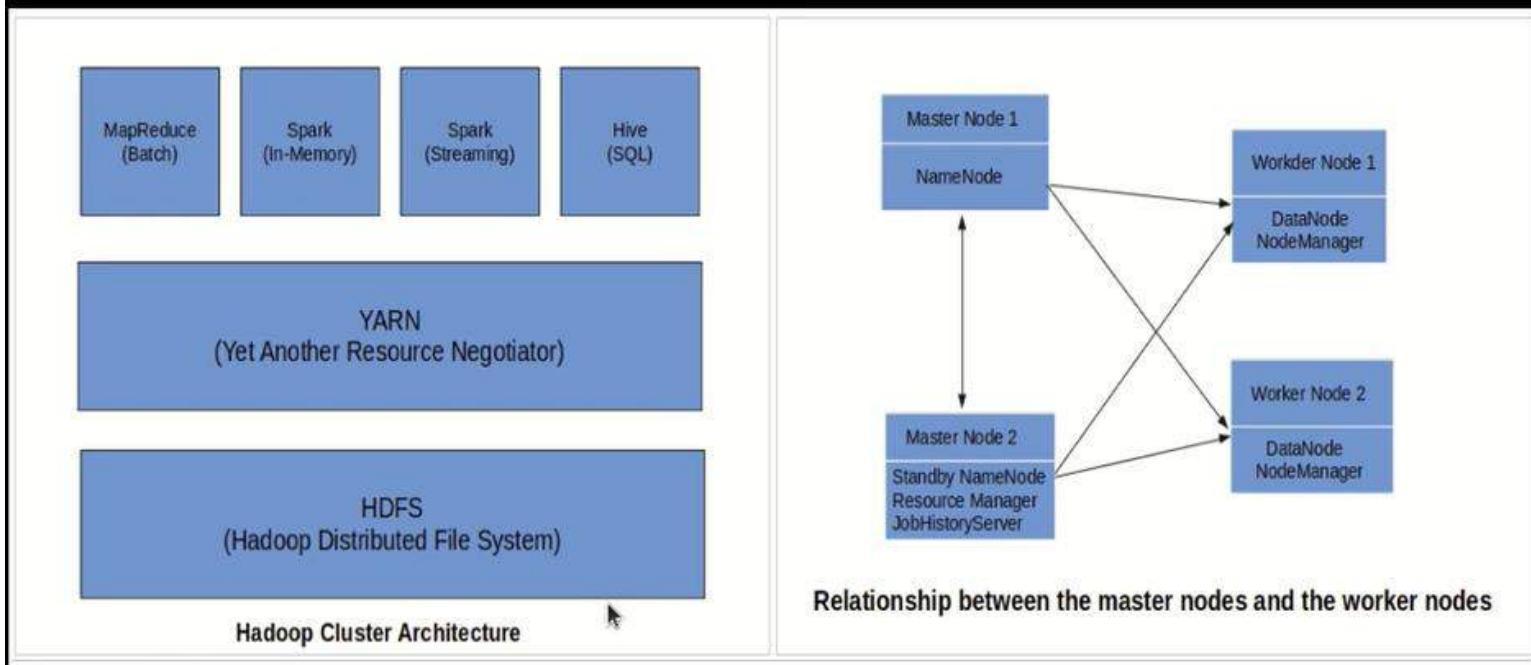
ii) Worker nodes

The nodes perform under the direction of process running on the master nodes. Most of cluster nodes are worker nodes. The worker nodes are where the data is stored and computations are performed.

Introduction:

- **HDFS (Hadoop Distributed File System)**: This is a distributed file system that stores data in blocks across the slave nodes. The master node runs a service called **NameNode**, which manages the file system namespace, the metadata of the files and directories, and the mapping of blocks to slave nodes. The slave nodes run a service called **DataNode**, which stores the actual data blocks and serves read and write requests from the clients.
- **YARN (Yet Another Resource Negotiator)**: This is a framework for resource management and job scheduling in Hadoop. The master node runs a service called **ResourceManager**, which allocates resources to different applications and monitors their progress. The slave nodes run a service called **NodeManager**, which launches and monitors the tasks assigned by the ResourceManager.
- **MapReduce**: This is a programming model for parallel processing of large data sets. The master node runs a service called **JobTracker**, which splits the input data into smaller chunks and assigns them to the slave nodes. The slave nodes run a service called **TaskTracker**, which executes the map and reduce tasks on the data chunks and reports the results back to the JobTracker.

Hadoop Architecture:



- Master Nodes: These are main servers .
 - 3 – 6 maximum created in hadoop cluster.
- Coordinate the work of slave nodes.
- Services: Name node service on master node, DataNode:
Running service on slave node
 - Datanode: Service of HDFS
 - NodeManager: Service of YARN

- Install hadoop: Default: 2 components will be installed (HDFS & YARN)
- Additional programs will be installed in hadoop env, like MapReduce, Spark, Hive etc.
- Tools help to facilitates : data management, streaming etc

MasterNode (NameNode):

. Namenode functions/master

- Maintaining metadata Ex: File name, format, directory etc
- Manages user access to the data files
- Mapping the datablocks to the Datanodes in the cluster
- Performs files system operations such as opening and closing the files and directories
- Providing registration services for Datanode cluster membership and handling periodic healthbeats from the Datanodes
- Determines on which nodes the data should be replicated and deleting over replicated blocks.
- Processing the block reports sent by the Datablocks and maintaining the location where data blocks live

DataNode: Slave Node

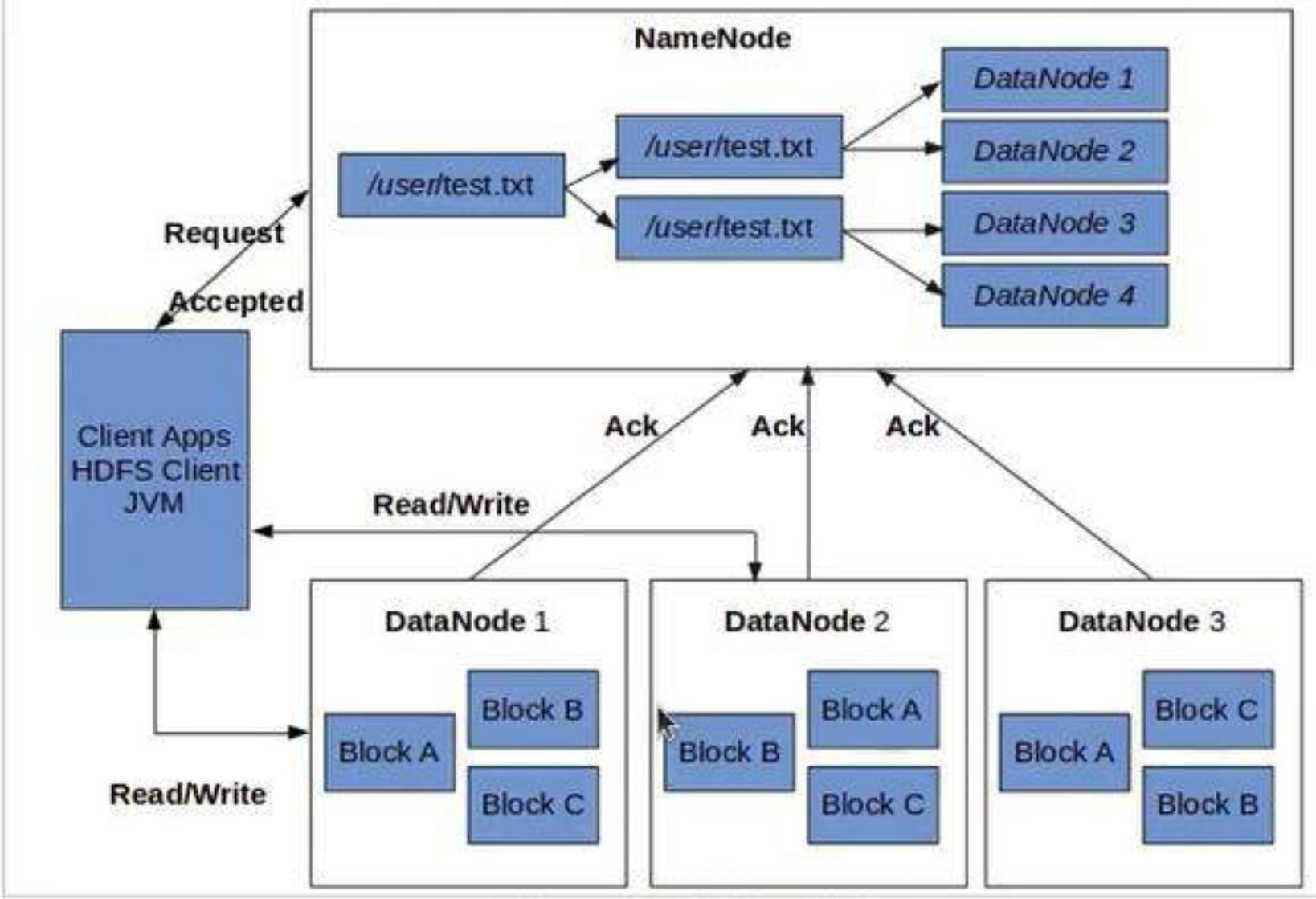
3. Datanode functions/worker

- Providing the block storage on the local file system
- Fulfilling the read/write requests from the clients
- Creating and deleting the datablocks
- Replicating data across the cluster
- Keeping in touch with the data node by sending periodic block reports and heartbeats.

Note:

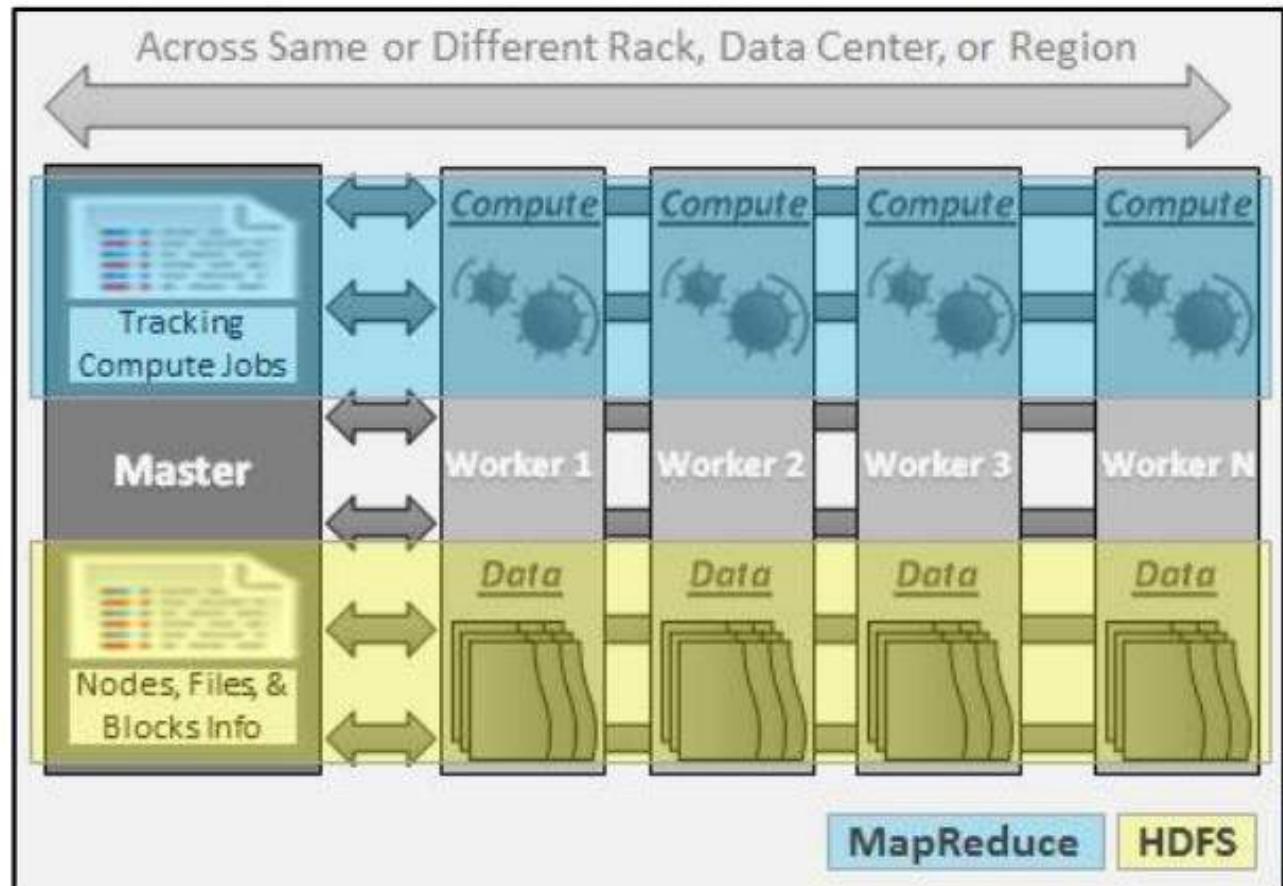
Heartbeat confirms the Datanode is alive and healthy, and a block report shows the blocks being managed by the Datanode.

How HDFS clients communicate with the NameNode and DataNodes

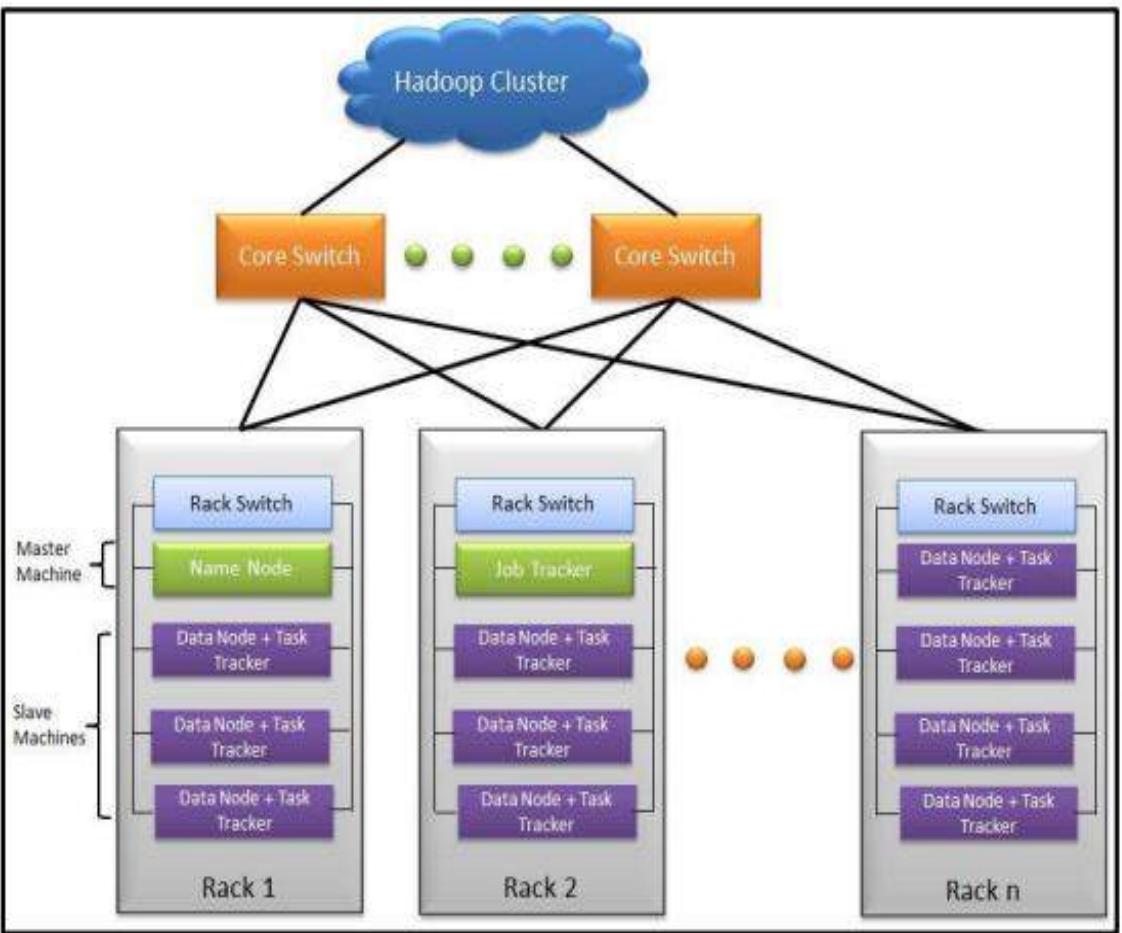


Hadoop Cluster

1. The architecture of Hadoop Cluster
2. Core Components of Hadoop Cluster
3. Work-flow of How File is Stored in Hadoop



1. Hadoop Cluster



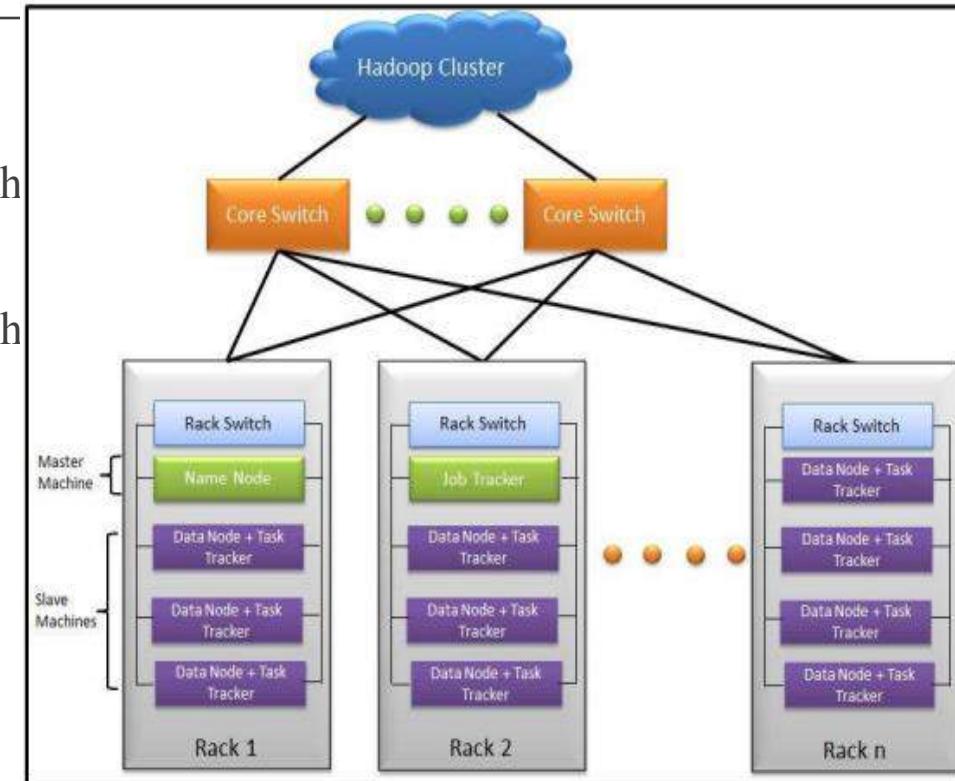
- These clusters run on low cost commodity computers.
- Hadoop clusters are often referred to as "**shared nothing**" systems because the only thing that is shared between nodes is the **network** that connects them.
- Large Hadoop Clusters are arranged in several racks.
- Network traffic between different nodes in the same rack is much more desirable than network traffic across the racks.
- Example: Yahoo's Hadoop cluster. They have more than **10,000** machines running Hadoop and nearly 1 petabyte of user data.
- A **small** Hadoop cluster includes a **single master node** and multiple **worker or slave node**.
- As discussed earlier, the entire cluster contains two layers.
- One of the layer of **MapReduce Layer** and another is of **HDFS Layer**.
- The master node consists of a JobTracker, TaskTracker, NameNode and DataNode.
- A slave or worker node consists of a DataNode and TaskTracker.
- It is also possible that slave node or worker node is **only data or compute node**.

Hadoop Cluster – 3 Components: Client, Master & Slave

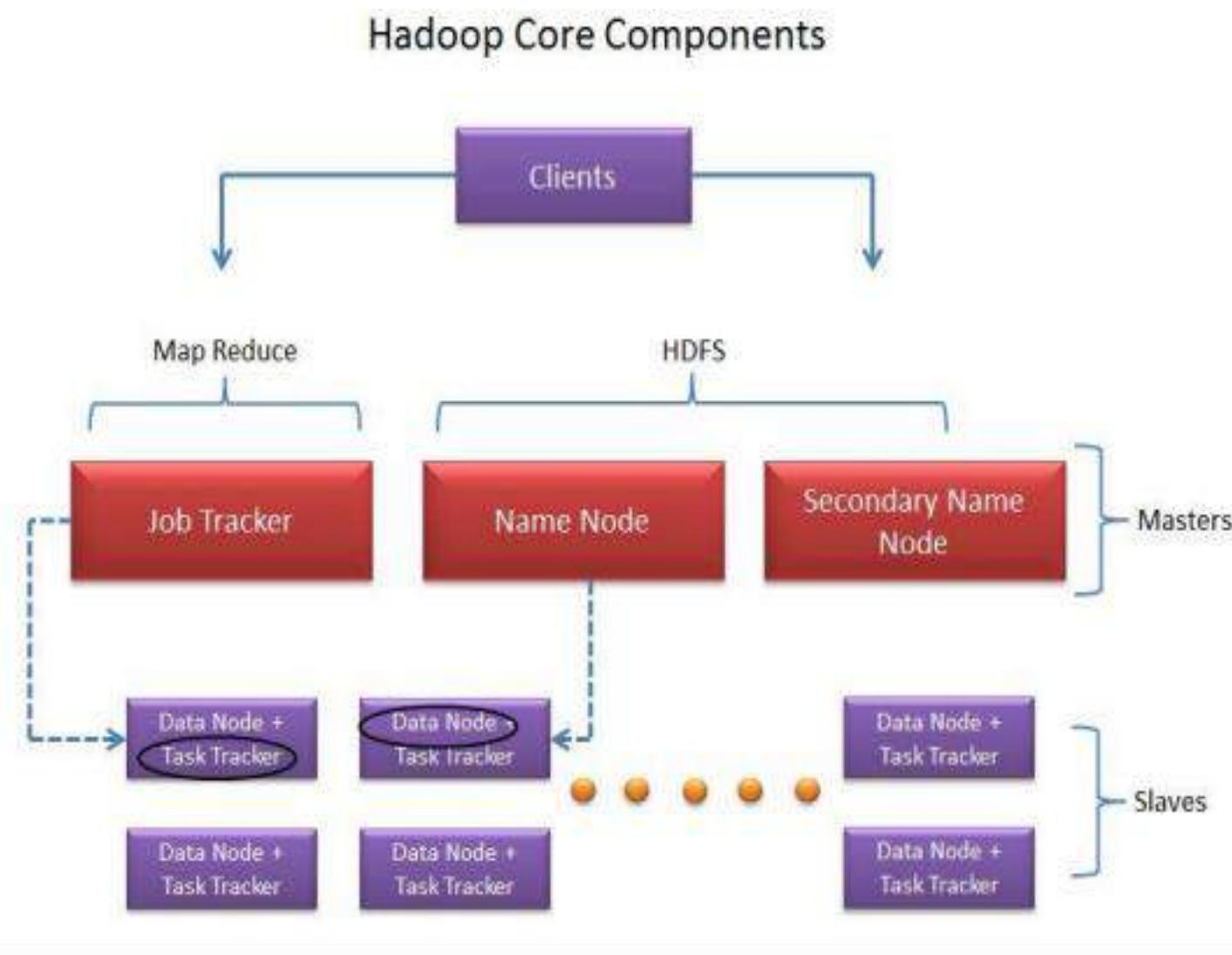
Hadoop Cluster would consist of

- 110 – Maximum Racks
- Rack - 40 slave machine
- At the top of each rack there is a **rack switch**
- Each slave machine(rack server in a rack) has cables coming out it from both the ends
- Cables are connected to rack switch at the top which means that top rack switch will have around 80 ports
- Global = **8 core switches**
- The rack switch has uplinks connected to core switches and hence connecting all other racks with uniform bandwidth, forming the Cluster
- In the cluster, you have few machines to act as Name node and as JobTracker.

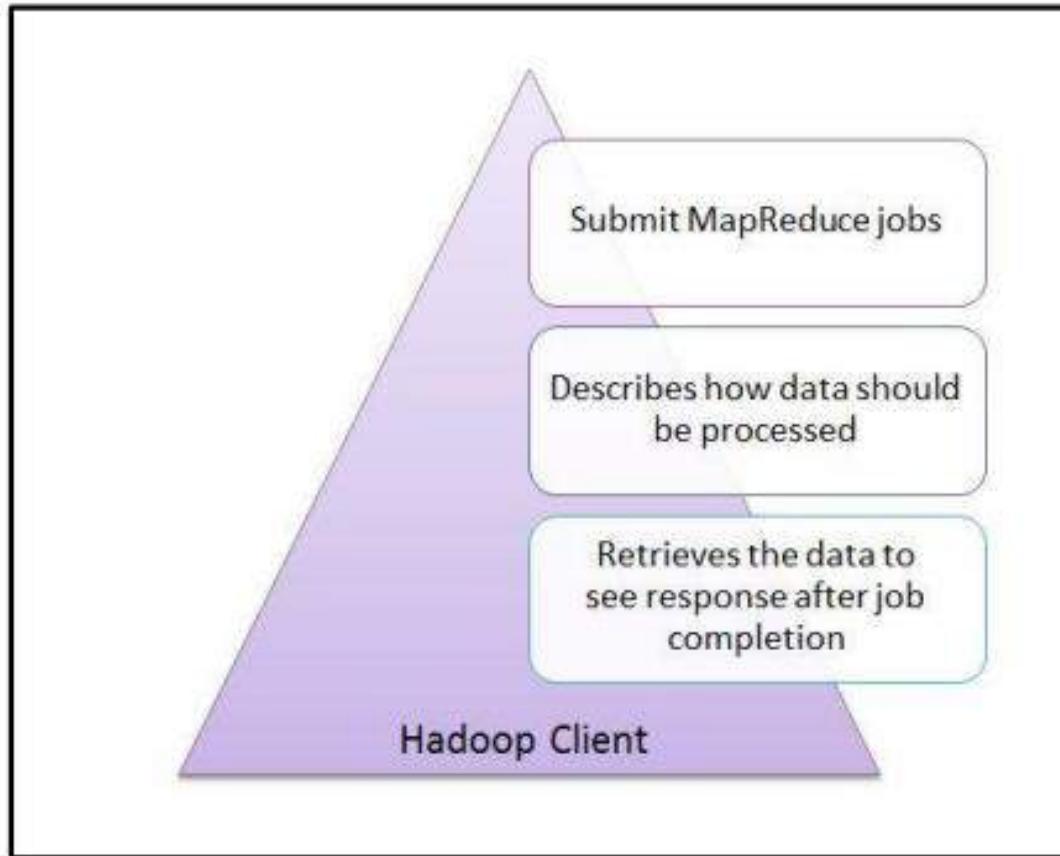
They are referred as Masters.



Cluster : Core Components



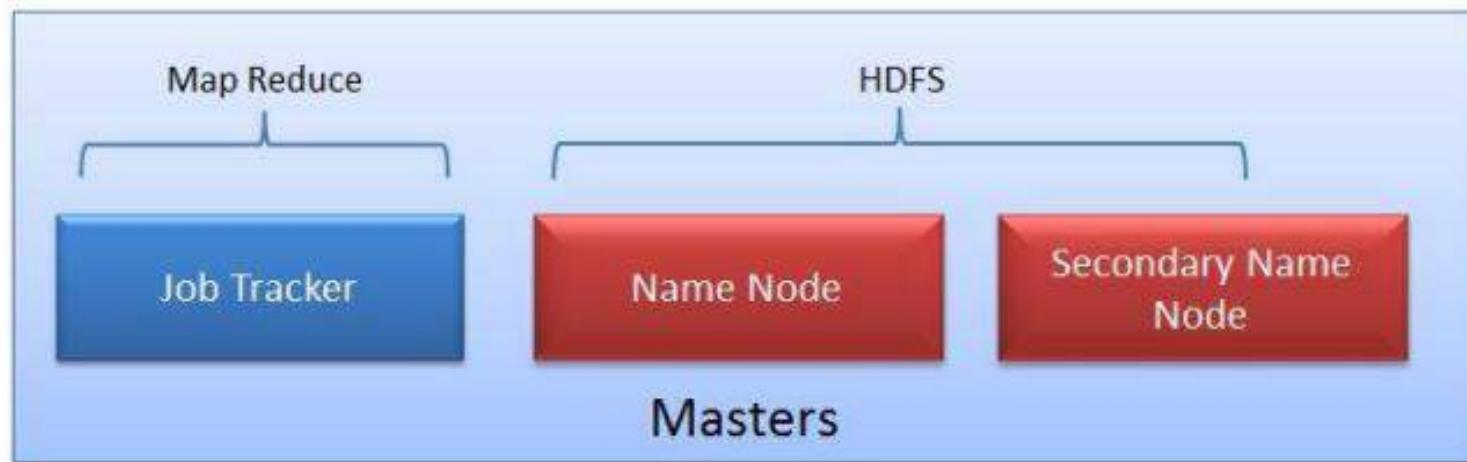
1. Client



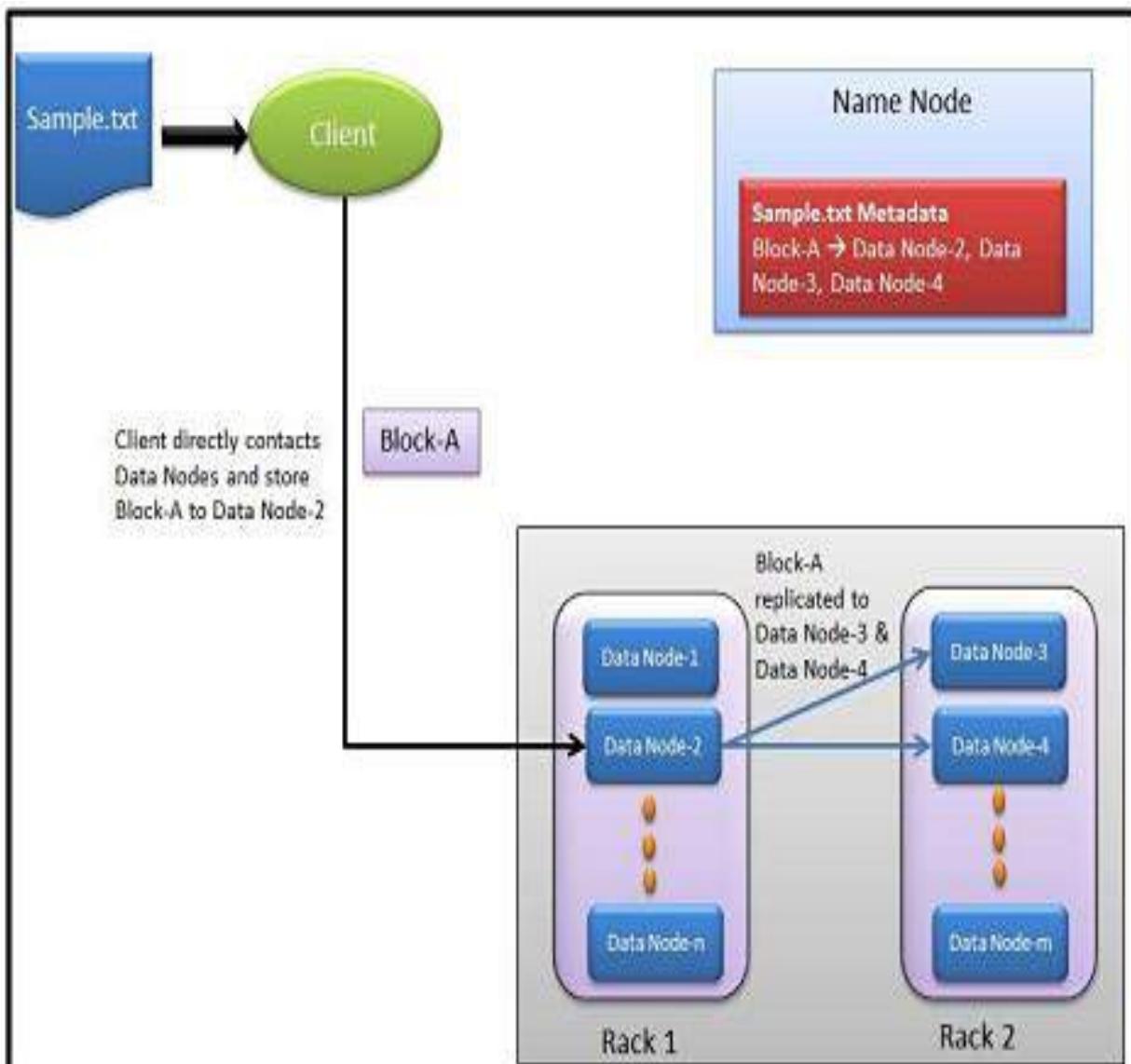
It is neither master nor slave, rather play a role of loading the data into cluster, submit MapReduce jobs describing how the data should be processed and then retrieve the data to see the response after job completion.

2. Masters: Name Node, Secondary Node & Job Tracker

The Masters consists of 3 components NameNode, Secondary Node name and JobTracker.



2.1: Name Node:



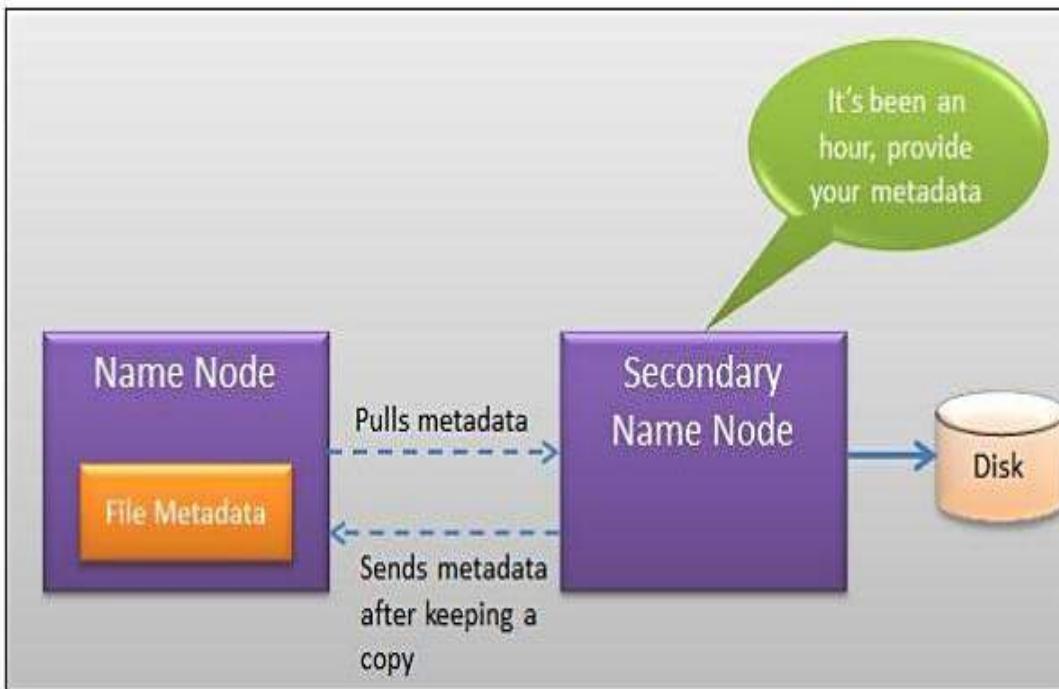
- NameNode oversees the **health of DataNode** and coordinates access to the data stored in DataNode.
- Name node keeps track of all the file system related information such as:
 - Which **section of file** is saved in **which part** of the cluster
 - **Last access time** for the files
 - **User permissions** like which user have access to the file

2.2 JobTracker:

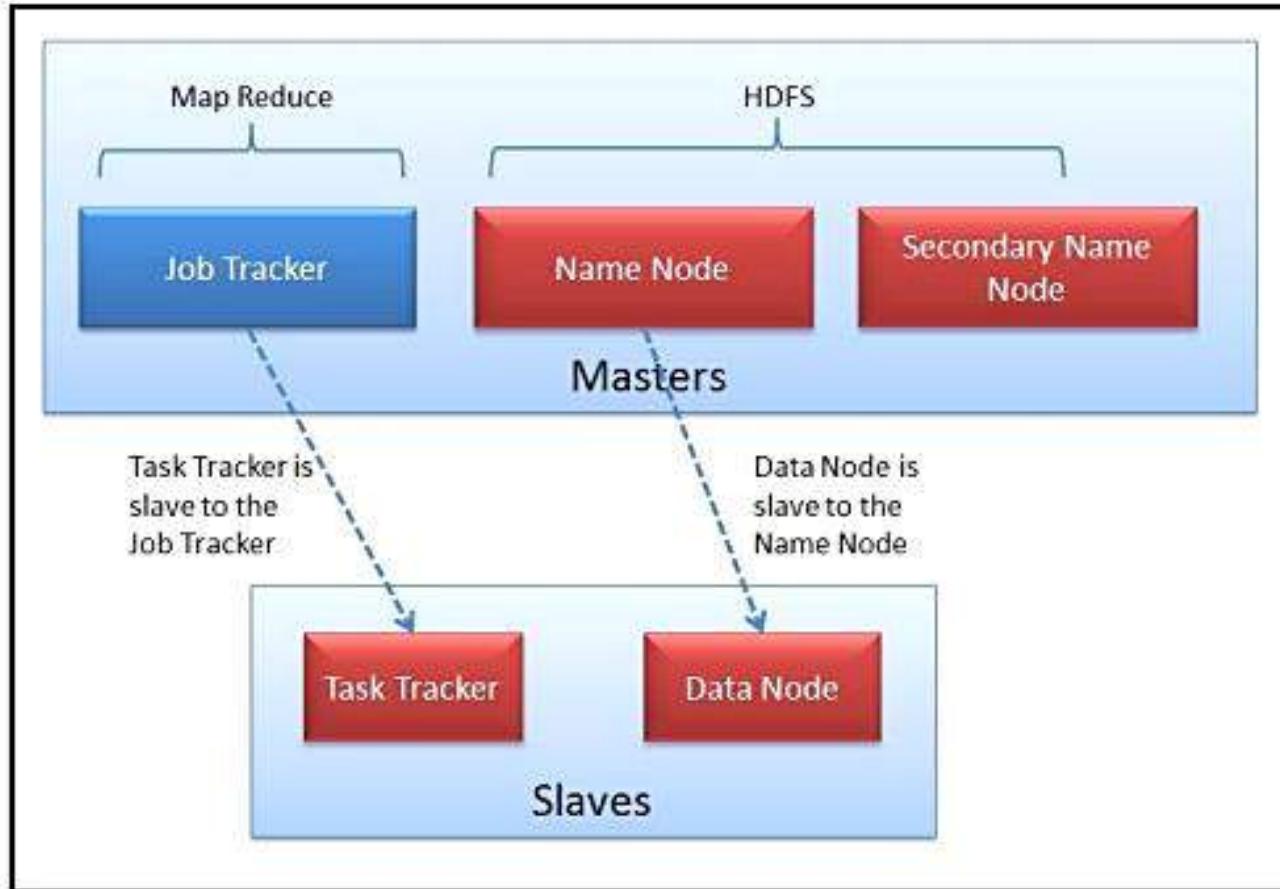
JobTracker : Coordinates the parallel processing of data using MapReduce.

2.3 Secondary Node:

- The job of Secondary Node is to contact NameNode in a periodic manner after certain time interval (by default 1 hour).
- NameNode which keeps all filesystem metadata in RAM has no capability to process that metadata on to disk.
- If NameNode crashes, you lose everything in RAM itself and you don't have any backup of filesystem.
- What secondary node does is it contacts NameNode in an hour and pulls copy of metadata information out of NameNode.
- It shuffle and merge this information into clean file folder and sent to back again to NameNode, while keeping a copy for itself.
- Hence Secondary Node is not the backup rather it does job of housekeeping.
- In case of NameNode failure, saved metadata can rebuild it easily.

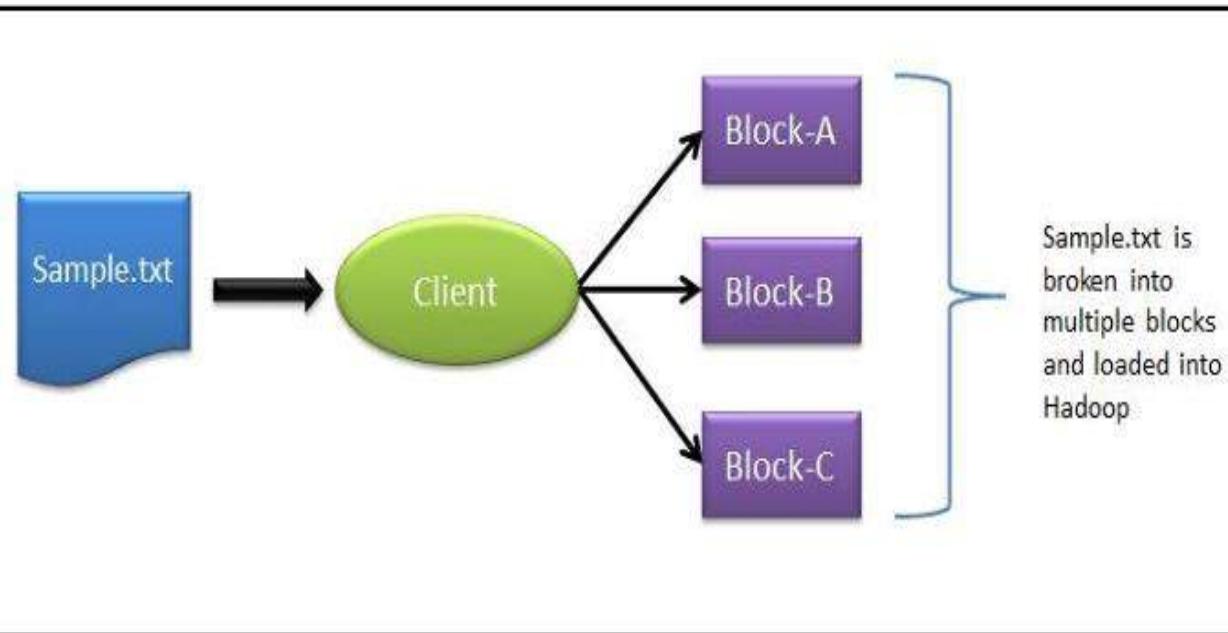


3: Slave



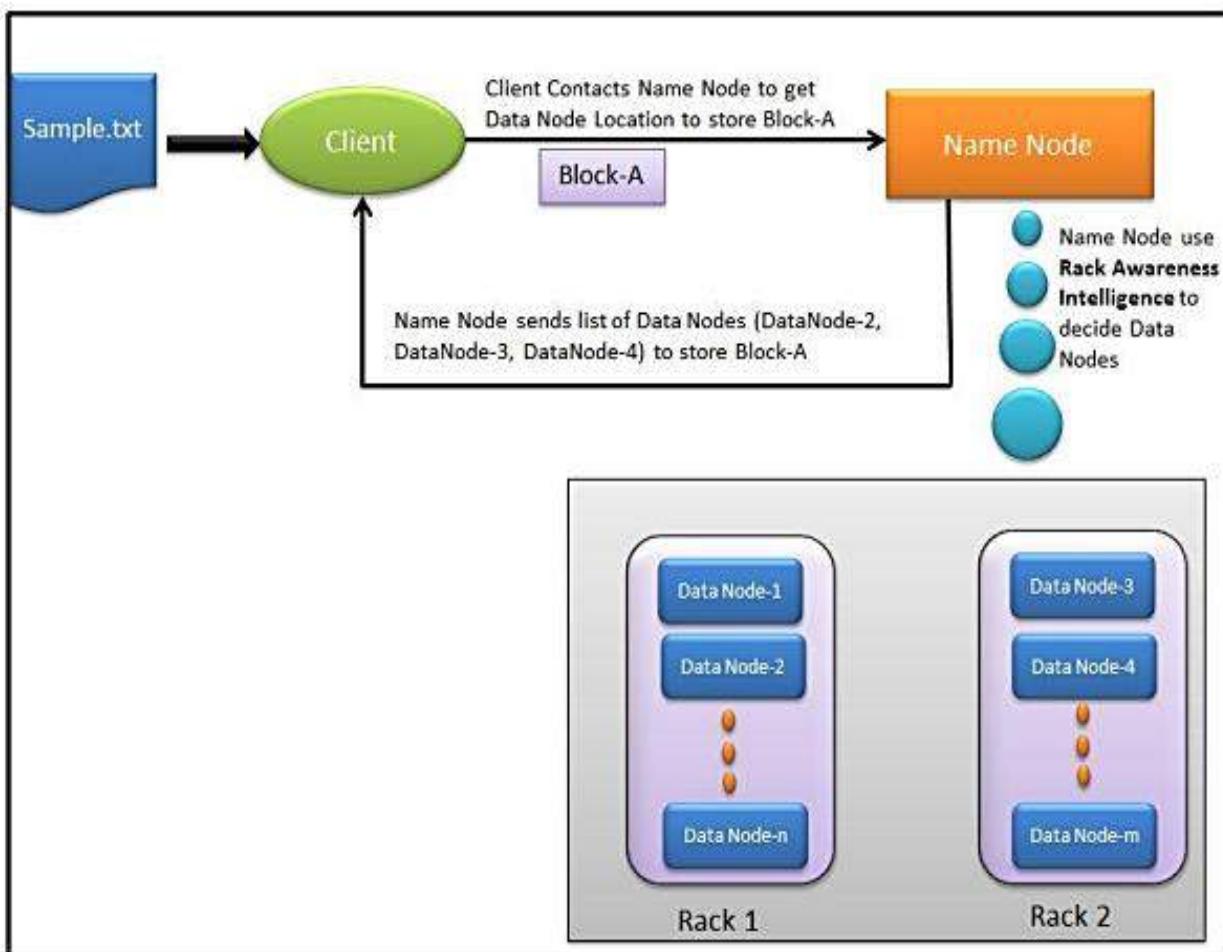
- Slave nodes are the majority of machines in Hadoop Cluster and are responsible to :
 - Store the data
 - Process the computation
- Each slave runs both a DataNode and Task Tracker daemon which communicates to their masters.
- The **Task Tracker** daemon is a **slave** to the **Job Tracker**
- **DataNode daemon** a slave to the **NameNode**

Loading File In Hadoop Cluster



- Client machine does this step and loads the Sample.txt into cluster.
- It breaks the sample.txt into smaller chunks which are known as "Blocks" in Hadoop context.
- Client put these blocks on different machines (data nodes) throughout the cluster.

Client knows that to which data nodes load the blocks?

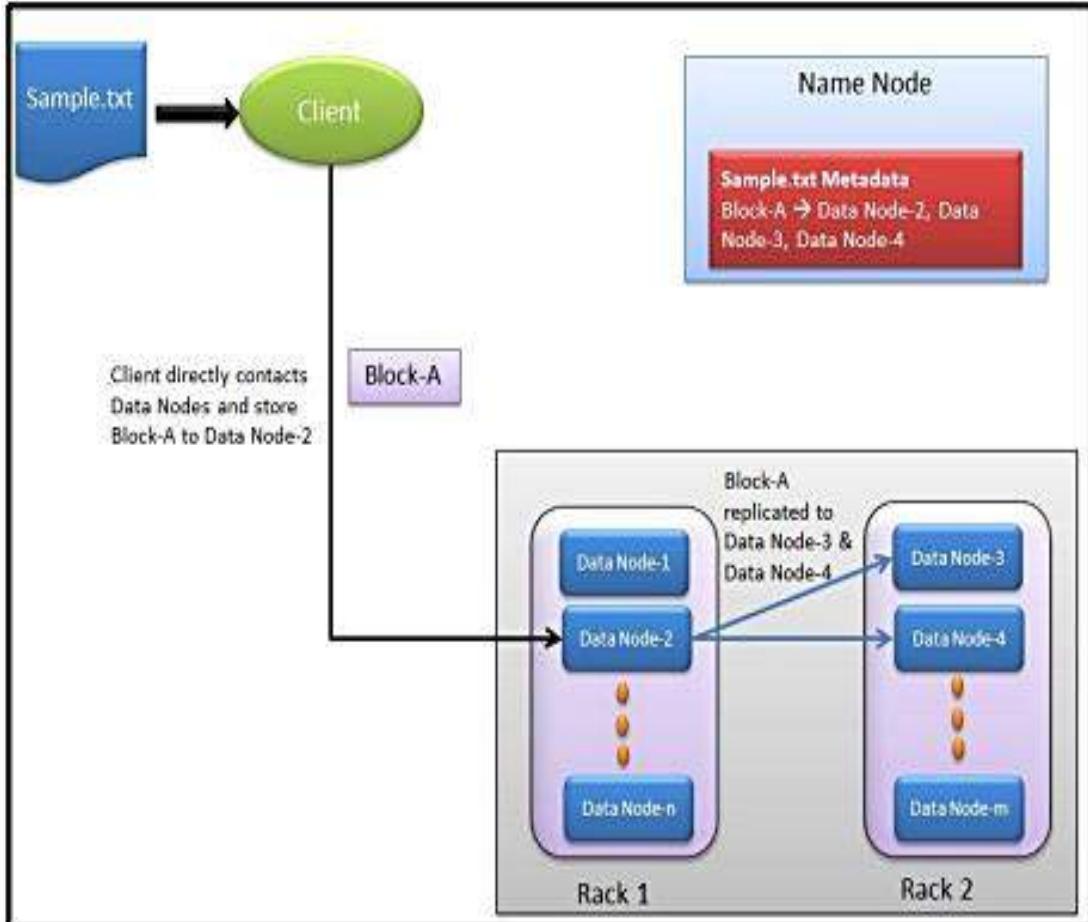


- 1) Now **NameNode** comes into picture.
- 2) The NameNode used its **Rack Awareness intelligence** to decide on which **DataNode** to provide.
- 3) For each of the data block (in this case Block-A, Block-B and Block-C), Client contacts NameNode and in response **NameNode** sends an ordered list of 3 **DataNodes**.

For example in response to Block-A request, Node Name may send DataNode-2, DataNode-3 and DataNode-4.

- ✓ Block-B DataNodes list DataNode-1, DataNode-3, DataNode-4 and for Block C data node list DataNode-1, DataNode-2, DataNode-3. Hence
 - ❖ Block A gets stored in DataNode-2, DataNode-3, DataNode-4
 - ❖ Block B gets stored in DataNode-1, DataNode-3, DataNode-4
 - ❖ Block C gets stored in DataNode-1, DataNode-2, DataNode-3
- ✓ Every block is replicated to more than 1 data nodes to ensure the data recovery on the time of machine failures. That's why NameNode send 3 DataNodes list for each individual block

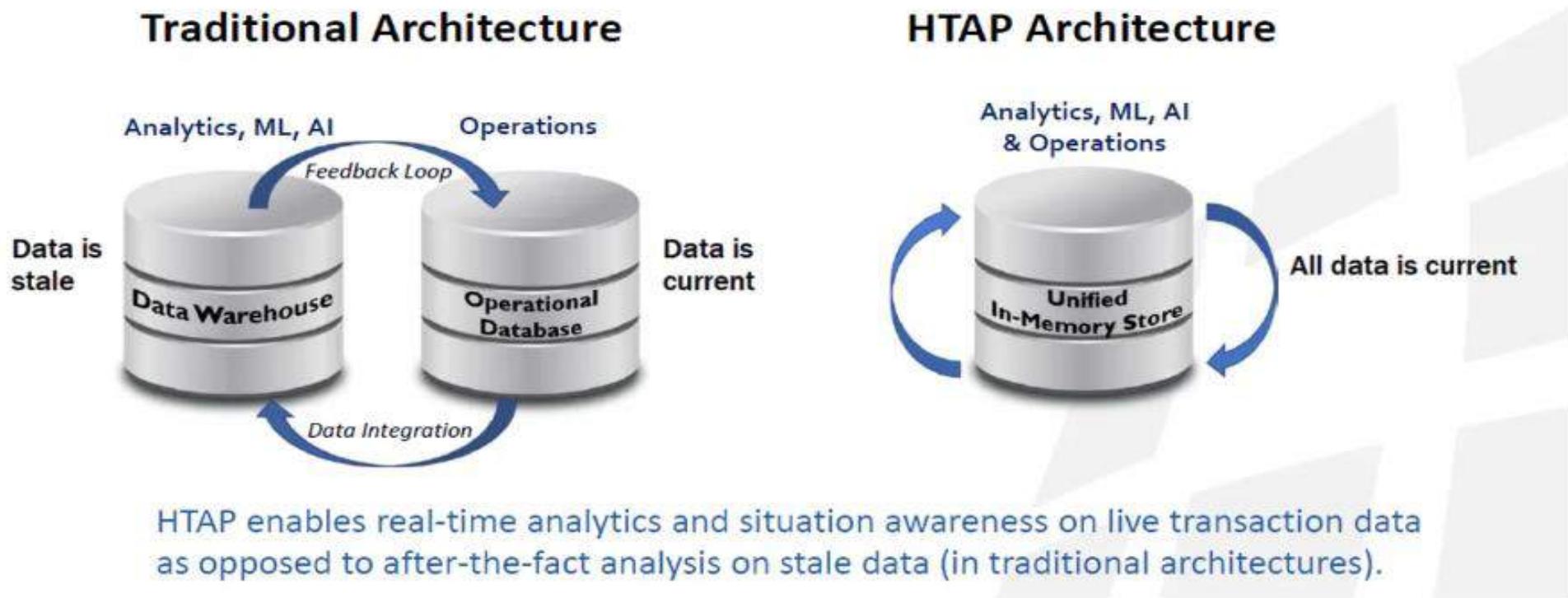
Block replication



1. Client write the data block directly to one DataNode.
2. DataNodes then replicate the block to other Data nodes.
3. When 1 block gets written in all 3 DataNode then only cycle repeats for next block.
4. In Hadoop Gen 1 there is only one NameNode.
5. In Hadoop Gen2 there is active passive model in NameNode where one more node "Passive Node" comes in picture.
6. The default setting for Hadoop is to have 3 copies of each block in the cluster.
7. This setting can be configured with "dfs.replication" parameter of hdfs-site.xml file.
8. Keep note that Client directly writes the block to the DataNode without any intervention of NameNode in this process.

Digital Transformation is Driving Hybrid Transaction/Analytical Processing (HTAP)

"IMC-enabled HTAP can have a transformational impact on the business." — Gartner 2/17



In-memory computing is said to enable HTAP (Hybrid Transaction/Analytical Processing), which brings benefits in terms of unified architecture and quick access to data and insights. Image: GridGain



MapReduce

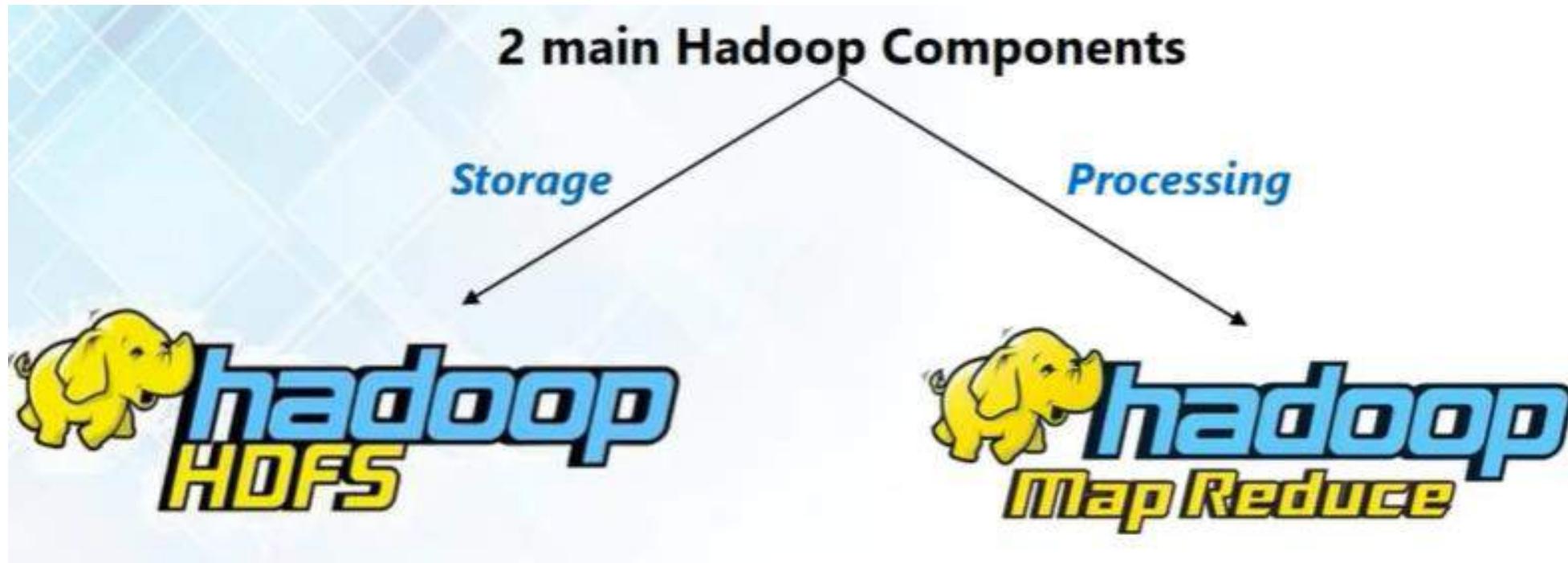
Part - 3

BY : DR.RASHMI L MALGHAN & MS.
SHAVANTREVVA S BILAKERI

AGENDA

1. What is Hadoop MapReduce?
2. MapReduce In Nutshell
3. Two Advantages of MapReduce
4. Hadoop MapReduce Approach with an Example
5. Hadoop MapReduce/YARN Components
6. YARN With MapReduce
7. Yarn Application Workflow

Hadoop Main Components:



Comparison: Convention Vs. MapReduce

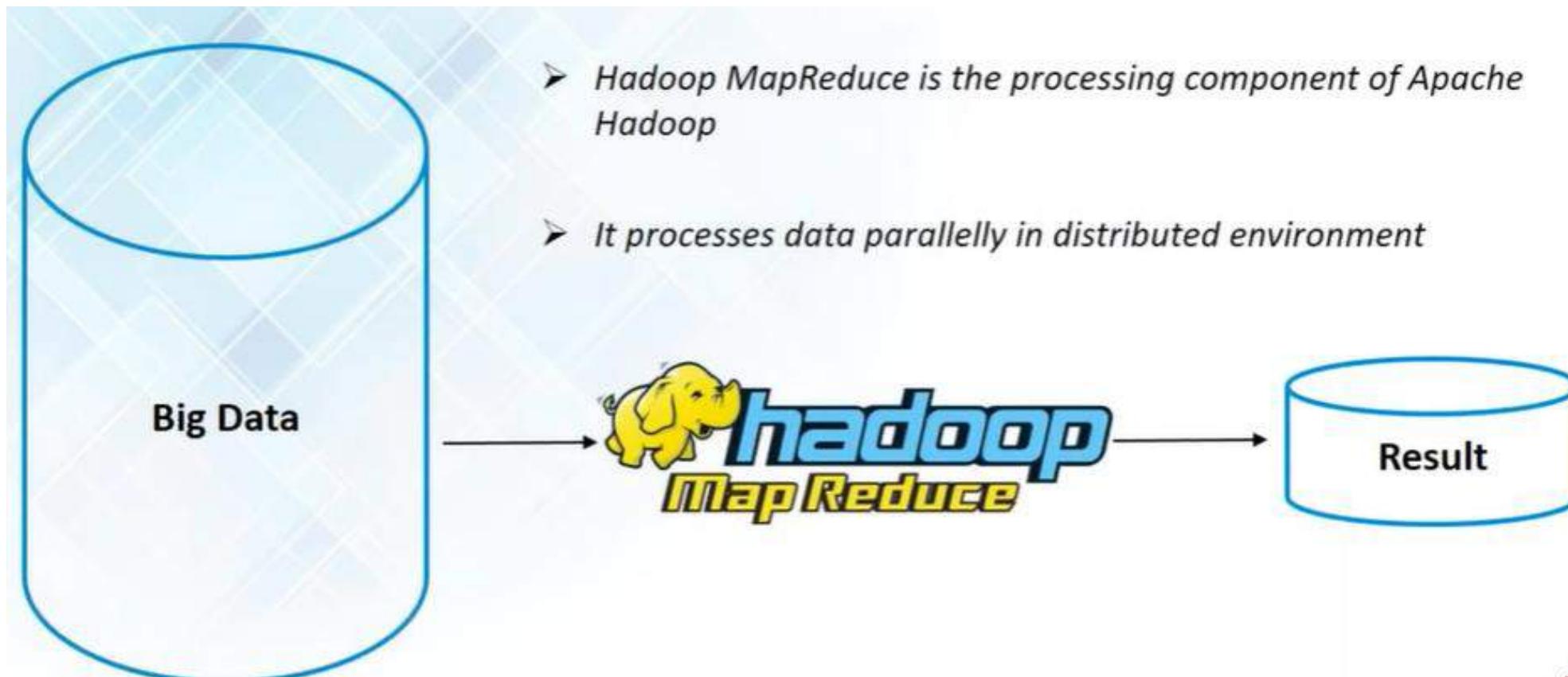
- **Conventional Approach**

- Single-machine processing.
- Suitable for **small datasets**.
- Limited **scalability**.
- May become a **bottleneck** for big data.

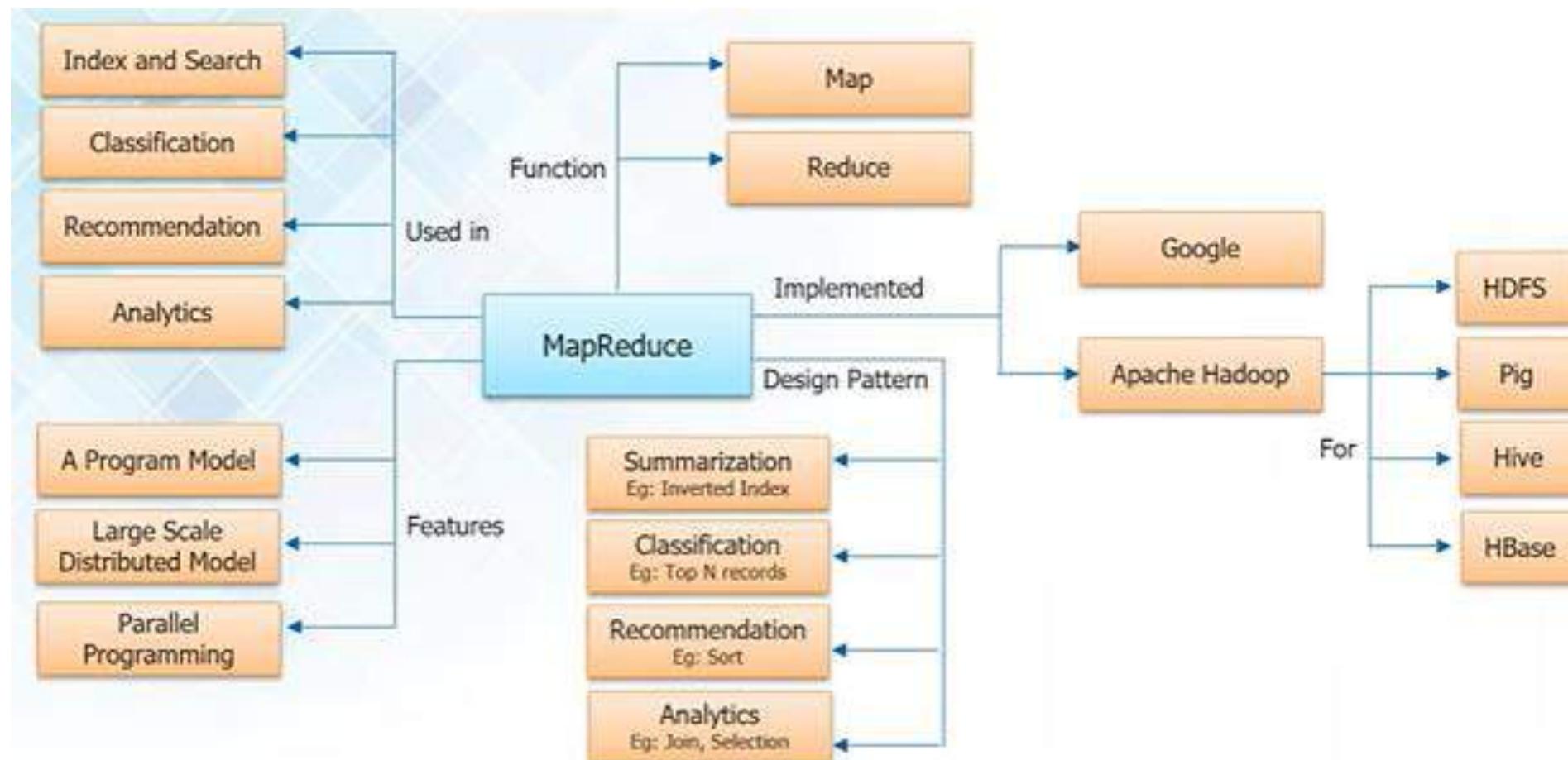
- **MapReduce Approach**

- **Distributed** processing across a cluster.
- Scalable for **large datasets**.
- Handles **parallel processing** efficiently.
- Tackles the **challenges of big data**

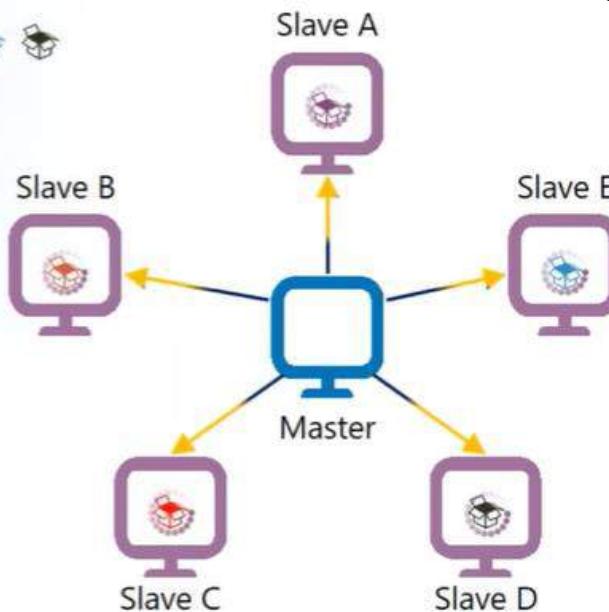
MapReduce:



MapReduce - Nutshell

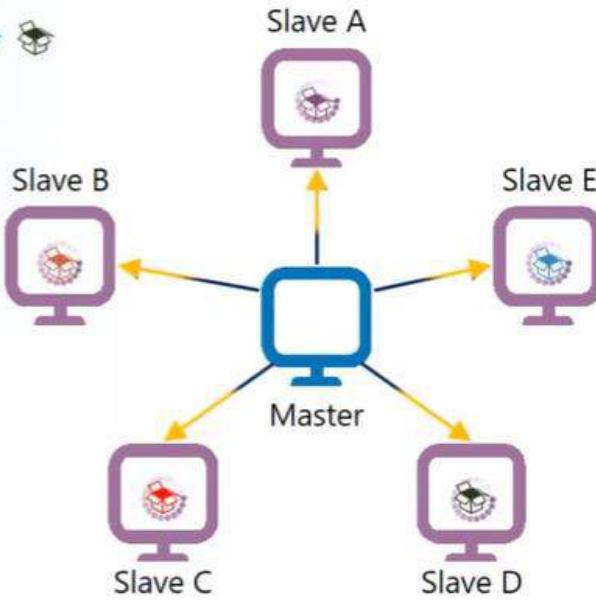


Advantage 1: Parallel Processing



- In Hadoop data gets divided in to small chunks called as HDFS blocks.
- Scalability
- Fault Tolerance
- Flexibility
- Efficient Resource Utilization

Advantage 2: Data Locality



- **Data Locality** - MapReduce is processing at “**Location**” where data is stored rather bringing data to “**centralised server**”
- **Client** Sends/submits data – To **Resource Manager** decides – Data usually resides on “**Nearest Data Node**” to reduce the network bandwidth.
- Master = Name Node – In Hdfs [Storage] , **Master** – Consideration -**resources manager** - **Processing**
- Processing – Master sends the logic to slaves that require for its job processing.
- Smaller chunk of data is getting processed in **multiple locations in parallel**.
- It saves “**time**” & “**network bandwidth**”- required to move/transfer large amount of data from one point/location to another.
- Results will be sent back to master machines and sent to client machine

MapReduce: Phases and Deamons

MapReduce Framework

Phases:

Map(): Converts input into Key Value pair

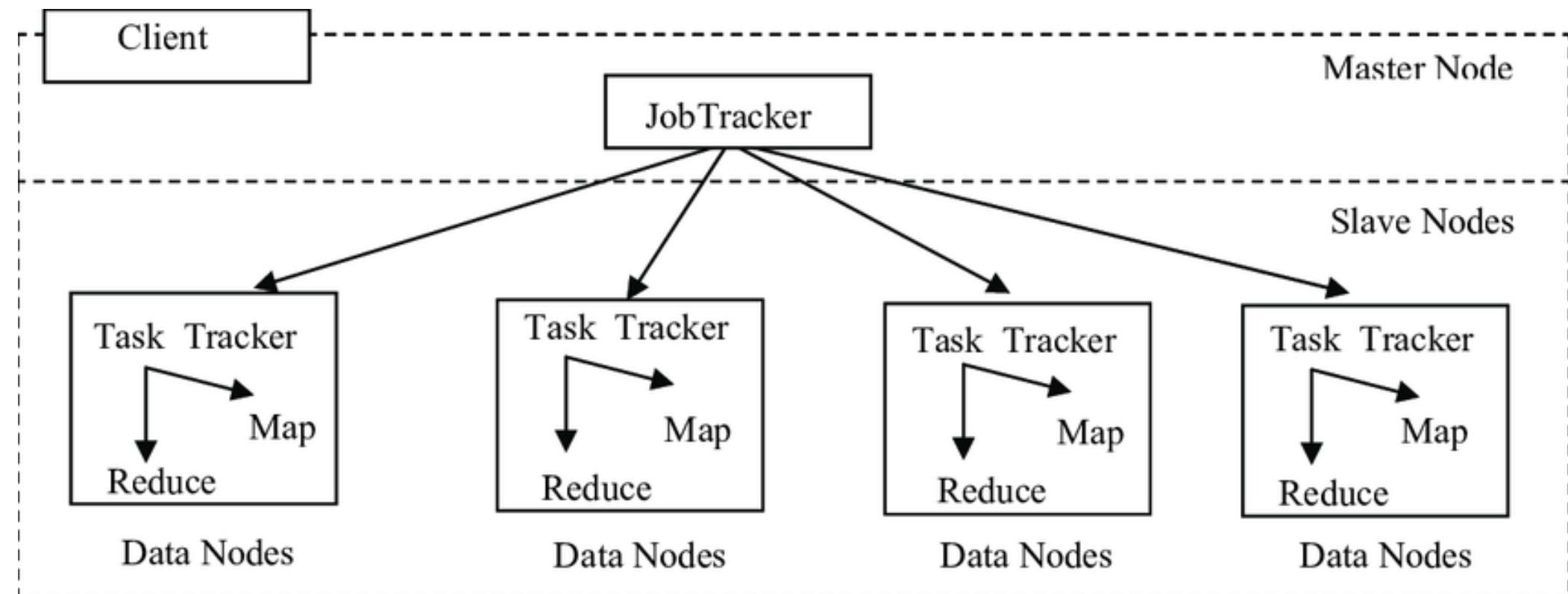
Reduce(): Combines output of mappers and produces a reduced result set.

Daemons:

JobTracker: Master schedules task

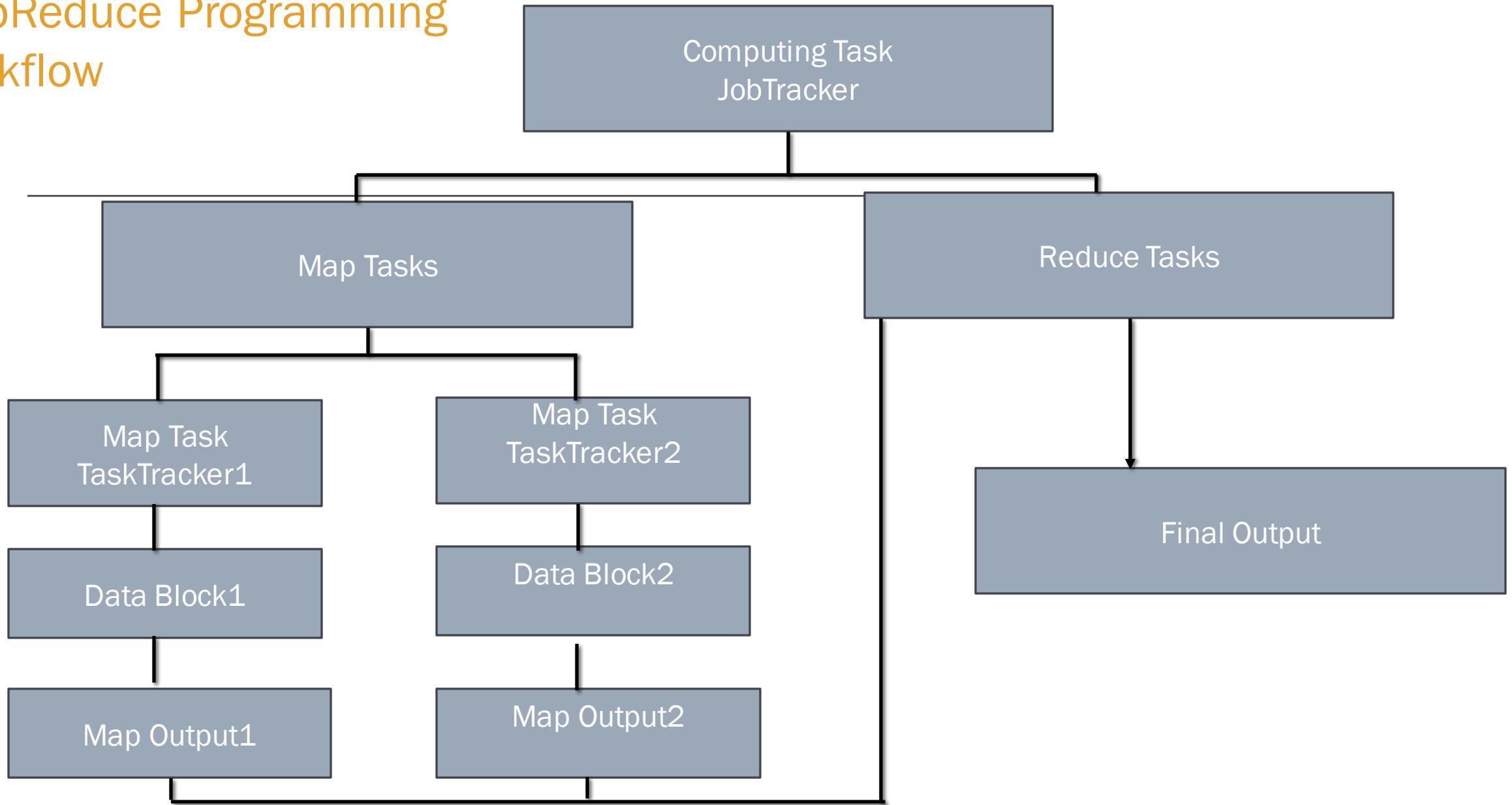
TaskTracker: Slave executes task

JobTracker and TaskTracker Interaction



MapReduce Programming

Workflow



Phases in Map and Reduce Tasks

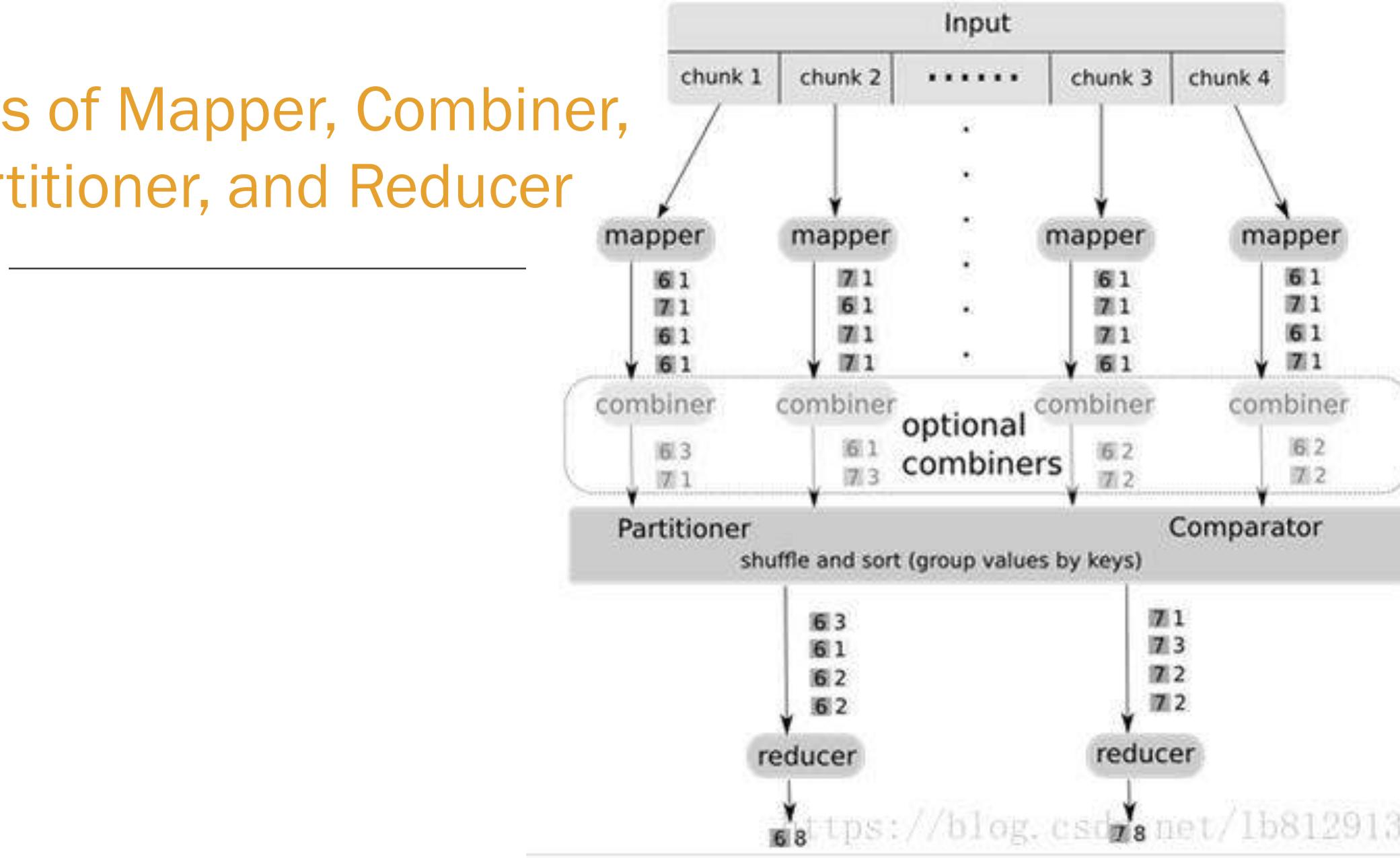
Map Phases

1. Record Reader
2. Mapper
3. Combiner
4. Partitioner

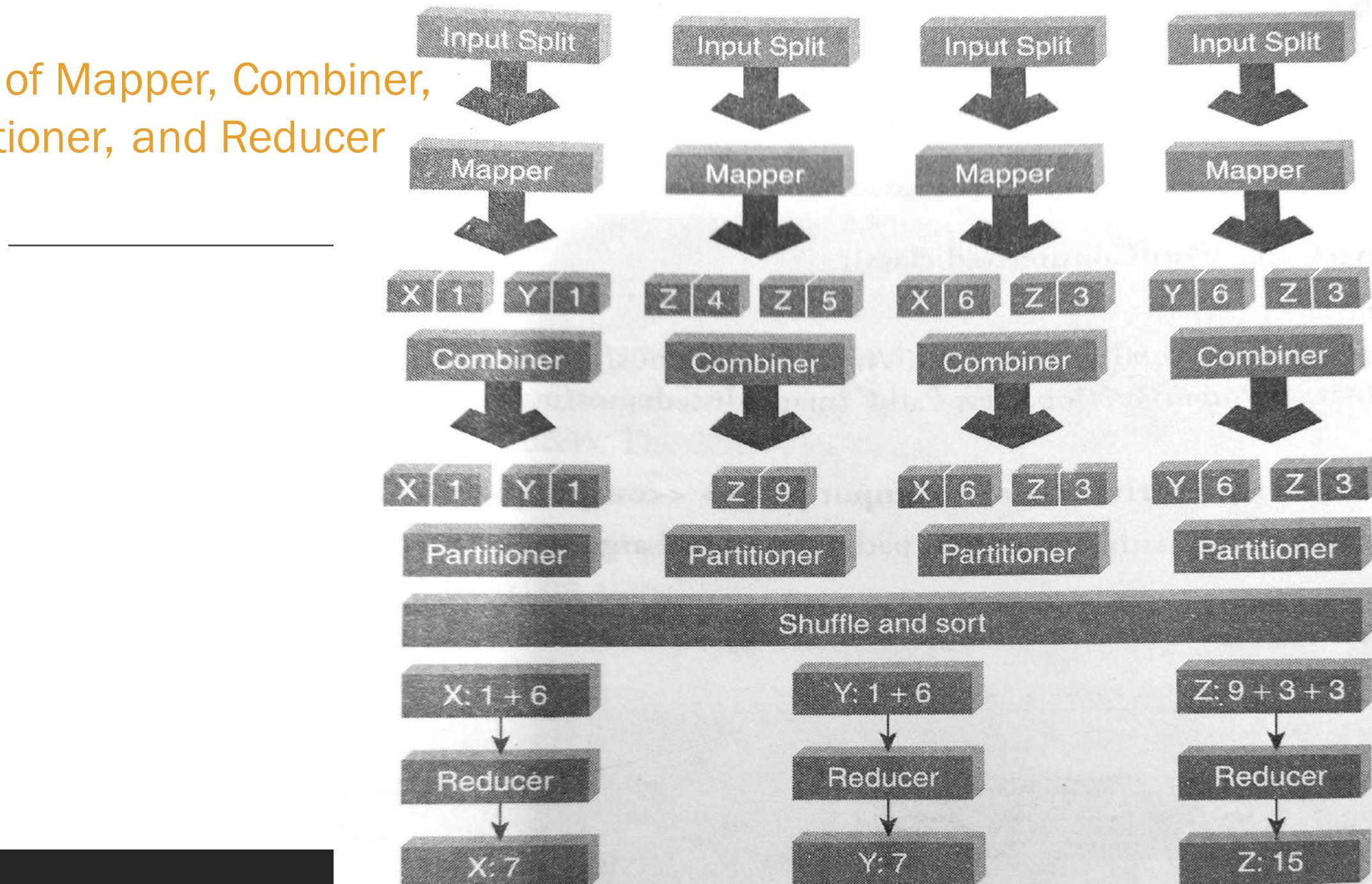
Reduce Phases

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

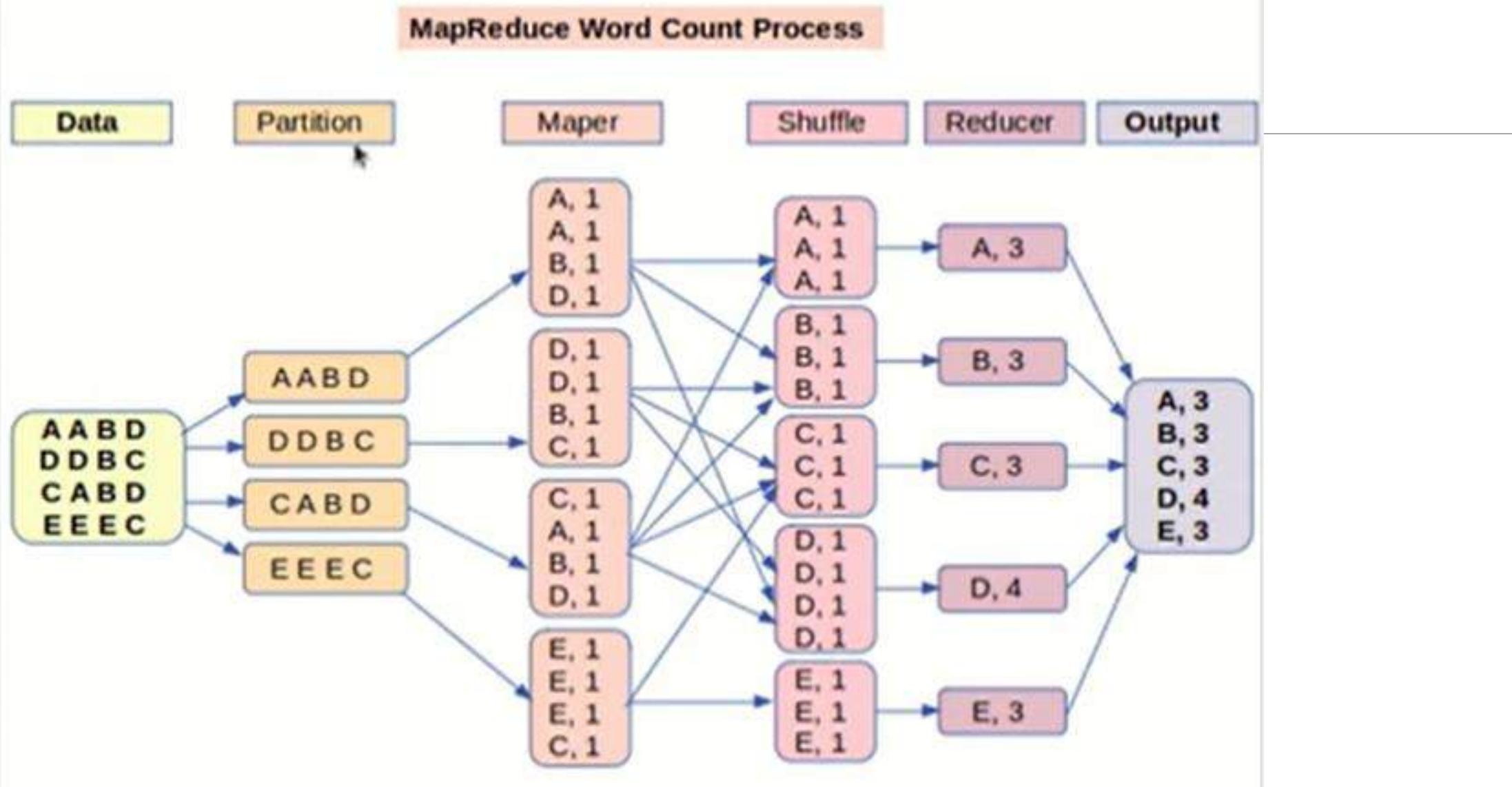
Chores of Mapper, Combiner, Partitioner, and Reducer



Chores of Mapper, Combiner, Partitioner, and Reducer



MapReduce: Word count problem



Word Count Program using MapReduce

Step 1: Create a file with the name `word_count_data.txt` and add some data to it

```
dikshant@dikshant-Inspiron-5567:~/Documents$ touch word_count_data.txt
dikshant@dikshant-Inspiron-5567:~/Documents$ nano word_count_data.txt
dikshant@dikshant-Inspiron-5567:~/Documents$ cat word_count_data.txt
geeks for geeks is best online coding platform
welcome to geeks for geeks hadoop streaming tutorial
dikshant@dikshant-Inspiron-5567:~/Documents$
```

Step 2: Create a `mapper.py` file that implements the mapper logic.

Mapper.py

```
#!/usr/bin/env python

# import sys because we need to read and write data to STDIN and
# STDOUT
import sys

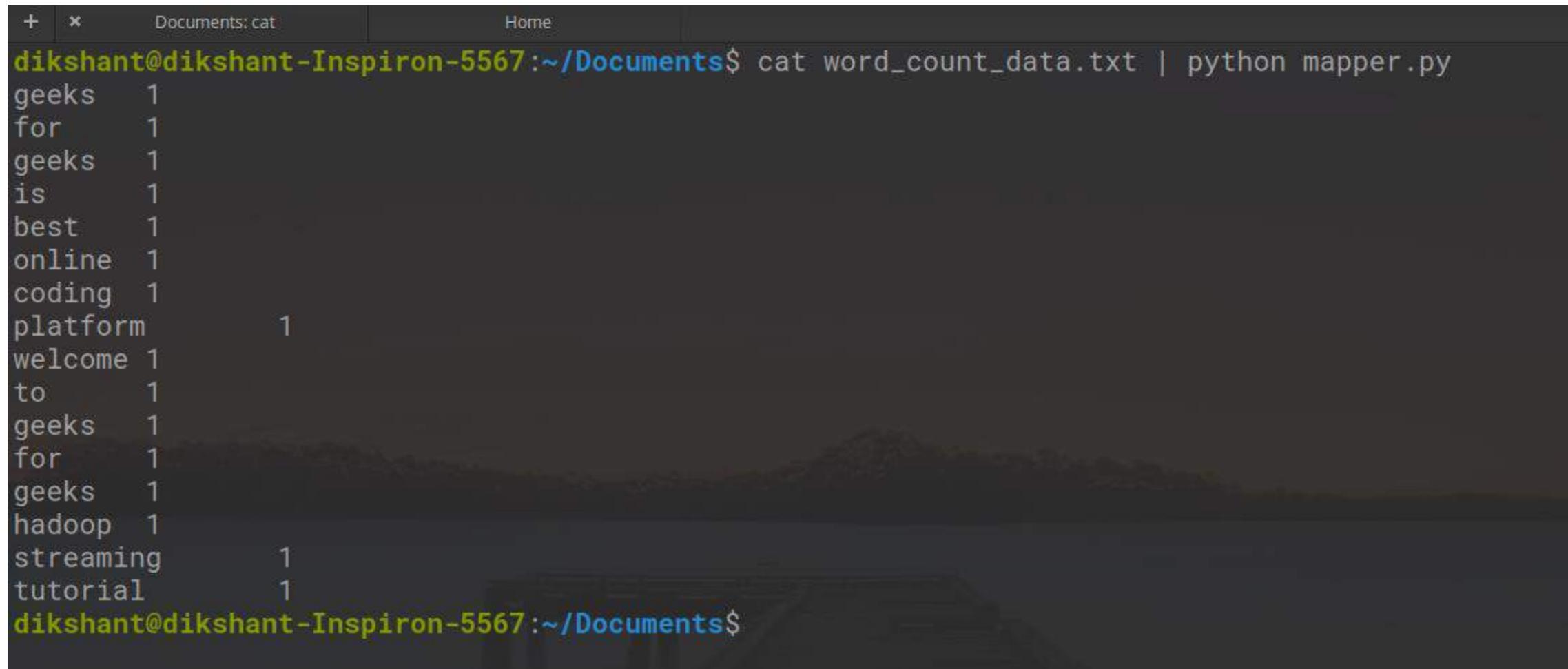
# reading entire line from STDIN (standard input)
for line in sys.stdin:
    # to remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()

    # we are looping over the words array and printing the word
    # with the count of 1 to the STDOUT
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        print '%s\t%s' % (word, 1)
```

#! is known as shebang and used for interpreting the script.

Test mapper.py locally

```
cat word_count_data.txt | python mapper.py
```



The screenshot shows a terminal window with a dark background and light-colored text. At the top, it says "Documents: cat" and "Home". The command entered is "dikshant@dikshant-Inspiron-5567:~/Documents\$ cat word_count_data.txt | python mapper.py". The output shows the following word counts:

```
geeks 1
for 1
geeks 1
is 1
best 1
online 1
coding 1
platform 1
welcome 1
to 1
geeks 1
for 1
geeks 1
hadoop 1
streaming 1
tutorial 1
```

The terminal prompt "dikshant@dikshant-Inspiron-5567:~/Documents\$" is visible at the bottom.

Step 3: Create a `reducer.py` file that implements the reducer logic.

Content: `#!/usr/bin/python3`

```
"reducer.py"
import sys
current_word = None
current_count = 0

for line in sys.stdin:
    # remove leading and trailing whitespaces
    line = line.strip()
    # parse the input we got from mapper.py
    word, count = line.split('\t')
    count = int(count)

    if current_word == word:
        current_count += count
    else:
        if current_word:
            print ('%s\t%s' % (current_word, current_count))
        current_count = count
        current_word = word
    if current_word == word:
        print ('%s\t%s' % (current_word, current_count))
```

Test mapper and reduce

```
cat word_count_data.txt | python mapper.py | sort | python reducer.py
```

```
dikshant@dikshant-Inspiron-5567:~/Documents$ cat word_count_data.txt | python mapper.py | sort -k1,1 | python reducer.py
best      1
coding    1
for       2
geeks     4
hadoop    1
is        1
online    1
platform  1
streaming 1
to        1
tutorial  1
welcome   1
dikshant@dikshant-Inspiron-5567:~/Documents$
```

```
dikshant@dikshant-Inspiron-5567:~/Documents$ cat word_count_data.txt | python mapper.py | sort -k1,1 | python reducer.py
best      1
coding    1
for       2
geeks     4
hadoop    1
is        1
online    1
platform  1
streaming 1
to        1
tutorial  1
welcome   1
dikshant@dikshant-Inspiron-5567:~/Documents$
```

Step 4: Now let's start all our Hadoop daemons with the command:

start-all.sh

```
hdfs dfs -mkdir /word_count_in_python
```

```
hdfs dfs -copyFromLocal /home/ssb/Documents/word_count_data.txt /word_count_in_python
```

```
chmod 777 mapper.py reducer.py
```

Step 5: Now download the latest **hadoop-streaming jar file**. Then place, this Hadoop,-streaming jar file to a place from you can easily access it.

Step6: Run our python files with the help of the Hadoop streaming utility

```
hadoop jar /home/ssb/Documents/hadoop-streaming-  
2.7.3.jar \
```

```
> -input /word_count_in_python/word_count_data.txt \
```

```
> -output /word_count_in_python/output \
```

```
> -mapper /home/ssb/Documents/mapper.py \
```

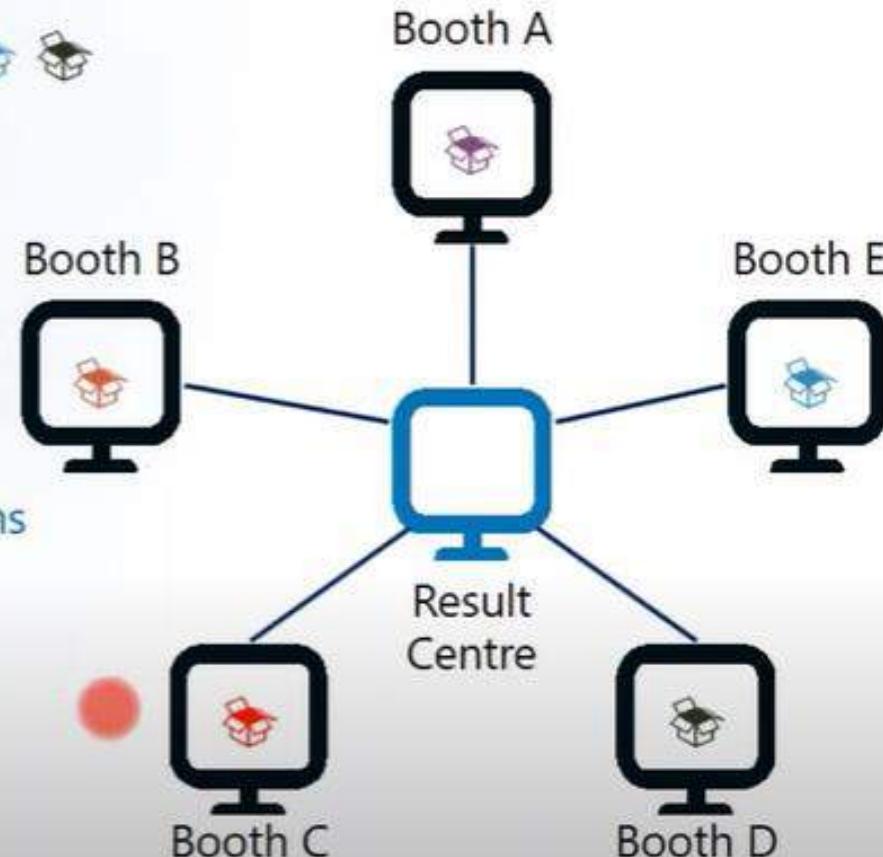
```
> -reducer /home/ssb/Documents/reducer.py
```

Example: Election Votes Counting

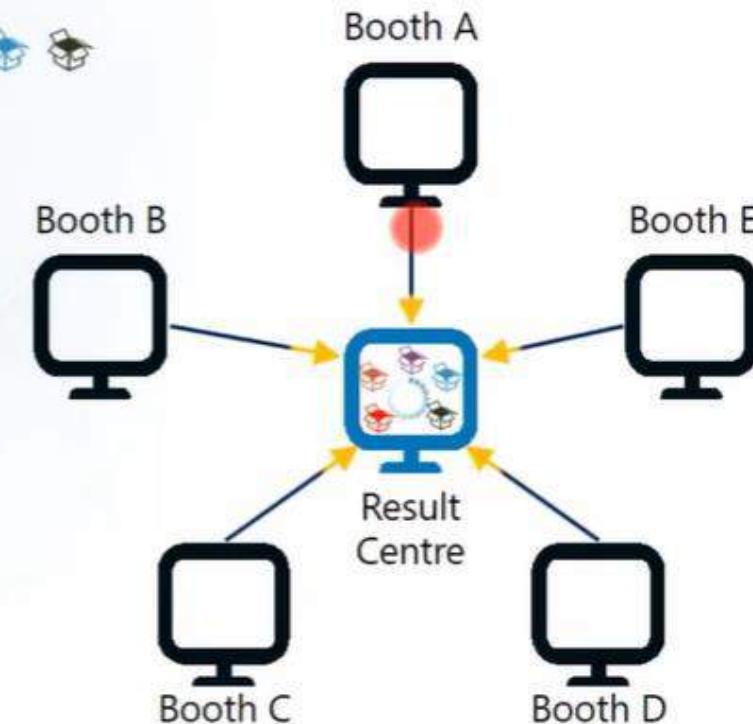
Election Votes Casting

- Votes is stored at different Booths
- Result Centre has the details of all the Booths

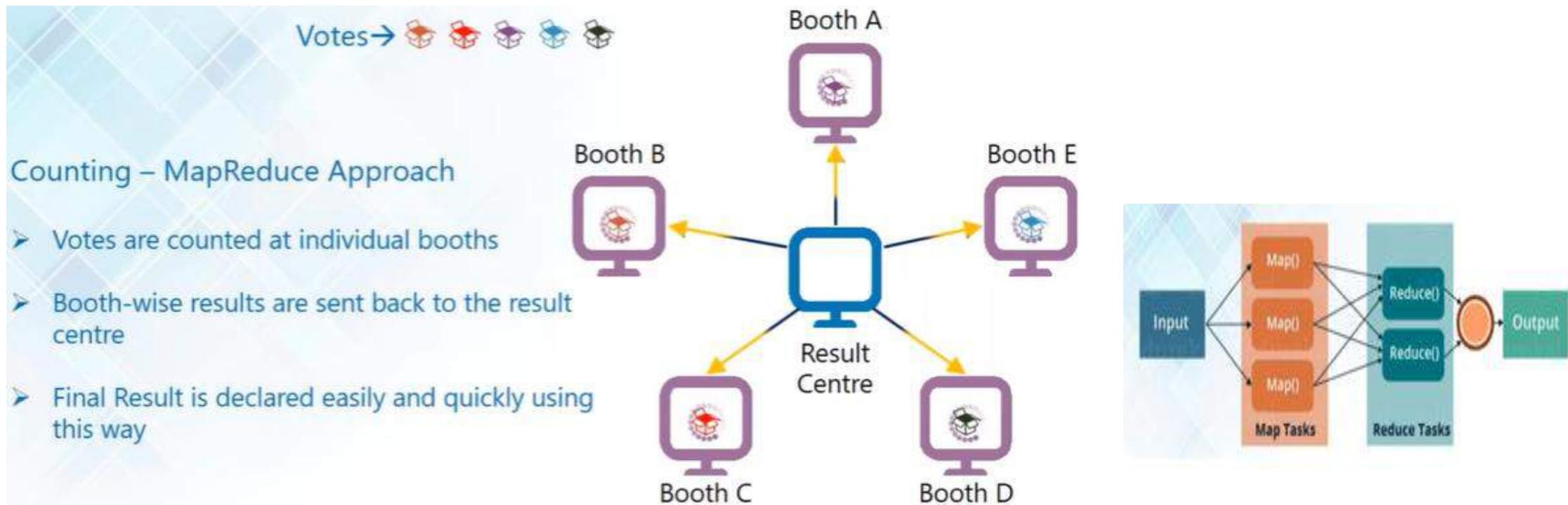
Data → 



Traditional Approach: Election Votes Counting

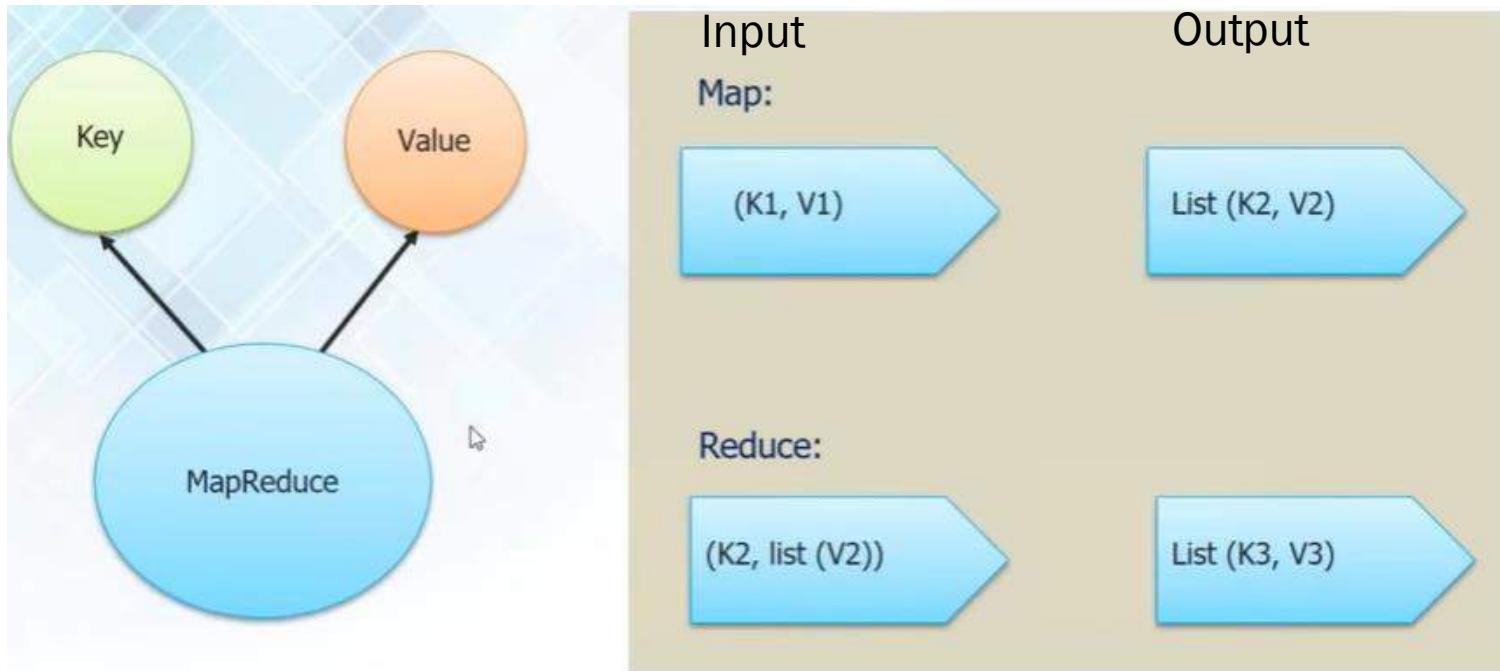


MapReduce Approach: Election Votes Counting



- Counting Part of respective booth was done by map function and sent to reducer.
- Combining of results was done by reducer function

Anatomy of MapReduce:



Exercise: MapReduce

1. Search specific keyword in a file
 2. Sort data by student name
 3. Arrange the data on user ID, then within user ID sort them in increasing order of page count
-
1. Write MapReduce program to find unit wise salary

-
- ```
graph TD; A[1. hadoop namenode -format] --> C["start-all.sh"]; B[2. hadoop secondarynamenode] --> C; C --> D[3. hadoop namenode]; C --> E[4. hadoop datanode]; E --- F[5. hadoop dfsadmin -report]; F --- G[6. hadoop mradmin -refreshNodes]; G --- H[7. hdfs dfsadmin -setBalancerBandwidth <bandwidth>]; H --- I[8. hdfs fsck /user/example/data]; I --- J[9. hdfs fsck /path/to/hdfs/directory -files -blocks -locations]; J --- K[10. hdfs fsck /path/to/hdfs/directory -files -blocks -locations -racks -replicaDetails]; K --- L[11. hdfs fsck /path/to/hdfs/directory -files -blocks -locations -racks -replicaDetails > fsck_report.txt]
```
1. hadoop namenode -format
  2. hadoop secondarynamenode
  3. hadoop namenode
  4. hadoop datanode
  5. hadoop dfsadmin -report
  6. hadoop mradmin -refreshNodes
  7. hdfs dfsadmin -setBalancerBandwidth <bandwidth>
  8. hdfs fsck /user/example/data
  9. hdfs fsck /path/to/hdfs/directory -files -blocks -locations
  10. hdfs fsck /path/to/hdfs/directory -files -blocks -locations -racks -replicaDetails
  11. hdfs fsck /path/to/hdfs/directory -files -blocks -locations -racks -replicaDetails > fsck\_report.txt

## How to Interact with HDFS

1. hadoop mradmin -refreshNodes -decommission <hostname>
2. hadoop mradmin -refreshNodes -commission <hostname>

---

3. hdfs dfs -ls /path/to/directory http://<TaskTracker-Hostname>:50060
4. hdfs dfs -copyToLocal /path/in/hdfs localfile http://<JobTracker-Hostname>:50030
5. hdfs dfs -rm /path/to/file hadoop balancer -h
6. hdfs dfs -rm -r /user/hadoop/data
7. hdfs dfs -mv /path/to/source /path/to/destination
8. hdfs dfs -put <src path> <dest path>
9. Hdfs -get <hdfs file path> <local path>

# Interact with MapReduce Jobs

| Sr.No. | GENERIC_OPTION & Description                                                                                                                    |   |                                                                                                                                                                                                                                                                  |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>-submit &lt;job-file&gt;</b><br>Submits the job.                                                                                             | — | <b>-history [all] &lt;jobOutputDir&gt;</b> - history <jobOutputDir><br>Prints job details, failed and killed tip details. More details about the job such as successful tasks and task attempts made for each task can be viewed by specifying the [all] option. |
| 2      | <b>-status &lt;job-id&gt;</b><br>Prints the map and reduce completion percentage and all job counters.                                          | 6 | <b>-list[all]</b><br>Displays all jobs. -list displays only jobs which are yet to complete.                                                                                                                                                                      |
| 3      | <b>-counter &lt;job-id&gt; &lt;group-name&gt; &lt;countername&gt;</b><br>Prints the counter value.                                              | 7 | <b>-kill-task &lt;task-id&gt;</b><br>Kills the task. Killed tasks are NOT counted against failed attempts.                                                                                                                                                       |
| 4      | <b>-kill &lt;job-id&gt;</b><br>Kills the job.                                                                                                   | 8 | <b>-fail-task &lt;task-id&gt;</b><br>Fails the task. Failed tasks are counted against failed attempts.                                                                                                                                                           |
| 5      | <b>-events &lt;job-id&gt; &lt;fromevent-#&gt; &lt;#-of-events&gt;</b><br>Prints the events' details received by jobtracker for the given range. | 9 | <b>-set-priority &lt;job-id&gt; &lt;priority&gt;</b><br>Changes the priority of the job. Allowed priority values are VERY_HIGH, HIGH, NORMAL, LOW, VERY_LOW                                                                                                      |

# Yet Another Resource Negotiator (YARN)

---

- YARN stands for “[Yet Another Resource Negotiator](#)“. It was introduced in Hadoop 2.0 **to remove the bottleneck on Job Tracker which was present in Hadoop 1.0.**
- YARN was described as a “*Redesigned Resource Manager*” at the time of its launching
- YARN architecture basically **separates resource management layer from the processing layer.**
- YARN also allows different data processing engines like graph processing, interactive processing, stream processing as well as batch processing.
- it can **dynamically allocate various resources** and schedule the application processing

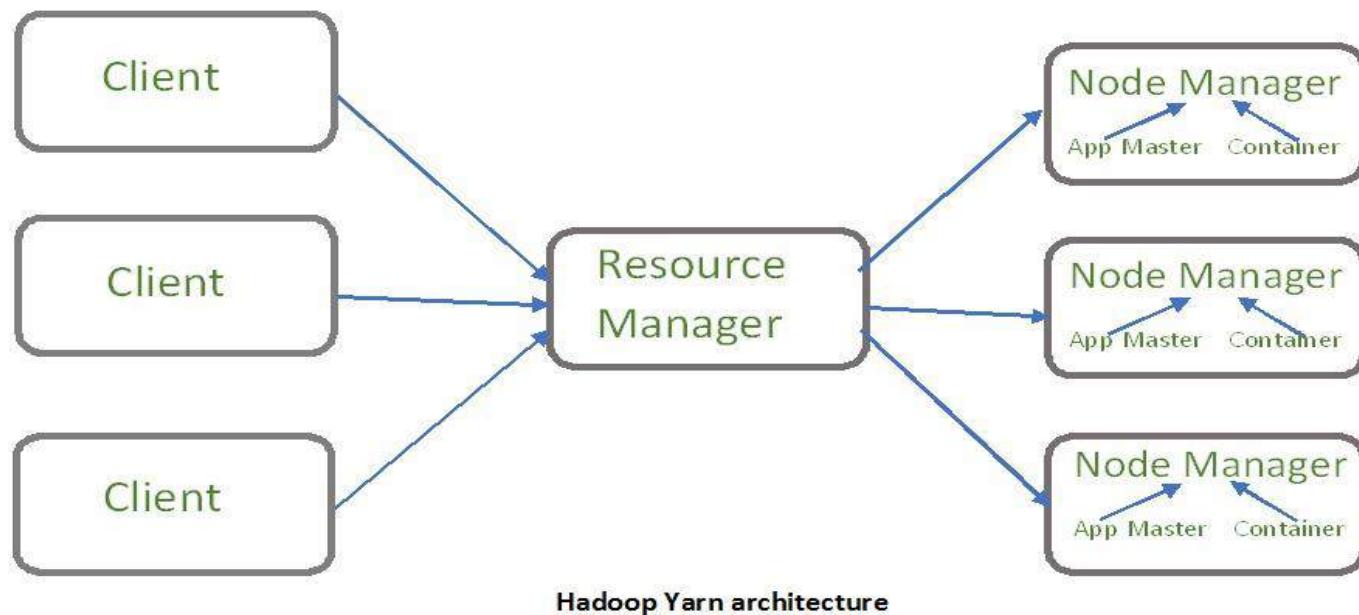
## YARN Features

---

- **Scalability:** The scheduler in Resource manager of YARN architecture allows Hadoop to **extend and manage thousands of nodes and clusters.**
- **Compatibility:** YARN supports the **existing map-reduce applications** without disruptions thus making it compatible with Hadoop 1.0 as well.
- **Cluster Utilization:** Since YARN supports Dynamic utilization of cluster in Hadoop, which enables optimized Cluster Utilization.
- **Multi-tenancy:** It allows **multiple engine** access thus giving organizations a benefit of multi-tenancy.

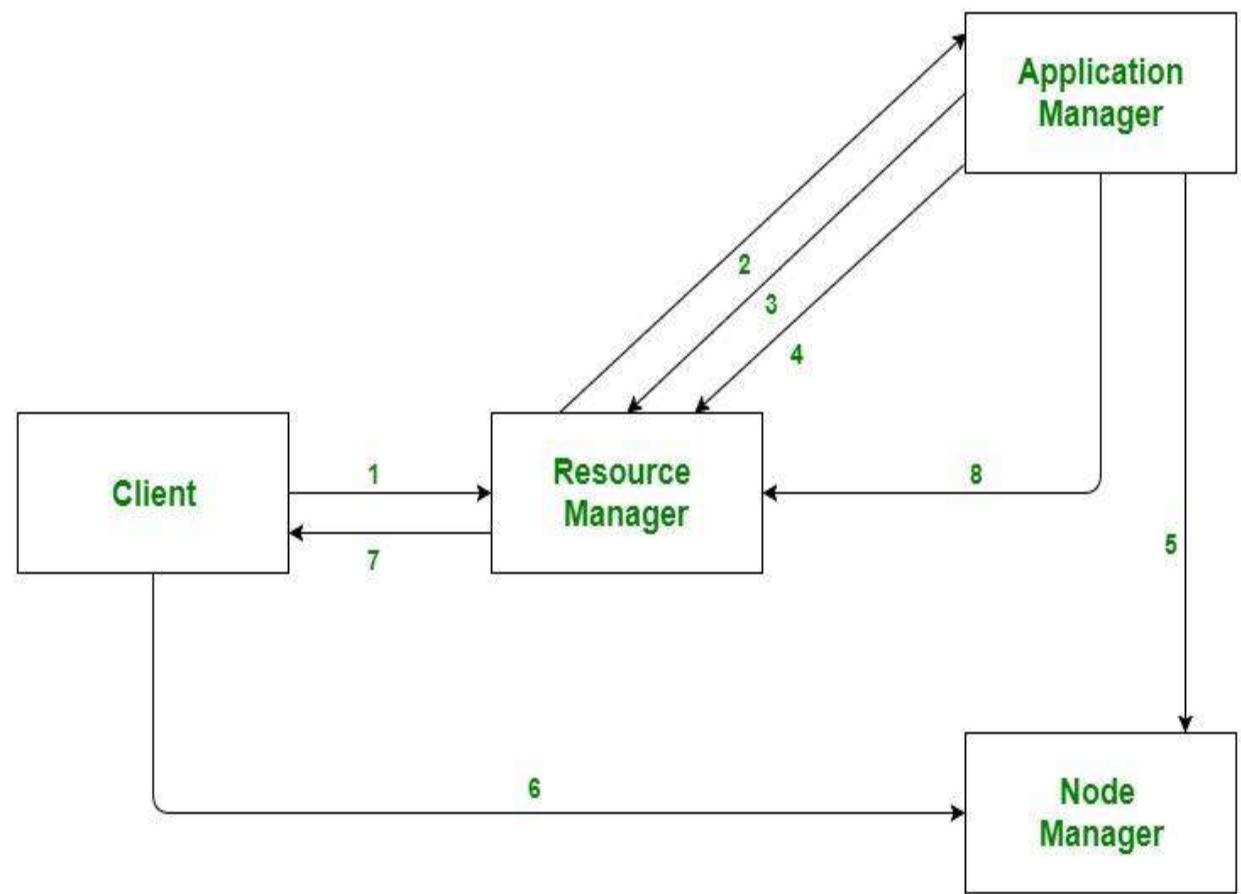
# The main components of YARN architecture

---



- Client
  - Resource Manager
- 1. Scheduler**
  - 2. Application manager**
- Node Manager
  - Application Master
  - Container

# Application workflow in Hadoop YARN

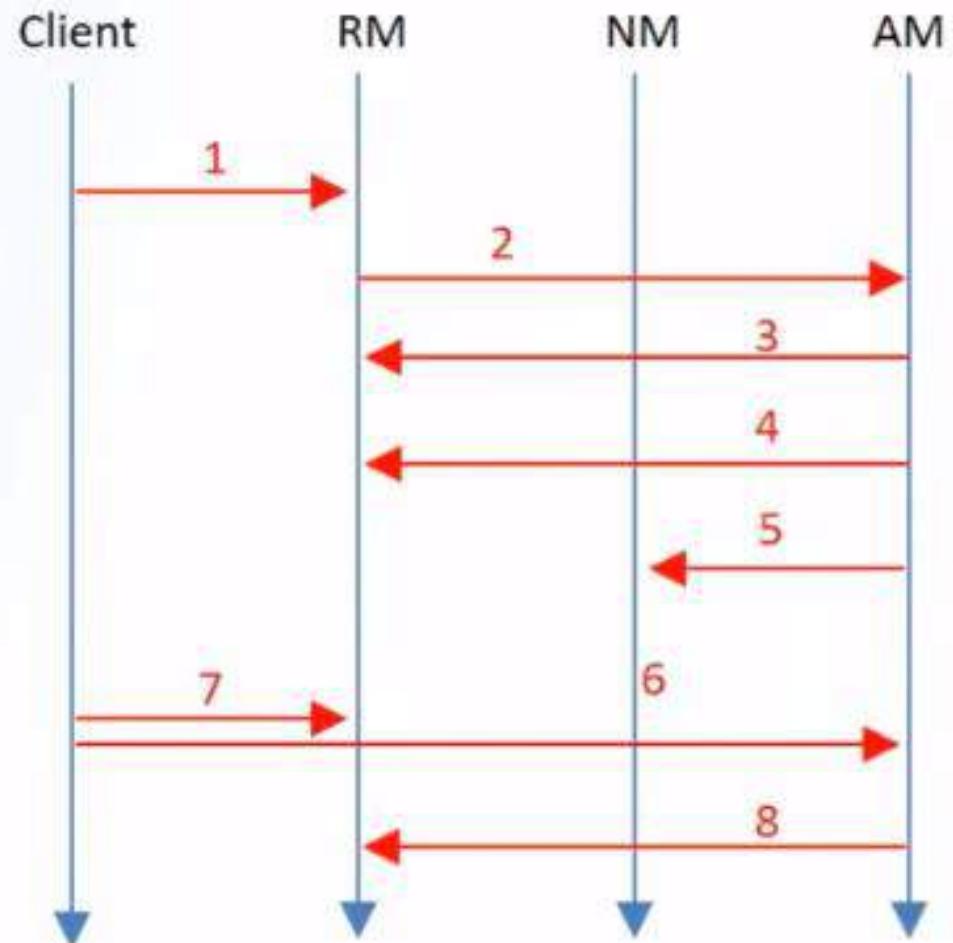


1. Client submits an application
2. The Resource Manager allocates a container to start the Application Manager
3. The Application Manager registers itself with the Resource Manager
4. The Application Manager negotiates containers from the Resource Manager
5. The Application Manager notifies the Node Manager to launch containers
6. Application code is executed in the container
7. Client contacts Resource Manager/Application Manager to monitor application's status
8. Once the processing is complete, the Application Manager un-registers with the Resource Manager

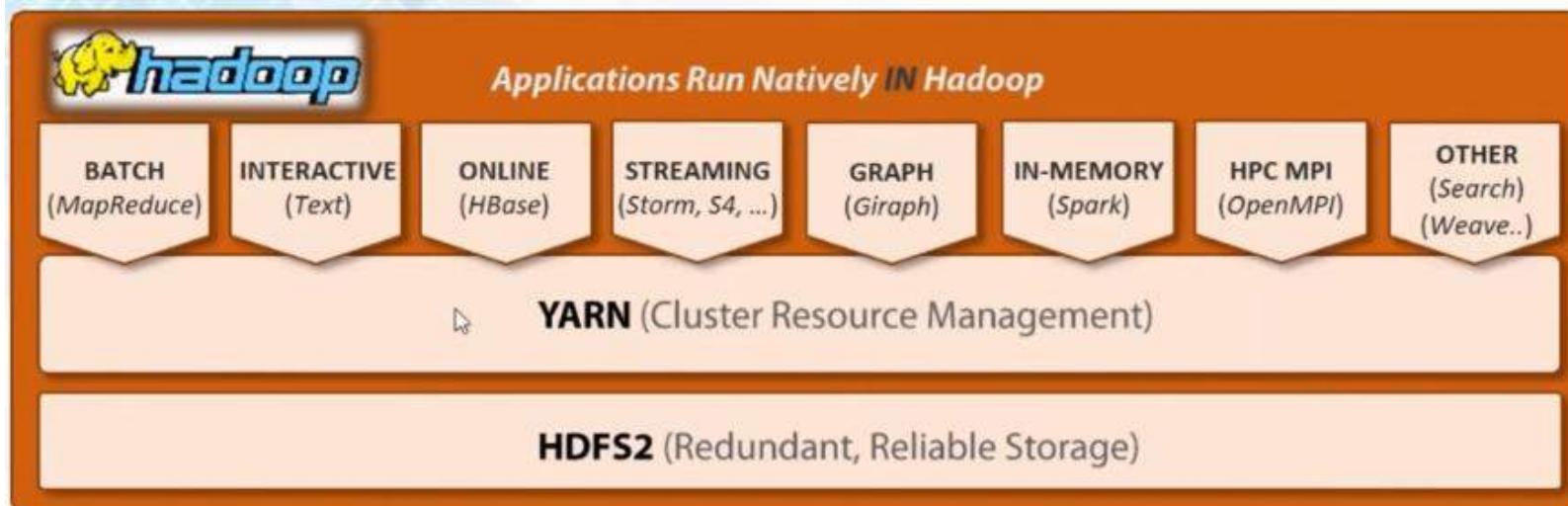
# Application : Workflow

→ Execution Sequence :

1. Client submits an application
2. RM allocates a container to start AM
3. AM registers with RM
4. AM asks containers from RM
5. AM notifies NM to launch containers
6. Application code is executed in container
7. Client contacts RM/AM to monitor application's status
8. AM unregisters with RM

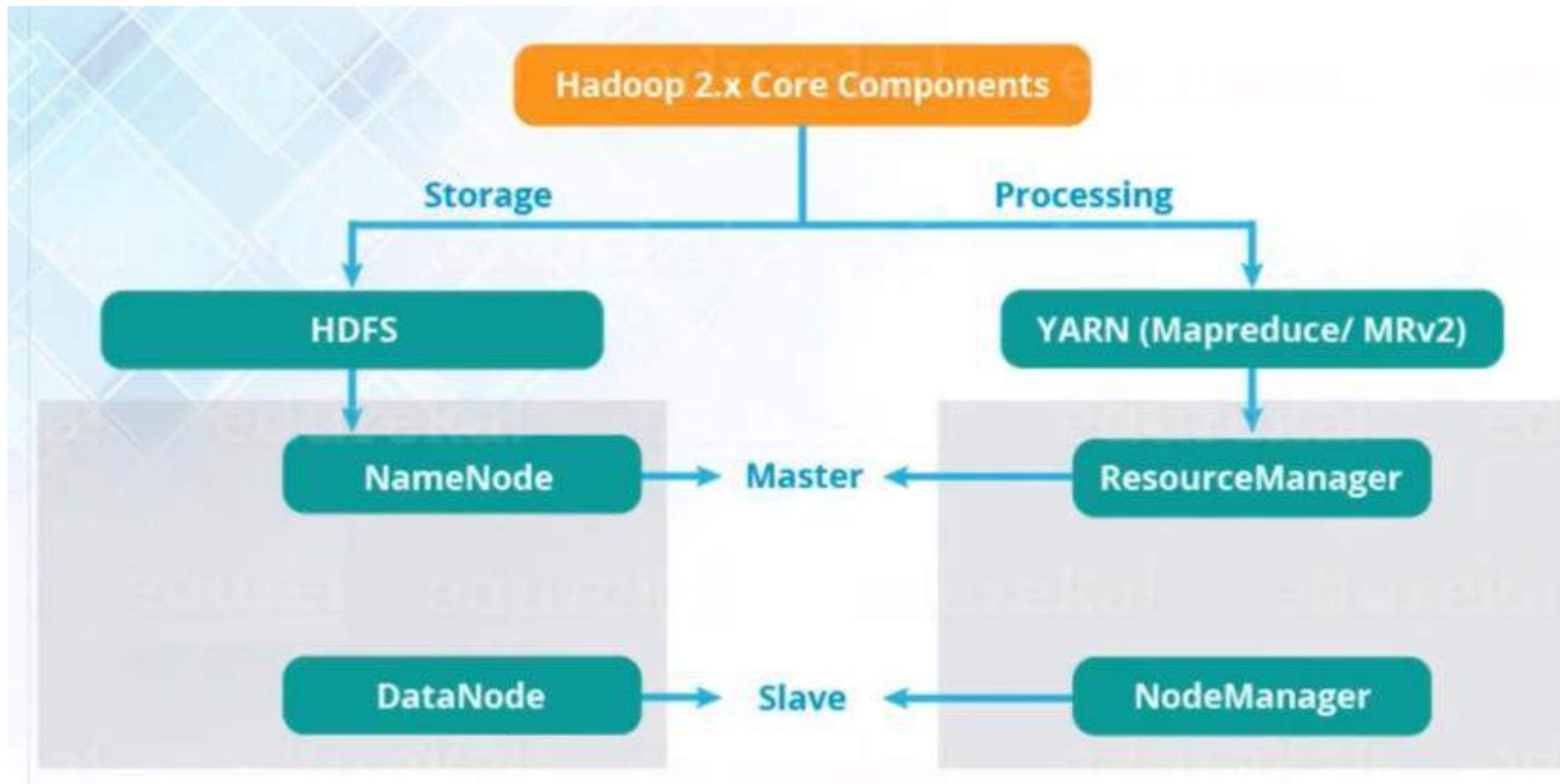


# YARN : MapReduce Beyond



- Using YARN lot of frameworks are able to utilize and connect to HDFS
- YARN – Open gate for frameworks, other search engines and even other big data applications also
- Application of “HDFS ” increased because of YARN as resource provider

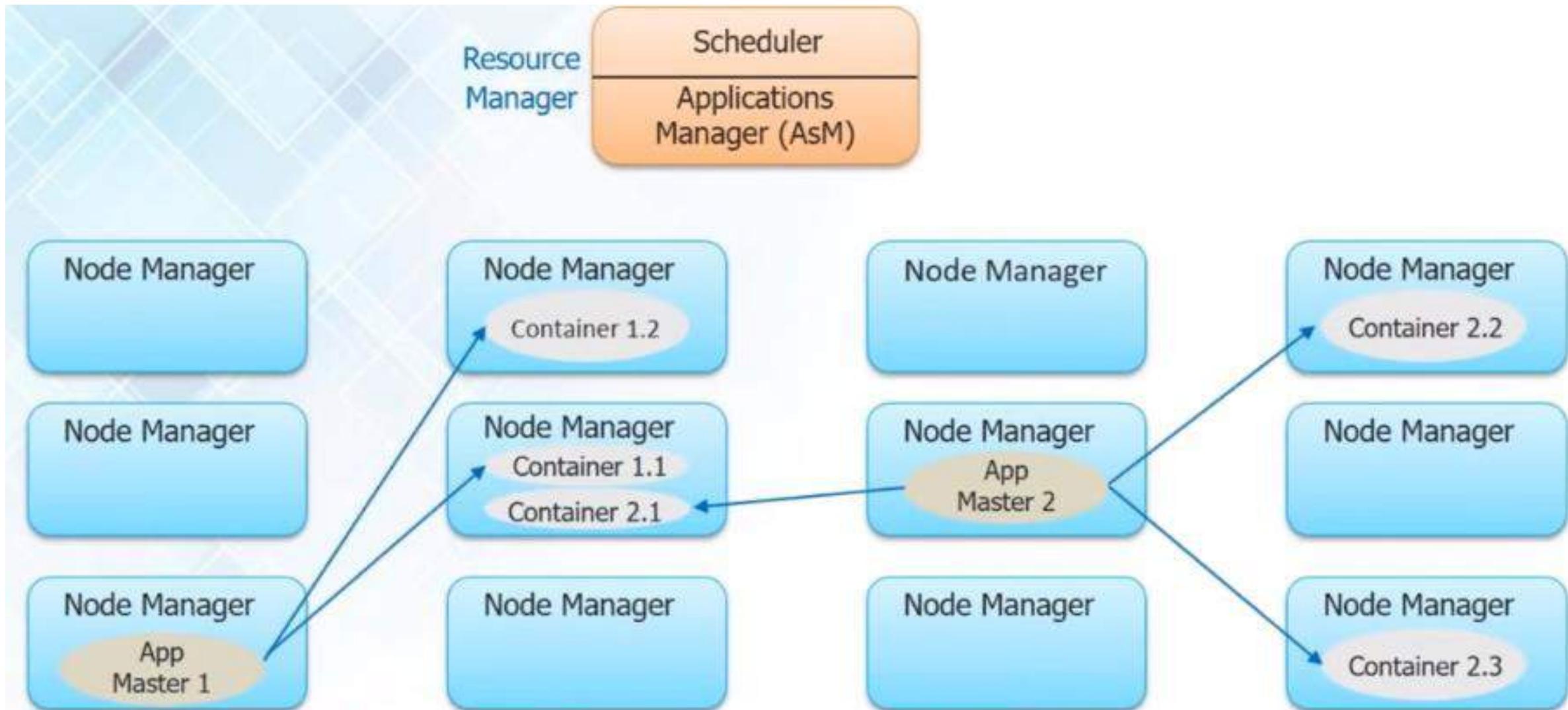
# Hadoop Daemons:



# Hadoop 2.X MapReduce Yarn Components:

- Client
  - » Submits a MapReduce Job
- Resource Manager
  - » Cluster Level resource manager
  - » Long Life, High Quality Hardware
- Node Manager
  - » One per Data Node
  - » Monitors resources on Data Node
- Job History Server
  - » Maintains information about submitted MapReduce jobs after their ApplicationMaster terminates
- ApplicationMaster
  - » One per application
  - » Short life
  - » Coordinates and Manages MapReduce Jobs
  - » Negotiates with Resource Manager to schedule tasks
  - » The tasks are started by NodeManager(s)
- Container
  - » Created by NM when requested
  - » Allocates certain amount of resources (memory, CPU etc.) on a slave node

# YARN: Workflow





THANK  
YOU.



# BIG DATA ANALYTICS

Part – 4

“PIG”

By: Dr.Rashmi L Malghan &  
Ms. Shavantrevva S Bilakeri

# Agenda:

- Entry of Apache Pig
- Pig vs MapReduce
- Twitter Case Study on Apache Pig
- Apache Pig Architecture
- Pig Components
- Pig Data Model & Operators
- Running Pig Commands and Pig Scripts (Log Analysis)

# Apache PIG

- Pig was introduced by yahoo and later on it was made fully open source by Apache Hadoop.
- It also provides a bridge to query data over Hadoop clusters but unlike hive, it implements a script implementation to make Hadoop data accessable by developers and business persons.
- Apache pig provides a high level programming platform for developers to process and analyses Big Data using user defined functions and programming efforts.
- In January 2013 Apache released Pig 0.10.1 which is defined for use with Hadoop 0.10.1 or later releases.

# MapReduce Way:

In MapReduce, you need to write a program in Java/Python to process the data.

# Apache PIG:

- Focus on the **data transformations** rather than the underlying MapReduce implementation.
- Apache Pig's **high-level dataflow engine** simplifies the development of large-scale data processing tasks on Hadoop clusters by providing an abstraction layer and leveraging the power of MapReduce **without requiring users to write complex Java code**.

# Key Features and example of Pig Latin code

- Declarative Language ([Pig Latin](#))
- Abstraction from MapReduce
- Data Flow Model
- Schema Flexibility
- Optimization Opportunities
- Ease of Use

-- Load data

```
data = LOAD 'input_data' USING PigStorage(',');
```

-- Filter data

```
filtered_data = FILTER data BY $1 > 50;
```

-- Group and aggregate

```
grouped_data = GROUP filtered_data BY $0;
result = FOREACH grouped_data GENERATE
group, AVG(filtered_data.$1);
```

-- Store the result

```
STORE result INTO 'output';
```

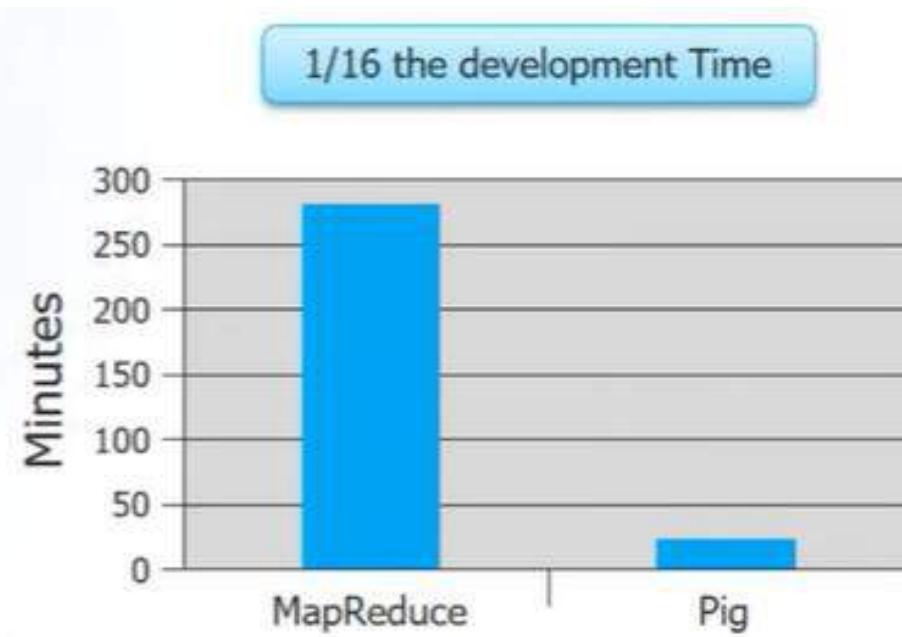
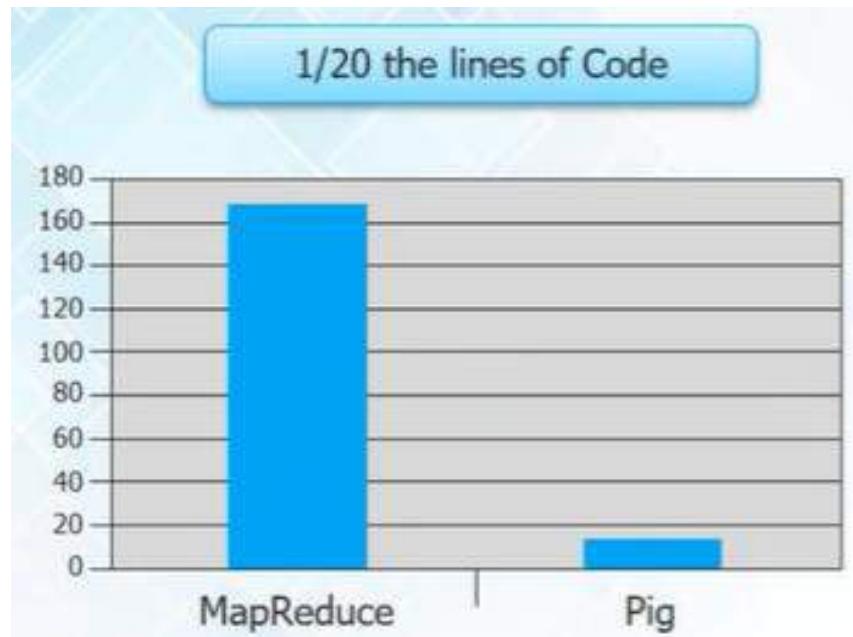
# Apache PIG:

- An open-source high-level dataflow system
- Introduced by Yahoo
- Provides abstraction over MapReduce
- Two main components – the *Pig Latin* language and the *Pig Execution*

## Fun Fact:

- ✓ 10 lines of pig latin= approx. 200 lines of Map-Reduce Java Program

# Why to Opt Pig instead of MapReduce:



# PIG VS MapReduce



- High-level data flow tool
- No need to write complex programs
- Built-in support for data operations like joins, filters, ordering, sorting etc.
- Provides nested data types like tuples, bags, and maps



- Low-level data processing paradigm
- You need write programs in Java/Python etc.
- Performing data operations in MapReduce is a humongous task
- Nested data types are not there in MapReduce

# Apache Pig: Advantages

→ Can take any data

- Structured data
- Semi-Structured data
- Unstructured data

→ Easy to learn, Easy to write and Easy to read

- Data Flow Language
- Reads like a series of steps

→ Extensible by UDF (User Defined Functions)

- Java
- Python
- JavaScript
- Ruby

→ Provides common data operations **filters**, **joins**, **ordering**, etc. and nested data types **tuples**, **bags**, and **maps** missing from MapReduce.

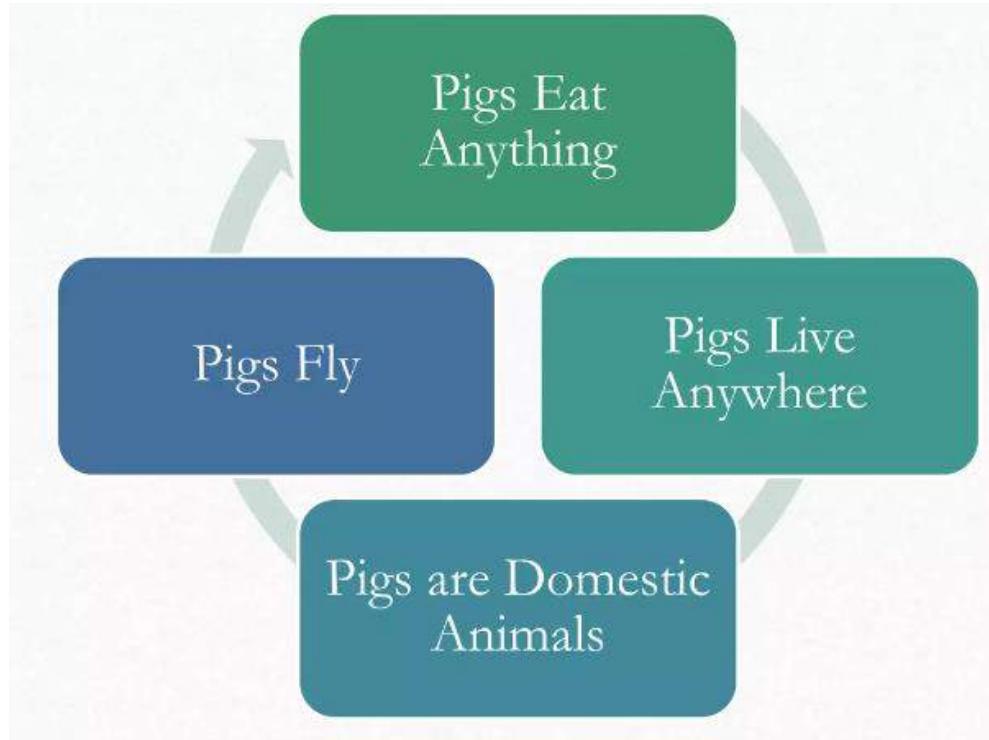
→ An **ad-hoc** way of creating and executing map-reduce jobs on very large data sets

→ Open source and actively supported by a community of developers.

# PIG Anatomy

1. Data flow Language- **pig latin**
2. Interactive shell- **Grunt**
3. Prig interpreter and execution engine

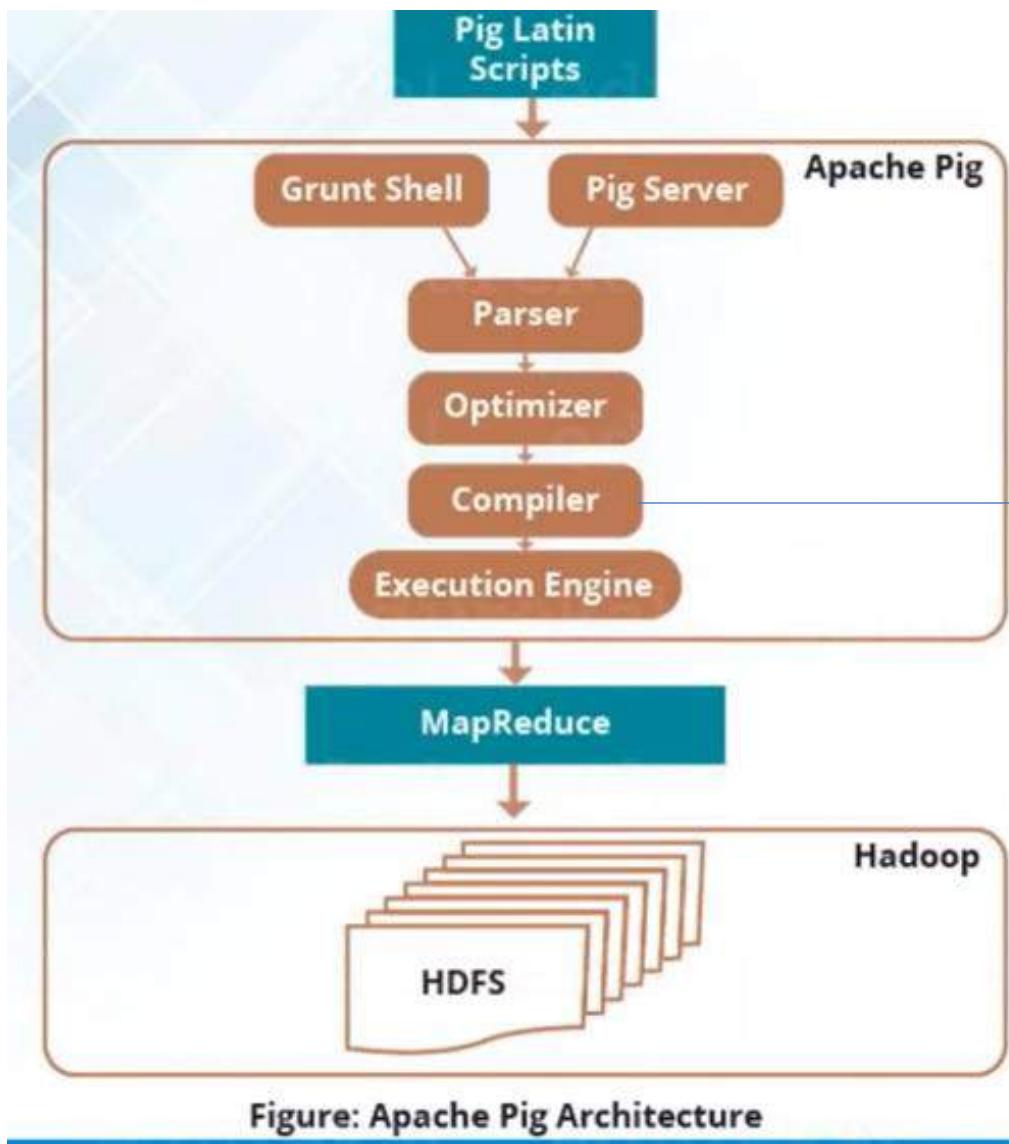
## PIG Philosophy



**PIG supports:**

1. HDFS commands
2. UNIX shell operators
3. Relational operators
4. Positional operators
5. Mathematical functions
6. User defined functions
7. Complex data structures

# Apache Pig – Architecture / Pig MapReduce Engine



- **Pig Latin Scripts:** Execute queries over big data.
- **Grunt Shell:** Native shell provided by Apache Pig, to execute pig queries.
- Submit pig scripts to **java client** to pig server & execute over Apache pig.
- **Compiler:** Converts pig latin scripts to Apache MapReduce code
- Executed using executing engine over Hadoop cluster

## Operators in Apache Pig

Pig Latin Operators are the basic constructs that allow **data manipulation** in Apache Pig. Some commonly used operators include:

- **LOAD** and **STORE**: These operators are used to **read and write** data.
- **FILTER**: The FILTER operator is used to **remove unwanted** data based on a condition.
- **GROUP**: The GROUP operator is used to **group the data in one or more relations**.
- **JOIN**: The JOIN operator **merges two or more relations**.
- **SPLIT**: The SPLIT operator is used to **split a single relation into two or more relations** based on some condition.

# Pig Syntax used for Data Processing

X= LOAD 'file name.txt'; #directory name

.....

Y= GROUP ....;

.....

Z= FILTER ...;

....

DUMP Y; #view result on screen

.....

STORE Z into 'temp'

## Example Latin Script: find the total distance travelled by a flight

-- Load the flight data

```
flight_data = LOAD 'path/to/flight_data' USING PigStorage(',') AS
(date:chararray, distance:int);
```

-- Filter out empty or invalid distance values

```
filtered_data = FILTER flight_data BY distance is not null and distance >= 0;
```

-- Calculate the total distance covered

```
total_distance = FOREACH (GROUP filtered_data ALL) GENERATE
SUM(filtered_data.distance) as total_distance;
```

-- Display the result

```
DUMP total_distance;
```

**Question:** load the student data (assuming data contains rollno, name, gpa),  
remove the students whose gpa is less than 5.0.

From the filtered data, find the student name with highest gpa.

Display and Store the result to output file

## SOLUTION

```
A = load 'student' (rollno, name, gpa) #A is a relational table not a variable
```

```
A = filter A by gpa>5.0
```

```
A = foreach A generate Upper (name);
```

```
STORE A INTO 'myreport'
```

**Note:** by default columns are indexed with \$01, \$02, \$03 when **USING PigStorage** is not used.

--Load student data

```
student_data = LOAD 'path/to/student_data' USING PigStorage(',')
AS (rollno:chararray, name:chararray, gpa:float);
```

-- Filter out students with GPA less than 5.0

```
filtered_data = FILTER student_data BY gpa >= 5.0;
```

-- Find the student with the highest GPA

```
max_gpa_student = ORDER filtered_data BY gpa DESC;
top_student = LIMIT max_gpa_student 1;
```

-- Display the result

```
DUMP top_student;
```

-- Store the result in an output file

```
STORE top_student INTO 'path/to/output' USING PigStorage(',');
```

## PIG LATIN IDENTIFIERS and COMMENTS

- Identifiers are the **names assigned to field or the other data structures**
- Should **begin with a letter** and should be followed only by letters and underscores
- Examples for valid: **Y, A1, A1\_2014, Sample**
- Examples for invalid: **5, sales\$, sales%, \_sales**
- Single line comment begin with “**--**”
- Multiline comment begin with “**/\*** and end with **\*/**”

# Case Study: Twitter

- **Objective:** To increase their user base / Enhance their offerings
- **Procedures:** To Extract insights monthly/weekly/daily
- **Results:** To scale up their infrastructure so that they will be able to handle larger user base they are targeting.

# Case Study: Twitter

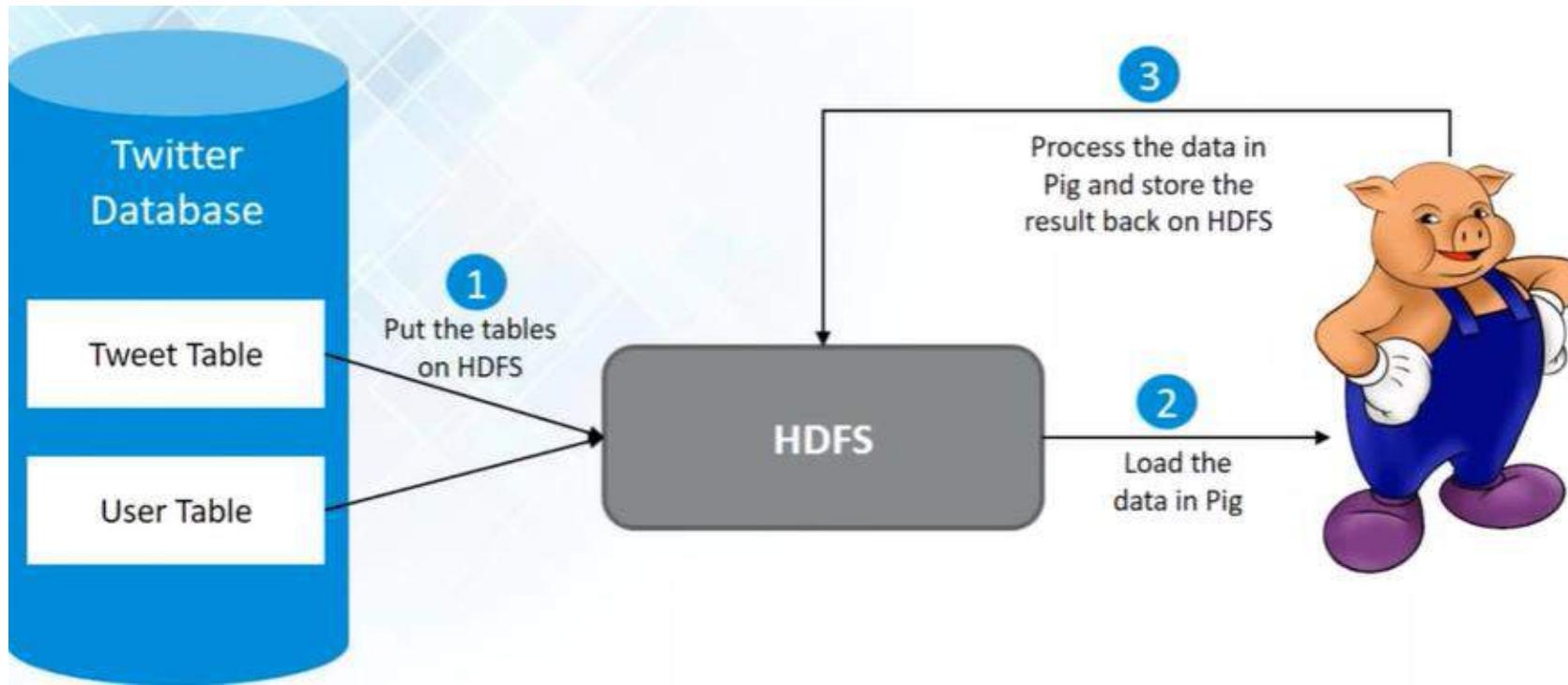


- Twitter's data was growing at an accelerating rate (i.e. 10 TB/day).
- Thus, Twitter decided to move the archived data to HDFS and adopt Hadoop for extracting the business values out of it.
- Their major aim was to analyse data stored in Hadoop to come up with the multiple insights on a daily, weekly or monthly basis.

Let me talk about one of the insight they wanted to know.

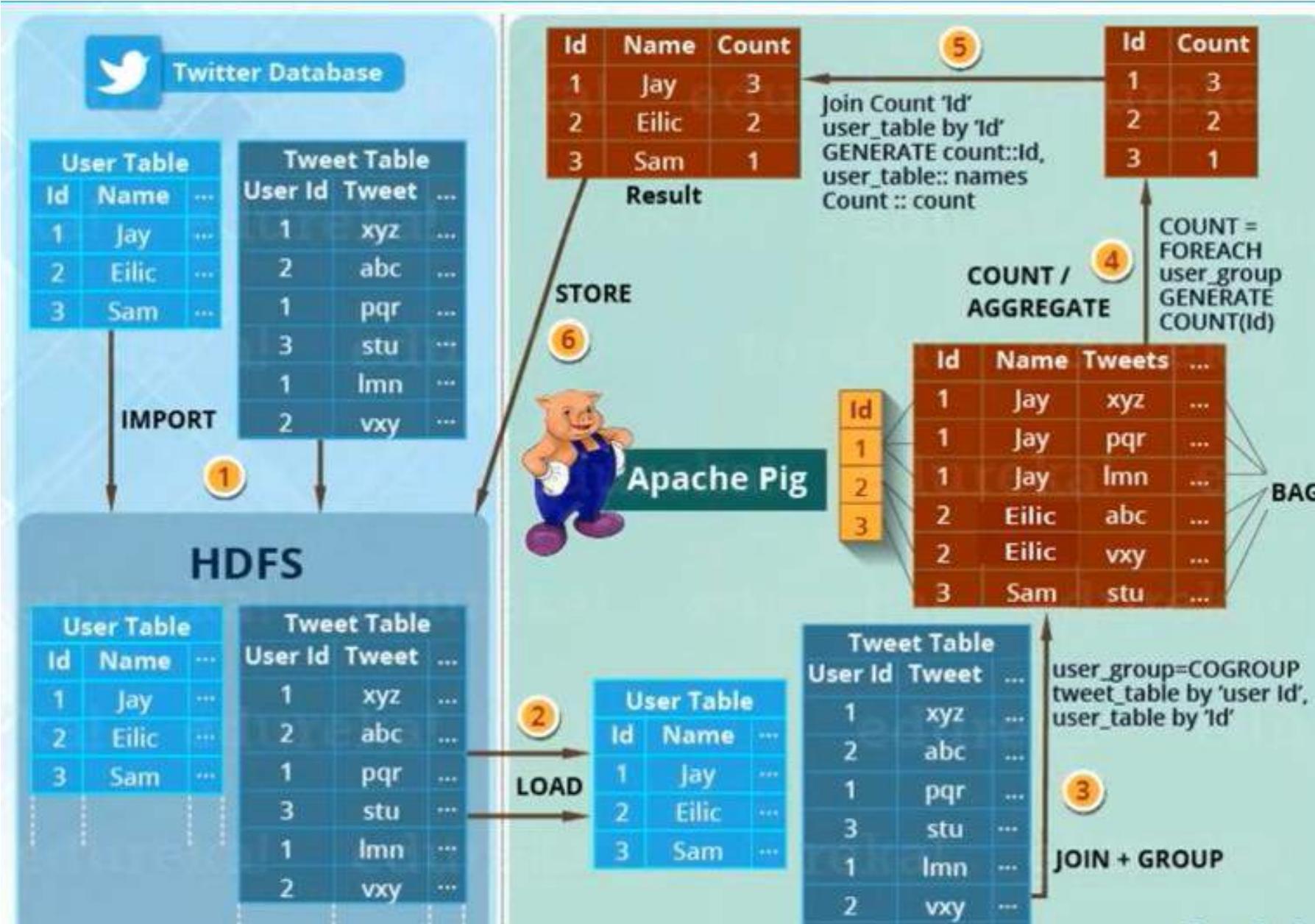
*Analyzing how many tweets are stored per user, in the given tweet tables?*

# High Level implementation – HDFS & Pig



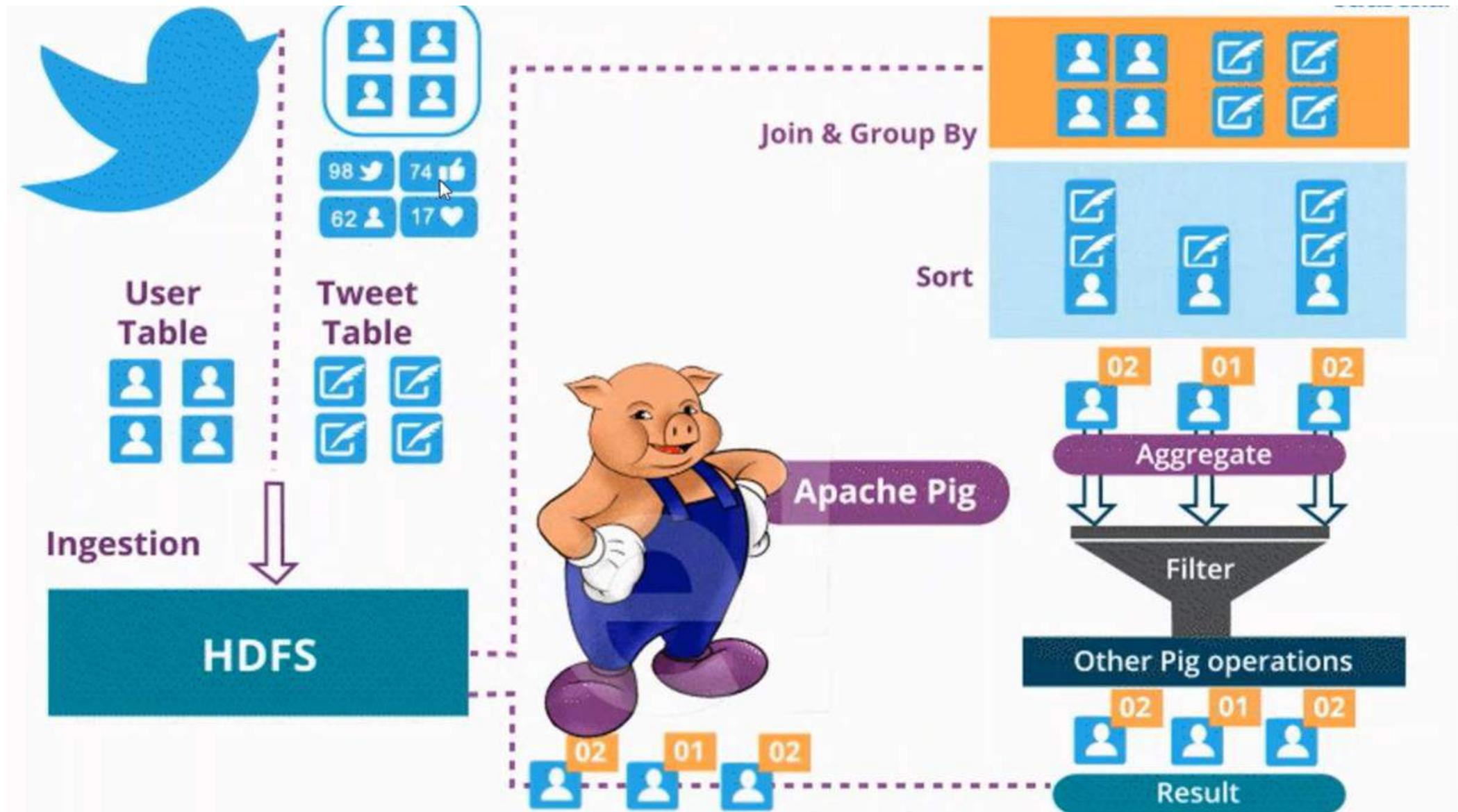
- Twitter Database had many tables, in which archive data was stored.
- The insight they want to extract was related to :Tweet & user table.

# Implementation Flow - Detail



- **Import -Sqoop-**  
transfer data between RDBM to HDFS or HDFS to RDMS
- Twitter used Pig instead of mapreduce thus –**saved their time and effort.**

# Case Study: Twitter



# What happens underneath the covers when you run/submit a Sqoop import job

- Sqoop will connect to the database.
- Sqoop uses JDBC to examine the table by retrieving a list of all the columns and their SQL data types. These SQL types (varchar, integer and more) can then be mapped to Java data types (String, Integer etc.)
- Sqoop's code generator will use this information to create a table-specific class to hold records extracted from the table.
- Sqoop will connect to cluster and submit a MapReduce job.
- The dataset being transferred is sliced up into different partitions and a map-only job is launched with individual mappers responsible for transferring a slice of this dataset.
- For databases, Sqoop will read the table row-by-row into HDFS.
- For mainframe datasets, sqoop will read records from each mainframe dataset into HDFS.
- The output of this import process is a set of files containing a copy of imported table or datasets.
- The import process is performed in parallel for this reason, the output will be in multiple files.
- These files may be delimited text files CSV, TSV or binary Avro or Sequence files containing serialized record data. By default it is CSV.

# Pig Latin: Case Sensitivity

- Keywords/operators are not case sensitive.

Ex: LOAD, STORE, GROUP, FOREACH DUMP

- Relations and paths are case sensitive
- Function names are case sensitive Ex:

PigStorage, COUNT

| Complex Types |       |                                                                                     |
|---------------|-------|-------------------------------------------------------------------------------------|
| 11            | Tuple | A tuple is an ordered set of fields.<br><b>Example :</b> (raja, 30)                 |
| 12            | Bag   | A bag is a collection of tuples.<br><b>Example :</b> {(raju,30),(Mohammad,45)}      |
| 13            | Map   | A Map is a set of key-value pairs.<br><b>Example :</b> [ 'name' #'Raju', 'age' #30] |

| S.N. | Data Type  | Description & Example                                                                                |
|------|------------|------------------------------------------------------------------------------------------------------|
| 1    | int        | Represents a signed 32-bit integer.<br><b>Example :</b> 8                                            |
| 2    | long       | Represents a signed 64-bit integer.<br><b>Example :</b> 5L                                           |
| 3    | float      | Represents a signed 32-bit floating point.<br><b>Example :</b> 5.5F                                  |
| 4    | double     | Represents a 64-bit floating point.<br><b>Example :</b> 10.5                                         |
| 5    | chararray  | Represents a character array (string) in Unicode UTF-8 format.<br><b>Example :</b> 'tutorials point' |
| 6    | Bytearray  | Represents a Byte array (blob).                                                                      |
| 7    | Boolean    | Represents a Boolean value.<br><b>Example :</b> true/ false.                                         |
| 8    | Datetime   | Represents a date-time.<br><b>Example :</b> 1970-01-01T00:00:00.000+00:00                            |
| 9    | Biginteger | Represents a Java BigInteger.<br><b>Example :</b> 60708090709                                        |
| 10   | BigDecimal | Represents a Java BigDecimal<br><b>Example :</b> 185.98376256272893883                               |

# Pig Latin – Arithmetic Operators

| Operator                         | Description                                                                                                                                                                   | Example                                                                                  |
|----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| +                                | <b>Addition</b> – Adds values on either side of the operator                                                                                                                  | a + b will give 30                                                                       |
| -                                | <b>Subtraction</b> – Subtracts right hand operand from left hand operand                                                                                                      | a - b will give -10                                                                      |
| *                                | <b>Multiplication</b> – Multiplies values on either side of the operator                                                                                                      | a * b will give 200                                                                      |
| /                                | <b>Division</b> – Divides left hand operand by right hand operand                                                                                                             | b / a will give 2                                                                        |
| %                                | <b>Modulus</b> – Divides left hand operand by right hand operand and returns remainder                                                                                        | b % a will give 0                                                                        |
| ? :                              | <b>Bincond</b> – Evaluates the Boolean operators. It has three operands as shown below.<br>variable <b>x</b> = (expression) ? <b>value1</b> if true : <b>value2</b> if false. | b = (a == 1)? 20: 30;<br>if a = 1 the value of b is 20.<br>if a!=1 the value of b is 30. |
| CASE<br>WHEN<br>THEN<br>ELSE END | <b>Case</b> – The case operator is equivalent to nested bincond operator.                                                                                                     | CASE f2 % 2<br>WHEN 0 THEN<br>'even'<br>WHEN 1 THEN 'odd'<br>END                         |

# Pig Latin – Comparison Operators

| Operator             | Description                                                                                                                                                                    | Example                                |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|
| <code>==</code>      | <b>Equal</b> – Checks if the values of two operands are equal or not; if yes, then the condition becomes true.                                                                 | $(a == b)$ is not true                 |
| <code>!=</code>      | <b>Not Equal</b> – Checks if the values of two operands are equal or not. If the values are not equal, then condition becomes true.                                            | $(a != b)$ is true.                    |
| <code>&gt;</code>    | <b>Greater than</b> – Checks if the value of the left operand is greater than the value of the right operand. If yes, then the condition becomes true.                         | $(a > b)$ is not true.                 |
| <code>&lt;</code>    | <b>Less than</b> – Checks if the value of the left operand is less than the value of the right operand. If yes, then the condition becomes true.                               | $(a < b)$ is true.                     |
| <code>&gt;=</code>   | <b>Greater than or equal to</b> – Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition becomes true. | $(a >= b)$ is not true.                |
| <code>&lt;=</code>   | <b>Less than or equal to</b> – Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true.       | $(a <= b)$ is true.                    |
| <code>matches</code> | <b>Pattern matching</b> – Checks whether the string in the left-hand side matches with the constant in the right-hand side.                                                    | <code>f1 matches '.*tutorial.*'</code> |

# Pig Latin – Relational Operations

| Operator                    | Description                                                         |
|-----------------------------|---------------------------------------------------------------------|
| <b>Loading and Storing</b>  |                                                                     |
| LOAD                        | To Load the data from the file system (local/HDFS) into a relation. |
| STORE                       | To save a relation to the file system (local/HDFS).                 |
| <b>Filtering</b>            |                                                                     |
| FILTER                      | To remove unwanted rows from a relation.                            |
| DISTINCT                    | To remove duplicate rows from a relation.                           |
| FOREACH, GENERATE           | To generate data transformations based on columns of data.          |
| STREAM                      | To transform a relation using an external program.                  |
| <b>Grouping and Joining</b> |                                                                     |
| JOIN                        | To join two or more relations.                                      |
| COGROUP                     | To group the data in two or more relations.                         |
| GROUP                       | To group the data in a single relation.                             |
| CROSS                       | To create the cross product of two or more relations.               |

# Pig Latin – Relational Operations

## Sorting

|       |                                                                                                |
|-------|------------------------------------------------------------------------------------------------|
| ORDER | To arrange a relation in a sorted order based on one or more fields (ascending or descending). |
| LIMIT | To get a limited number of tuples from a relation.                                             |

## Combining and Splitting

|       |                                                          |
|-------|----------------------------------------------------------|
| UNION | To combine two or more relations into a single relation. |
| SPLIT | To split a single relation into two or more relations.   |

## Diagnostic Operators

|            |                                                                                    |
|------------|------------------------------------------------------------------------------------|
| DUMP       | To print the contents of a relation on the console.                                |
| DESCRIBE   | To describe the schema of a relation.                                              |
| EXPLAIN    | To view the logical, physical, or MapReduce execution plans to compute a relation. |
| ILLUSTRATE | To view the step-by-step execution of a series of statements.                      |

# Pig Latin – Type Construction Operators

| Operator | Description                                                                     | Example                      |
|----------|---------------------------------------------------------------------------------|------------------------------|
| ()       | <b>Tuple constructor operator</b> – This operator is used to construct a tuple. | (Raju, 30)                   |
| {}       | <b>Bag constructor operator</b> – This operator is used to construct a bag.     | {(Raju, 30), (Mohammad, 45)} |
| []       | <b>Map constructor operator</b> – This operator is used to construct a tuple.   | [name#Raja, age#30]          |

# Apache Pig - Diagnostic Operators

To verify the execution of the **Load** statement, you have to use the **Diagnostic Operators**.

1. Dump operator
2. Describe operator
3. Explanation operator
4. Illustration operator

## Dump Operator

The **Dump** operator is used to run the Pig Latin statements and display the results on the screen. It is generally used for debugging Purpose.

syntax of the Dump operator:

*grunt> Dump Relation\_Name*

## Example

Assume we have a file **student\_data.txt** in HDFS with the following content.

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

And we have read it into a relation **student** using the LOAD operator as shown below.

```
grunt> student = LOAD 'hdfs://localhost:9000/pig_data/student_data.txt'
 USING PigStorage(',')
 as (id:int, firstname:chararray, lastname:chararray, phone:chararray,
 city:chararray);
```

Now, let us print the contents of the relation using the **Dump operator** as shown below.

```
grunt> Dump student
```

2015-10-01 15:05:27,642 [main]

INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLaun  
100% complete

2015-10-01 15:05:27,652 [main]

INFO org.apache.pig.tools.pigstats.mapreduce.SimplePigStats - Script Statistics:

| HadoopVersion | PigVersion | UserId | StartedAt           | FinishedAt       | Features |
|---------------|------------|--------|---------------------|------------------|----------|
| 2.6.0         | 0.15.0     | Hadoop | 2015-10-01 15:03:11 | 2015-10-01 05:27 | UNKNOWN  |

Success!

Job Stats (time in seconds):

JobId job\_14459\_0004

Maps 1

Reduces 0

MaxMapTime n/a

MinMapTime n/a

AvgMapTime n/a

MedianMapTime n/a

MaxReduceTime 0

MinReduceTime 0

AvgReduceTime 0

MedianReducetime 0

Alias student

Feature MAP\_ONLY

Outputs hdfs://localhost:9000/tmp/temp580182027/tmp757878456,

2015-10-01 15:06:28,485 [main]

INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths  
to process : 1

2015-10-01 15:06:28,485 [main]

INFO org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input path  
to process : 1

(1,Rajiv,Reddy,9848022337,Hyderabad)

(2,siddarth,Battacharya,9848022338,Kolkata)

(3,Rajesh,Khanna,9848022339,Delhi)

(4,Preethi,Agarwal,9848022330,Pune)

(5,Trupthi,Mohanthy,9848022336,Bhuwaneshwar)

(6,Archana,Mishra,9848022335,Chennai)

# Describe operator

- The **describe** operator is used to **view the schema** of a relation.

## Syntax

```
grunt> Describe Relation_name
```

## Example

```
grunt> describe student;
```

```
grunt> student: { id: int,firstname: chararray,lastname: chararray,phone:
chararray,city: chararray }
```

# ILLUSTRATE OPERATOR

The **illustrate** operator gives you the step-by-step execution of a sequence of statements.

## Syntax

```
grunt> illustrate Relation_name;
```

```
grunt> illustrate student;
```

```
INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.PigMapOnly$M a
being processed per job phase (AliasName[line,offset]): M: student[1,10] C: R:

```

```
|student | id:int | firstname:chararray | lastname:chararray | phone:chararray | city:char

```

```
| | 002 | siddarth | Battacharya | 9848022338 | Kolkata |

```

# GROUP operator

The **GROUP** operator is used to group the data in one or more relations. It collects the data having the same key.

## Syntax

```
grunt> Group_data = GROUP Relation_name BY age;
```

## Example

```
grunt> group_data = GROUP student_details by age;
```

```
(21,{(4,Preethi,Agarwal,21,9848022330,Pune),(1,Rajiv,Reddy,21,9848022337,Hydera bad)})
(22,{(3,Rajesh,Khanna,22,9848022339,Delhi),(2,siddarth,Battacharya,22,9848022338,Kolkata)})
(23,{(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,9848022336 ,Bhuwaneshwar)})
(24,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,9848022334, trivendram)})
```

- The **COGROUP** operator works more or less in the same way as the [GROUP](#) operator.
- The only difference between the two operators is that the **group** operator is normally used with one relation, while the **cogroup** operator is used in statements involving two or more relations.

## Grouping Two Relations using Cogroup

Assume that we have two files namely **student\_details.txt** and **employee\_details.txt** in the HDFS directory **/pig\_data/** .

```
grunt> cogroup_data = COGROUP student_details by age, employee_details by
age;
```

**student\_details.txt**

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivedram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

**employee\_details.txt**

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuwaneshwar
006,Maggy,22,Chennai
```

The **UNION operator** of Pig Latin is used to merge the content of two relations.

To perform UNION operation on two relations, their columns and domains must be identical.

**Syntax:** `grunt> Relation_name3 = UNION Relation_name1, Relation_name2;`

#### **Student\_data1.txt**

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

#### **Student\_data2.txt**

```
7,Komal,Nayak,9848022334,trivendram.
8,Bharathi,Nambiayar,9848022333,Chennai.
```

```
grunt> student = UNION student1,
student2;
```

#### **Output**

```
(1,Rajiv,Reddy,9848022337,Hyderabad) (2,siddarth,Battacharya,9848022338,Kolkata)
(3,Rajesh,Khanna,9848022339,Delhi)
(4,Preethi,Agarwal,9848022330,Pune)
(5,Trupthi,Mohanthy,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,9848022335,Chennai)
(7,Komal,Nayak,9848022334,trivendram)
(8,Bharathi,Nambiayar,9848022333,Chennai)
```

The **SPLIT** operator is used to split a relation into two or more relations.

**Syntax:** `grunt> SPLIT Relation1_name INTO Relation2_name IF (condition1), Relation2_name (condition2)`

**student\_details.txt**

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

**Output**

```
grunt> Dump student_details1;
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(4,Preethi,Agarwal,21,9848022330,Pune)

grunt> Dump student_details2;
(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,23,9848022335,Chennai)
(7,Komal,Nayak,24,9848022334,trivendram)
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
```

```
SPLIT student_details into student_details1 if
age<23, student_details2 if (22<age and age>25);
```

The **FILTER operator** is used to select the required tuples from a relation based on a condition.

### Syntax

```
grunt> Relation2_name = FILTER Relation1_name BY (condition);
```

**student\_details.txt**

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

### Output

```
(6,Archana,Mishra,23,9848022335,Chennai)
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
```

```
filter_data = FILTER student_details BY
city == 'Chennai';
```

The **DISTINCT** operator is used to remove redundant (duplicate) tuples from a relation.

**Syntax:** grunt> Relation\_name2 = DISTINCT Relatin\_name1;

The **FOREACH** operator is used to generate specified data transformations based on the column data.

**Syntax:** grunt> Relation\_name2 = FOREACH Relatin\_name1 GENERATE (required data);

#### student\_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

```
grunt> foreach_data = FOREACH
student_details GENERATE id,age,city;
```

#### Output

```
(1,21,Hyderabad)
(2,22,Kolkata)
(3,22,Delhi)
(4,21,Pune)
(5,23,Bhuwaneshwar)
(6,23,Chennai)
(7,24,trivendram)
(8,24,Chennai)
```

The **ORDER BY operator** is used to display the contents of a relation in a sorted order based on one or more fields.

### Syntax

```
grunt> Relation_name2 = ORDER Relation_name1 BY (ASC|DESC);
```

**student\_details.txt**

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

```
grunt> order_by_data = ORDER
student_details BY age DESC;
```

### Output

```
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
(7,Komal,Nayak,24,9848022334,trivendram)
(6,Archana,Mishra,23,9848022335,Chennai)
(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(4,Preethi,Agarwal,21,9848022330,Pune)
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
```

The **LIMIT** operator is used to get a limited number of tuples from a relation.

### Syntax

```
grunt> Result = LIMIT Relation_name required number of tuples;
```

### Example:

```
grunt> limit_data = LIMIT student_details 4;
```

# Apache Pig - Eval Functions

Apache Pig provides various built-in functions namely **eval**, **load**, **store**, **math**, **string**, **bag** and **tuple** functions.

| S.N. | Function & Description                                                                                                                                     |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | <b>AVG()</b><br>To compute the average of the numerical values within a bag.                                                                               |
| 2    | <b>BagToString()</b><br>To concatenate the elements of a bag into a string. While concatenating, we can place a delimiter between these values (optional). |
| 3    | <b>CONCAT()</b><br>To concatenate two or more expressions of same type.                                                                                    |
| 4    | <b>COUNT()</b><br>To get the number of elements in a bag, while counting the number of tuples in a bag.                                                    |
| 5    | <b>COUNT_STAR()</b><br>It is similar to the <b>COUNT()</b> function. It is used to get the number of elements in a bag.                                    |
| 6    | <b>DIFF()</b><br>To compare two bags (fields) in a tuple.                                                                                                  |
| 7    | <b>IsEmpty()</b><br>To check if a bag or map is empty.                                                                                                     |

# Apache Pig - Eval Functions

|    |                                                                                                                                                        |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | <b>MAX()</b>                                                                                                                                           |
| 8  | To calculate the highest value for a column (numeric values or chararrays) in a single-column bag.                                                     |
|    | <b>MIN()</b>                                                                                                                                           |
| 9  | To get the minimum (lowest) value (numeric or chararray) for a certain column in a single-column bag.                                                  |
|    | <b>PluckTuple()</b>                                                                                                                                    |
| 10 | Using the Pig Latin <b>PluckTuple()</b> function, we can define a string Prefix and filter the columns in a relation that begin with the given prefix. |
|    | <b>SIZE()</b>                                                                                                                                          |
| 11 | To compute the number of elements based on any Pig data type.                                                                                          |
|    | <b>SUBTRACT()</b>                                                                                                                                      |
| 12 | To subtract two bags. It takes two bags as inputs and returns a bag which contains the tuples of the first bag that are not in the second bag.         |
|    | <b>SUM()</b>                                                                                                                                           |
| 13 | To get the total of the numeric values of a column in a single-column bag.                                                                             |
|    | <b>TOKENIZE()</b>                                                                                                                                      |
| 14 | To split a string (which contains a group of words) in a single tuple and return a bag which contains the output of the split operation.               |

## Apache Pig - Load & Store Functions

The **Load** and **Store** functions in Apache Pig are used to determine how the data goes ad comes out of Pig. These functions are used with the load and store operators. Given below is the list of load and store functions available in Pig.

| S.N. | Function & Description                                                                |
|------|---------------------------------------------------------------------------------------|
| 1    | <b>PigStorage()</b><br>To load and store structured files.                            |
| 2    | <b>TextLoader()</b><br>To load unstructured data into Pig.                            |
| 3    | <b>BinStorage()</b><br>To load and store data into Pig using machine readable format. |
| 4    | <b>Handling Compression</b><br>In Pig Latin, we can load and store compressed data.   |

# Apache Pig - Bag & Tuple Functions

| S.N. | Function & Description                                               |
|------|----------------------------------------------------------------------|
| 1    | <b>TOBAG()</b><br>To convert two or more expressions into a bag.     |
| 2    | <b>TOP()</b><br>To get the top <b>N</b> tuples of a relation.        |
| 3    | <b>TOTUPLE()</b><br>To convert one or more expressions into a tuple. |
| 4    | <b>TOMAP()</b><br>To convert the key-value pairs into a Map.         |

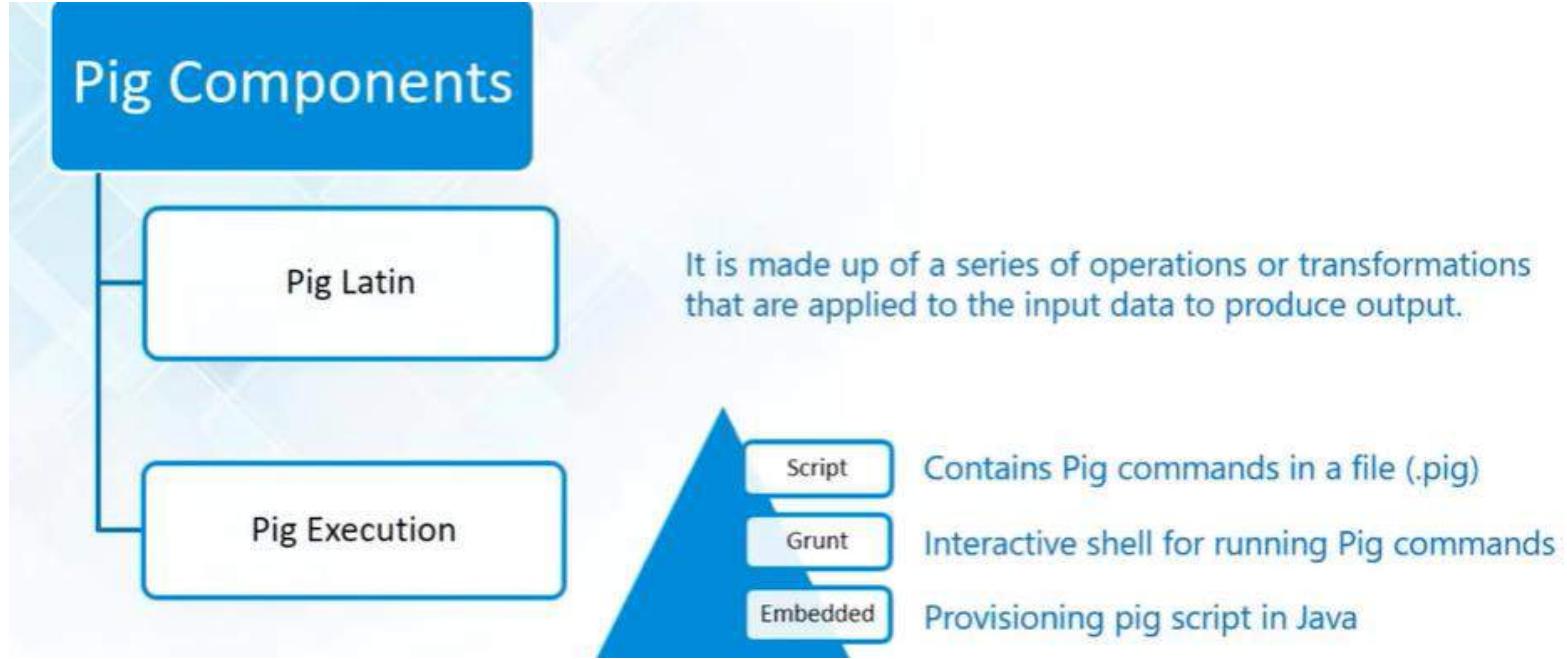
# Apache Pig - String Functions

| S.N. | Functions & Description                                                                                                                           |                                                                                                                                                                   |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | <b>ENDSWITH(string, testAgainst)</b><br>To verify whether a given string ends with a particular substring.                                        | 9                                                                                                                                                                 |
| 2    | <b>STARTSWITH(string, substring)</b><br>Accepts two string parameters and verifies whether the first string starts with the second.               | 10                                                                                                                                                                |
| 3    | <b>SUBSTRING(string, startIndex, stopIndex)</b><br>Returns a substring from a given string.                                                       | 11                                                                                                                                                                |
| 4    | <b>EqualsIgnoreCase(string1, string2)</b><br>To compare two strings ignoring the case.                                                            | 12                                                                                                                                                                |
| 5    | <b>INDEXOF(string, 'character', startIndex)</b><br>Returns the first occurrence of a character in a string, searching forward from a start index. | 13                                                                                                                                                                |
| 6    | <b>LAST_INDEX_OF(expression)</b><br>Returns the index of the last occurrence of a character in a string, searching backward from a start index.   | 14                                                                                                                                                                |
| 7    | <b>LCFIRST(expression)</b><br>Converts the first character in a string to lower case.                                                             | 15                                                                                                                                                                |
| 8    | <b>UCFIRST(expression)</b><br>Returns a string with the first character converted to upper case.                                                  | 16                                                                                                                                                                |
|      |                                                                                                                                                   | <b>UPPER(expression)</b><br>UPPER(expression) Returns a string converted to upper case.                                                                           |
|      |                                                                                                                                                   | <b>LOWER(expression)</b><br>Converts all characters in a string to lower case.                                                                                    |
|      |                                                                                                                                                   | <b>REPLACE(string, 'oldChar', 'newChar');</b><br>To replace existing characters in a string with new characters.                                                  |
|      |                                                                                                                                                   | <b>STRSPLIT(string, regex, limit)</b><br>To split a string around matches of a given regular expression.                                                          |
|      |                                                                                                                                                   | <b>STRSPLITTOBAG(string, regex, limit)</b><br>Similar to the <b>STRSPLIT()</b> function, it splits the string by given delimiter and returns the result in a bag. |
|      |                                                                                                                                                   | <b>TRIM(expression)</b><br>Returns a copy of a string with leading and trailing whitespaces removed.                                                              |
|      |                                                                                                                                                   | <b>LTRIM(expression)</b><br>Returns a copy of a string with leading whitespaces removed.                                                                          |
|      |                                                                                                                                                   | <b>RTRIM(expression)</b><br>Returns a copy of a string with trailing whitespaces removed.                                                                         |

# Apache Pig - Date-time Functions

| S.N. | Functions & Description                                                                                                                                                                                                                          |                                                                                                                                |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| 1    | <b>ToDate(milliseconds)</b><br>This function returns a date-time object according to the given parameters.<br>The other alternative for this function are ToDate(iosstring),<br>ToDate(userstring, format), ToDate(userstring, format, timezone) | 9                                                                                                                              |
| 2    | <b>CurrentTime()</b><br>returns the date-time object of the current time.                                                                                                                                                                        | 10                                                                                                                             |
| 3    | <b>GetDay(datetime)</b><br>Returns the day of a month from the date-time object.                                                                                                                                                                 | 11                                                                                                                             |
| 4    | <b>GetHour(datetime)</b><br>Returns the hour of a day from the date-time object.                                                                                                                                                                 | 12                                                                                                                             |
| 5    | <b>GetMilliSecond(datetime)</b><br>Returns the millisecond of a second from the date-time object.                                                                                                                                                | 13                                                                                                                             |
| 6    | <b>GetMinute(datetime)</b><br>Returns the minute of an hour from the date-time object.                                                                                                                                                           | 14                                                                                                                             |
| 7    | <b>GetMonth(datetime)</b><br>Returns the month of a year from the date-time object.                                                                                                                                                              | 15                                                                                                                             |
| 8    | <b>GetSecond(datetime)</b><br>Returns the second of a minute from the date-time object.                                                                                                                                                          | 16                                                                                                                             |
|      |                                                                                                                                                                                                                                                  | <b>GetWeek(datetime)</b><br>Returns the week of a year from the date-time object.                                              |
|      |                                                                                                                                                                                                                                                  | <b>GetWeekYear(datetime)</b><br>Returns the week year from the date-time object.                                               |
|      |                                                                                                                                                                                                                                                  | <b>GetYear(datetime)</b><br>Returns the year from the date-time object.                                                        |
|      |                                                                                                                                                                                                                                                  | <b>AddDuration(datetime, duration)</b><br>Returns the result of a date-time object along with the duration object.             |
|      |                                                                                                                                                                                                                                                  | <b>SubtractDuration(datetime, duration)</b><br>Subtracts the Duration object from the Date-Time object and returns the result. |
|      |                                                                                                                                                                                                                                                  | <b>DaysBetween(datetime1, datetime2)</b><br>Returns the number of days between the two date-time objects.                      |
|      |                                                                                                                                                                                                                                                  | <b>HoursBetween(datetime1, datetime2)</b><br>Returns the number of hours between two date-time objects.                        |
|      |                                                                                                                                                                                                                                                  | <b>MilliSecondsBetween(datetime1, datetime2)</b><br>Returns the number of milliseconds between two date-time objects.          |
|      |                                                                                                                                                                                                                                                  | <b>MinutesBetween(datetime1, datetime2)</b><br>Returns the number of minutes between two date-time objects.                    |
|      |                                                                                                                                                                                                                                                  | <b>MonthsBetween(datetime1, datetime2)</b><br>Returns the number of months between two date-time objects.                      |

# Apache Pig - Components



- Various ways to execute Pig Scripts
- Embedded : Execute over pigserver.

- **Pig Latin :** Very simple data flow language given by Apache Pig .
- Write , transformation and analysis can be performed over input data set

# Pig – Execution Modes

You can run  
Apache Pig  
in 2 modes:

**MapReduce Mode** – This is the default mode, which requires access to a Hadoop cluster and HDFS installation. The input and output in this mode are present on HDFS.

*Command: pig*

---

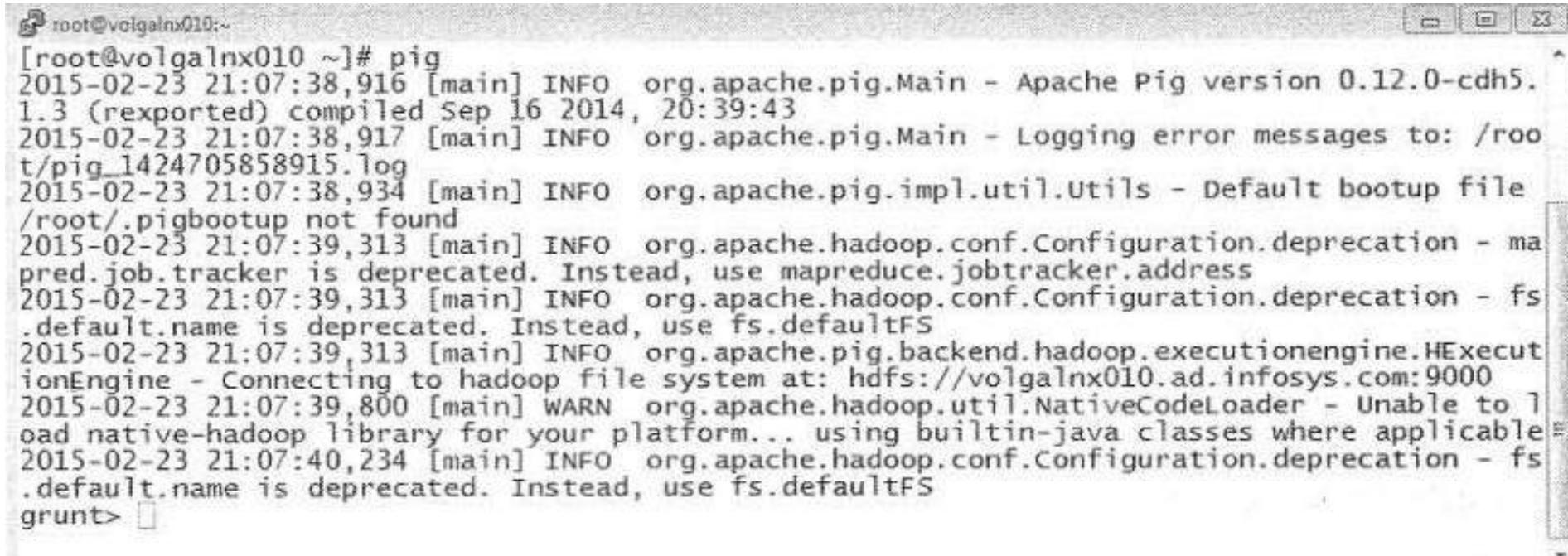
**Local Mode** – With access to a single machine, all files are installed and run using a local host and file system. Here the local mode is specified using '-x flag' (pig -x local). The input and output in this mode are present on local file system.

*Command: pig -x local*

---

# Running PIG

1. Interactive mode: Run pig in interactive mode by invoking **grunt** shell.



```
[root@volgalnx010 ~]# pig
2015-02-23 21:07:38,916 [main] INFO org.apache.pig.Main - Apache Pig version 0.12.0-cdh5.
1.3 (rexported) compiled Sep 16 2014, 20:39:43
2015-02-23 21:07:38,917 [main] INFO org.apache.pig.Main - Logging error messages to: /root/pig_1424705858915.log
2015-02-23 21:07:38,934 [main] INFO org.apache.pig.impl.util.Utils - Default bootup file /root/.pigbootup not found
2015-02-23 21:07:39,313 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2015-02-23 21:07:39,313 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-23 21:07:39,313 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://volgalnx010.ad.infosys.com:9000
2015-02-23 21:07:39,800 [main] WARN org.apache.hadoop.util.NativeCodeLoader - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2015-02-23 21:07:40,234 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

2. Batch mode: Create **pig script** to run in batch mode. Write pig latin statements in a file and save it **with .pig extension**

## Executing Pig Script in Batch mode

While executing Apache Pig statements in batch mode, follow the steps given below.

### Step 1

Write all the required Pig Latin statements in a single file. We can write all the Pig Latin statements and commands in a single file and save it as **.pig** file.

### Step 2

Execute the Apache Pig script. You can execute the Pig script from the shell (Linux) as shown below.

| Local mode                               | MapReduce mode                               |
|------------------------------------------|----------------------------------------------|
| \$ pig -x local <b>Sample_script.pig</b> | \$ pig -x mapreduce <b>Sample_script.pig</b> |

You can execute it from the Grunt shell as well using the exec command as shown below.

```
grunt> exec /sample_script.pig
```

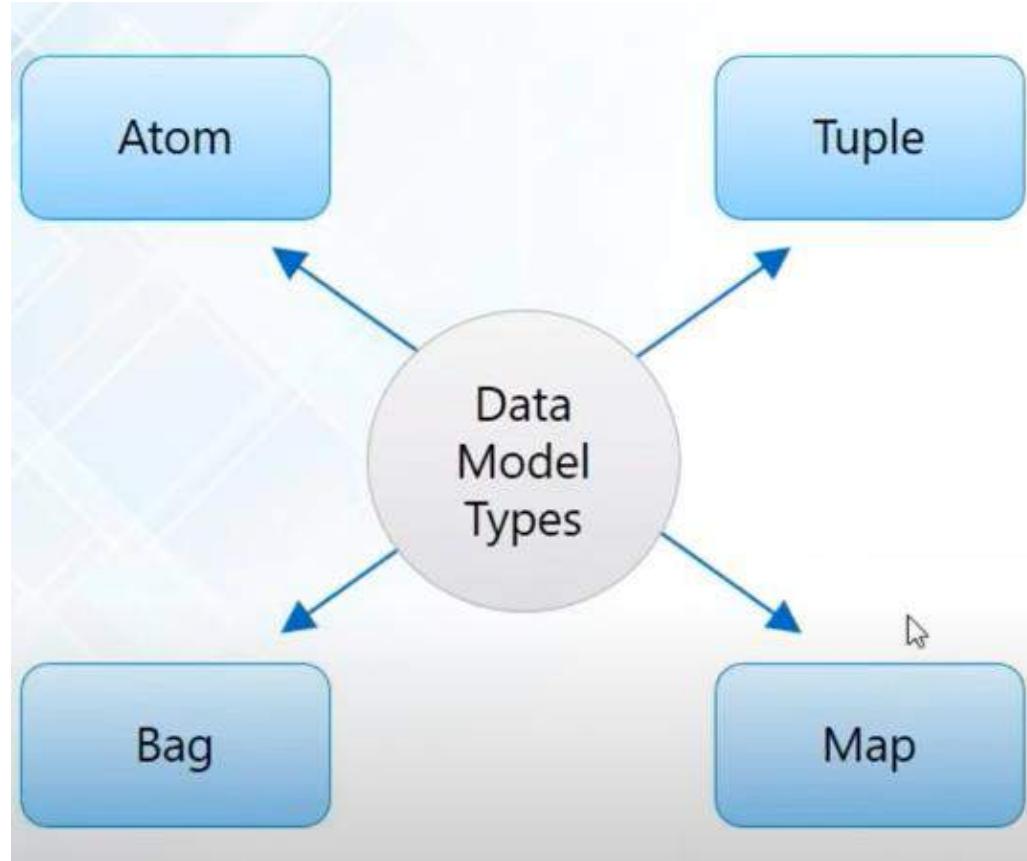
## Executing a Pig Script from HDFS

We can also execute a Pig script that resides in the HDFS.

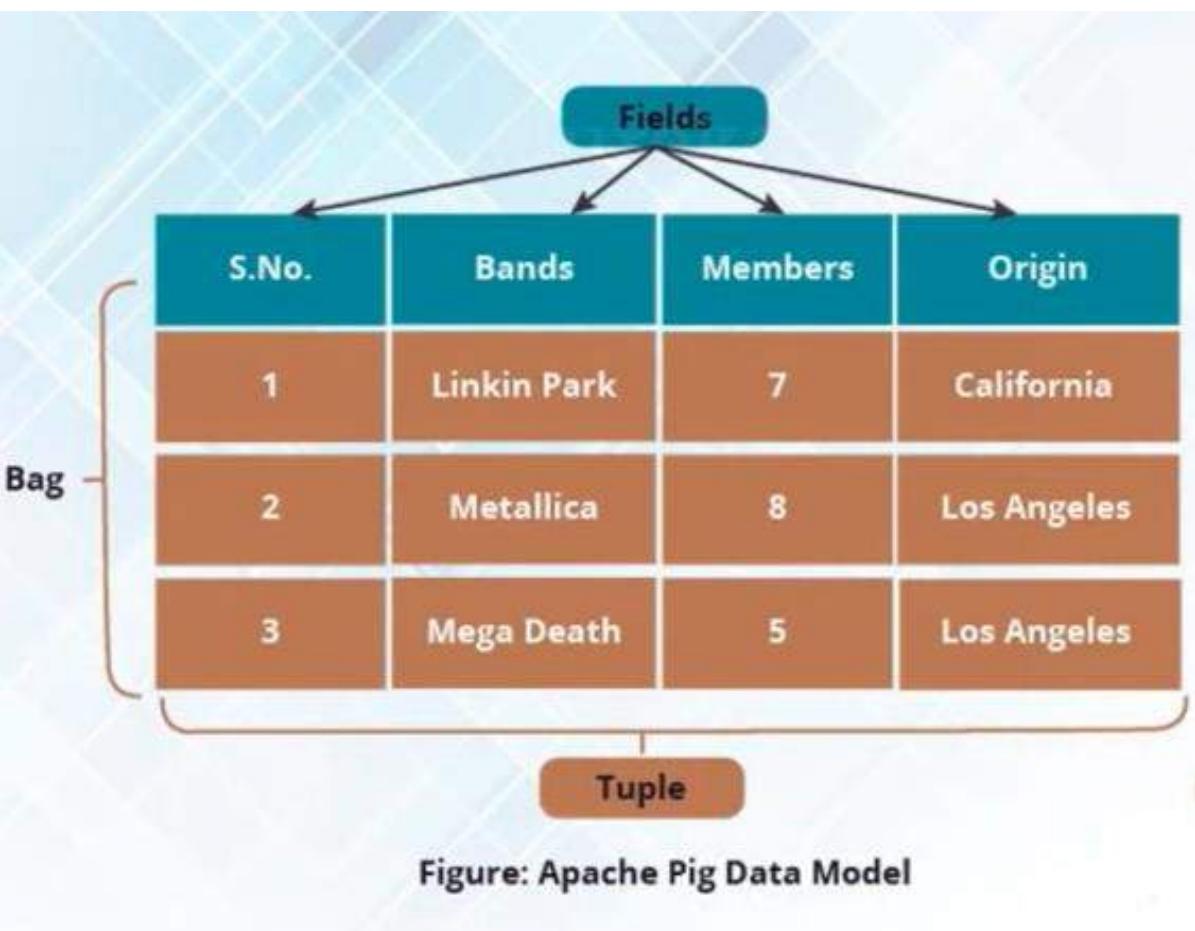
Suppose there is a Pig script with the name **Sample\_script.pig** in the HDFS directory named **/pig\_data/**. We can execute it as shown below.

```
$ pig -x mapreduce hdfs://localhost:9000/pig_data/Sample_script.pig
```

# Data Model : Pig



# Pig Data Model: Bag & Tuple



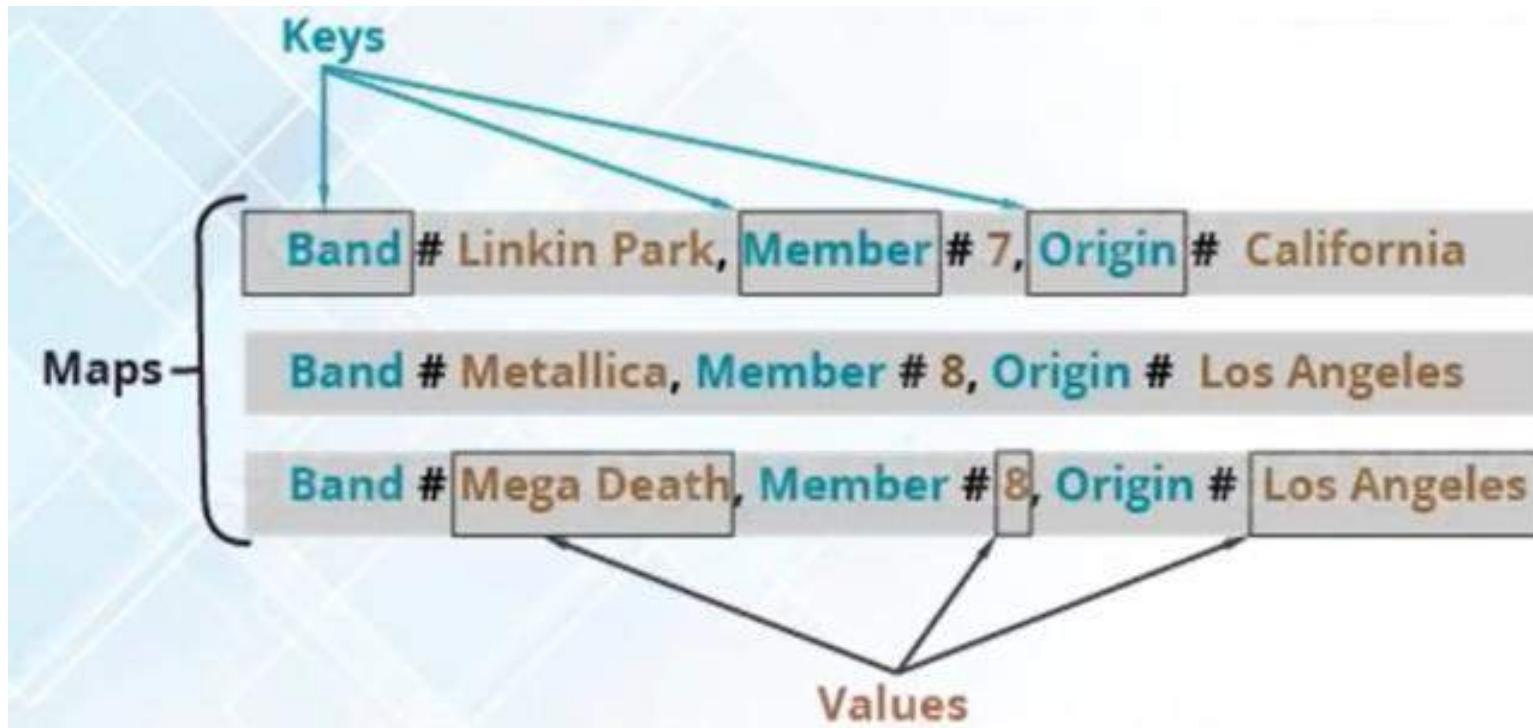
- **Tuple** is an ordered set of fields which may contain different data types for each field.

*Example of tuple – (1, Linkin Park, 7, California)*

- A **Bag** is a collection of a set of tuples and these tuples are subset of rows or entire rows of a table.

*Example of a bag – {(Linkin Park, 7, California), (Metallica, 8), (Mega Death, Los Angeles)}*

# Pig Data Model: Map & Atom



- A **Map** is key-value pairs used to represent data elements.  
Example of maps– [band#Linkin Park, members#7 ], [band#Metallica, members#8 ]
- **Atoms** are basic data types which are used in all the languages like string, int, float, long, double, char[], byte[]

# Pig: Operators

| Operator | Description                                                                   |
|----------|-------------------------------------------------------------------------------|
| LOAD     | Load data from the local file system or HDFS storage into Pig                 |
| FOREACH  | Generates data transformations based on columns of data                       |
| FILTER   | Selects tuples from a relation based on a condition                           |
| JOIN     | Join the relations based on the column                                        |
| ORDER BY | Sort a relation based on one or more fields                                   |
| STORE    | Save results to the local file system or HDFS                                 |
| DISTINCT | Removes duplicate tuples in a relation                                        |
| GROUP    | Groups together the tuples with the same group key (key field)                |
| COGROUP  | It is same as GROUP. But COGROUP is used when multiple relations are involved |

## Fill in??????

Pig is a scripting language.

In Pig, Pig latin is used to specify data flow.

Pig provides an Pig engine to execute data flow.

local, mapreduce are execution modes of Pig.

The interactive mode of Pig is grunt.

relation and path are case sensitive in Pig.

Bag, tuple, map, \_\_\_\_\_ are Complex Data Types of Pig.

Pig is used in ETL process.

## Can you Match ??????????

| Column A       | Column B                        |
|----------------|---------------------------------|
| Map            | Hadoop Cluster                  |
| Bag            | An Ordered Collection of Fields |
| Local Mode     | Collection of Tuples            |
| Tuple          | Key/Value Pair                  |
| MapReduce Mode | Local File System               |

## True / False ??????????

PigStorage() function is case sensitive. ✓

Local Mode is the default mode of Pig. ✗

DISTINCT Keyword removes duplicate fields. ✗

LIMIT keyword is used to display limited number of tuples in Pig. ✓

ORDER BY is used for sorting. ✓

# Piggy Bank

- Apache Pig provides extensive support for **User Defined Functions** (UDF's).
- The UDF support is provided in six programming languages, namely, Java, Jython, Python, JavaScript, Ruby and Groovy.
- For writing UDF's, complete support is provided in Java and limited support is provided in all the remaining languages.
- Since Apache Pig has been written in Java, the UDF's written using Java language work efficiently compared to other languages.
- In Apache Pig, we also have a Java repository for UDF's named **Piggybank**.
  - User can use piggy bank functions in pig latin script and can share their functions in piggy bank

## PIG EXECUTION : Load and Store data locally and on Hadoop

**Step1: Create input.txt file**

**Step2: Transfer to HDFS**

```
hdfs dfs put /home/hadoop/input.txt bda1/
```

**Step3: Create Pigscript file**

```
sudo gedit pigscript.pig
```

OR

```
vi pigscript.pig
```

**Code to be typed in pigscript.pig**

```
record = load '/bda1/input.txt/';
store record into '/bda1/out';
```

**Step4: Run pigscript in mapreduce mode**

```
pig -x mapreduce pigscript.pig
```

**Step5: Check the status of execution**

**Output:** Found 2 items

|                                                                           |
|---------------------------------------------------------------------------|
| -rw-r--r-- 2 hdoop supergroup 0 2024-01-12 15:05 /bda1/out/_SUCCESS       |
| -rw-r--r-- 2 hdoop supergroup 112 2024-01-12 15:05 /bda1/out/part-m-00000 |

**Step 6: View the output file**

```
hdfs dfs -cat /bda1/out/part-m-00000
```

## WORD COUNT PROGRAM

**Step1: Create a text file and add some contents to text file**

**Step 2: Open .pig file and edit the following script into that**

--LOAD THE DATA

```
records = LOAD '/pig1/input.txt';
```

-- SPLIT EACH LINE OF TEXT AND ELIMINATE NESTING

```
terms = FOREACH records GENERATE FLATTEN(TOKENIZE((chararray) $0)) AS word;
```

--GROUP SIMILAR TERMS

```
grouped_terms = GROUP terms BY word;
```

--COUNT THE NUMBER OF TUPLES IN EACH GROUP

```
word_counts = FOREACH grouped_terms GENERATE COUNT(terms), group;
```

--STORE THE RESULT

```
STORE word_counts INTO '/pig1/output';
```

Step 3: pig (type this at command prompt )

Step4: grunt>pig wordcount.pig

Step 5: grunt>run wordcount.pig

Step 6: grunt> pwd

Step7: grunt> cd /pig1/output

Step8: grunt> ls

**Output:**

hdfs://192.168.159.101:9000/pig1/output/\_SUCCESS<r 2> 0

hdfs://192.168.159.101:9000/pig1/output/part-r-00000<r 2>127

Step9: grunt> cat part-r-00000

**Output:**

2 i

2 am

1 hi

2 in

1 are

2 how

|   |          |
|---|----------|
| 1 | too      |
| 2 | you      |
| 1 | Data     |
| 2 | good     |
| 1 | hope     |
| 1 | you.     |
| 1 | Btech    |
| 1 | about    |
| 1 | doing    |
| 1 | manipal  |
| 1 | science. |
| 1 | studying |



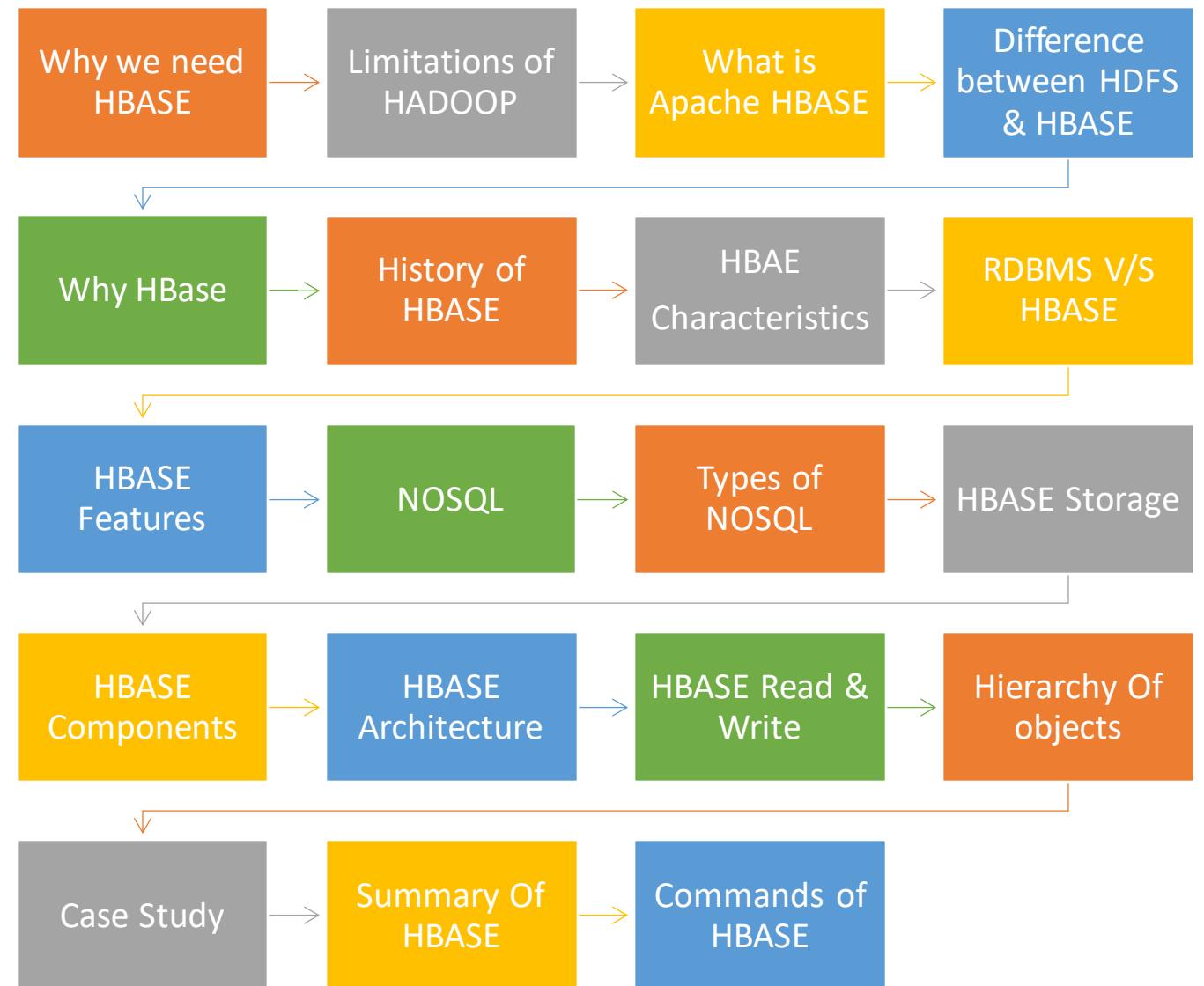
---

— Thank You! —



# Module 5 : HBASE

# AGENDA:

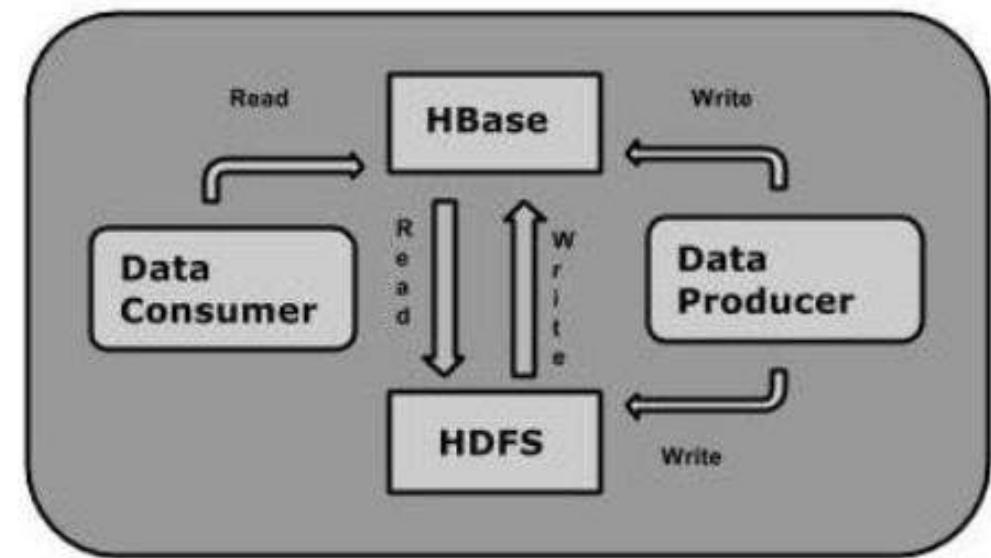


# What is HBase?

- HBase is a distributed **column-oriented database** built on top of the Hadoop file system.
- It is an **open-source** project and is **horizontally scalable**.
- Provides **random real-time read/write access to data** in the Hadoop File System.

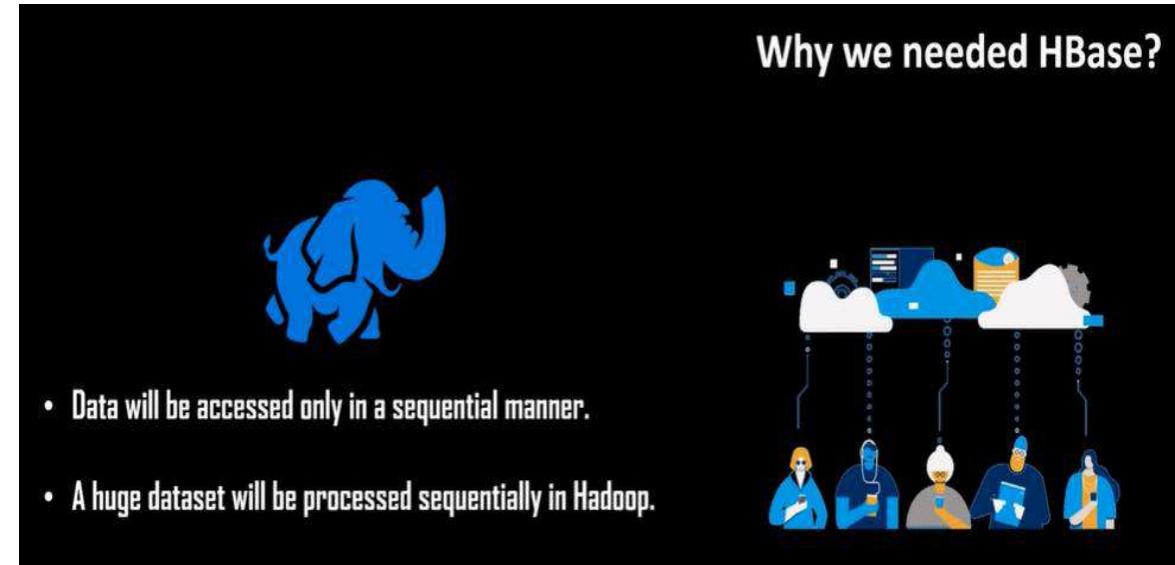
**COLUMN FAMILIES**

| Row key | personal data |           | professional data |        |
|---------|---------------|-----------|-------------------|--------|
| empid   | name          | city      | designation       | salary |
| 1       | raju          | hyderabad | manager           | 50,000 |
| 2       | ravi          | chennai   | sr.engineer       | 30,000 |
| 3       | rajesh        | delhi     | jr.engineer       | 25,000 |



# Why HBASE?

- RDBMS get exponentially slow as the data becomes large
- Expects data to be highly structured, i.e. ability to fit in a well-defined schema
- Any change in schema might require a **downtime**: Downtime can disrupt business operations, affect productivity, and potentially lead to data inconsistency or loss if not managed properly.
- For sparse datasets (many columns have NULL values), too much of overhead of maintaining NULL values
- It can store huge amounts of data in a tabular format for extremely fast reads and writes.
- HBase is used in a scenario that requires regular, consistent insertion and overwriting of data.
- HDFS stores, processes, and manages large amounts of data efficiently. Then, why HBASE???



# Why HBase:

HDFS stores, processes, and manages large amounts of data efficiently. However, it performs only batch processing and the data will be accessed in a sequential manner.

Data Analyst Jobs



Therefore, a solution is required to access, read, or write data anytime regardless of its sequence in the clusters of data.



Map Reduce  
(Hadoop)

Bigtable  
(Hypertable)  
anytime

Google File System  
(Hadoop)

# Limitations of HADOOP



Advantages :

Huge data storage

Data accessed – Sequential manner

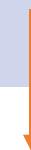


Disadvantage/Limitations:

To fetch few records it take **long time**.

It has scan for entire distributed file system – To fetch small record

Hadoop doesn't provide random access to database



Solution  
**“HBASE”**

# APACHE HBASE: Definition

HBASE: Opensource non-relation distributed database written in Java. It is developed as part of “Apache Software Foundation’s ” Apache Hadoop project and runs on “Top of HDFS”.

Its a **column-oriented database** built on top of HDFS.

Open-source project and “Horizontally Scalable”

HBASE – Data Model – Similar – “Google’s Big Data Table”.

Designed – To provide **quick random access** to huge amounts of structured data.

It leverages the “fault tolerance” provided by Hadoop file system and it is part of Hadoop ecosystem

It provides “random real –time read and write access to the data in HDFS

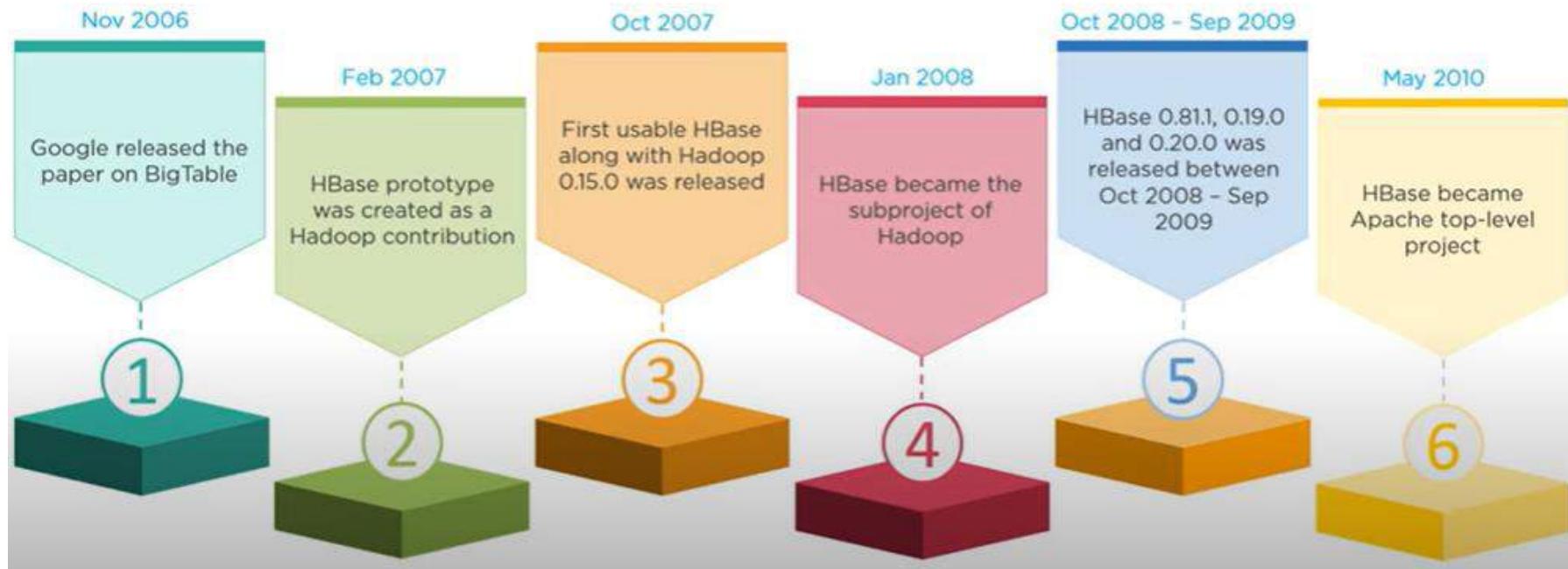
# HDFS VS HBASE

- |                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• HDFS is a distributed file system that stores huge files</li><li>• HDFS does not Support individual file lookups</li><li>• It has High latency</li><li>• Only sequential memory access is available</li></ul> | <ul style="list-style-type: none"><li>• HBase is built on top of HDFS</li><li>• HBase faster and individual file Lookups</li><li>• It has Low latency</li><li>• It has in-built Hash tables enabling faster lookups.</li></ul> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

# RDBMS Vs. HBase

| HBase                                                    | RDBMS                            |
|----------------------------------------------------------|----------------------------------|
| Column-oriented                                          | Row oriented (mostly)            |
| Flexible schema, add columns on the fly                  | Fixed schema.                    |
| Good with sparse tables,                                 | Not optimized for sparse tables. |
| Joins using MR –not optimized                            | Optimized for joins.             |
| Tight integration with MR                                | Not really...                    |
| Horizontal scalability –just add hardware                | Hard to shard and scale          |
| Good for semi-structured data as well as structured data | Good for structured data         |

# History of HBase



# Companies using Hbase:



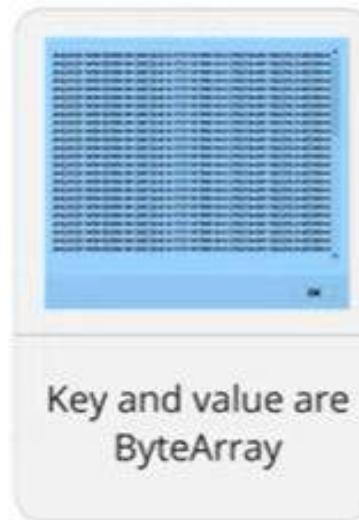
# Characteristics of HBase:

HBase is a type of NoSQL database and is classified as a key-value store.

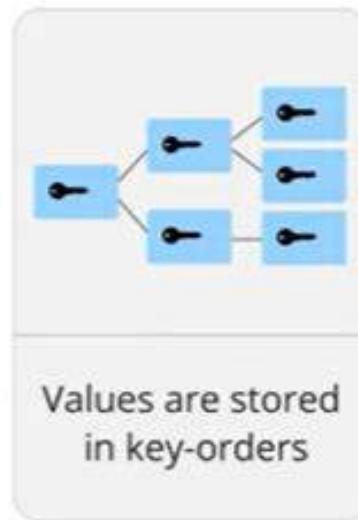
In HBase:



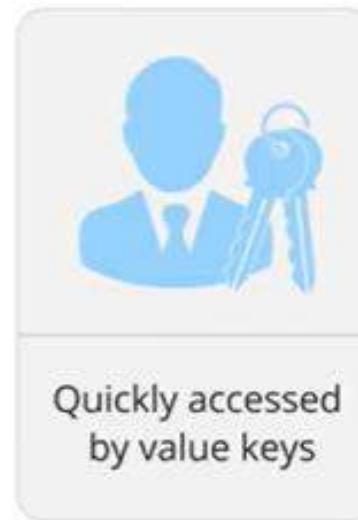
Value is identified  
with a key



Key and value are  
ByteArray



Values are stored  
in key-orders



Quickly accessed  
by value keys

HBase is a database in which tables have no schema. At the time of table creation, column families are defined, not columns.







# HBase Features:

Scalable



Data can be scaled across various nodes as it is stored in HDFS

Automatic failure support



Write Ahead Log across clusters which provides automatic support against failure

Consistent read and write



HBase provides consistent read and write of data

JAVA API for client access



Provides easy to use JAVA API for clients

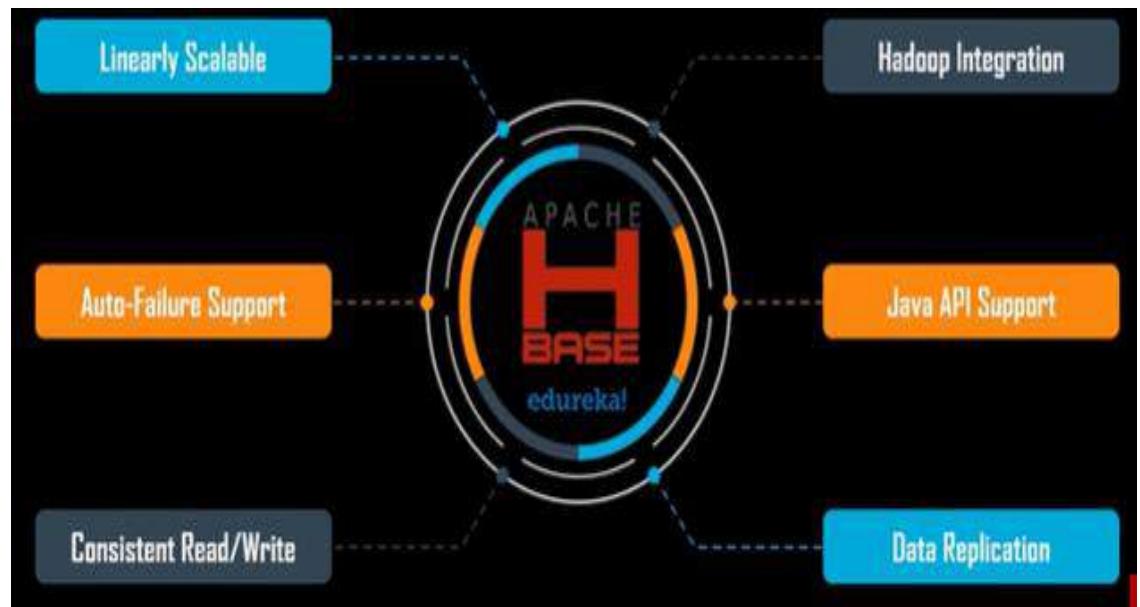
Block cache and bloom filters



Supports block cache and bloom filters for high volume query optimization

# HBASE Features

- **Linear Scalable** : HBASE built on top of HDFS.
  - HDFS – Horizontal scalable (Same feature is adopted by HBASE).
- **Auto Failure Support:** Support for “Fault Tolerance”
- **Consistent Read / Write:** Random Access of reading & writing data
- **HDFS Integration:** Integrates with Hadoop and both as a source and destination
- **Java API Support:** Easy for Java API for client
- **Data Replication:** Data replication across all the clusters

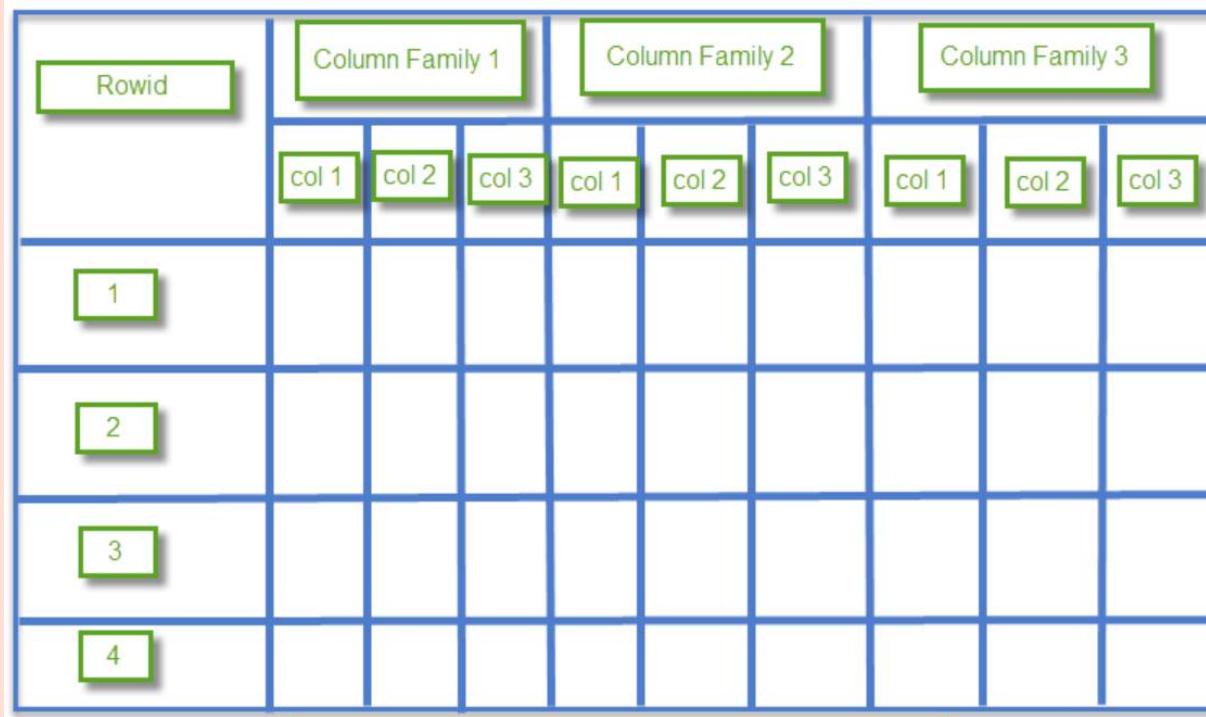


# HBASE – Storage Mechanism

| Row-ID | Column Family |        | Column Family |     |
|--------|---------------|--------|---------------|-----|
|        | Name          | Role   | Salary        | Age |
| 1      | Rajesh        | Tester | 35,000        | 28  |

- It is “**Column - Oriented**” database
- **Tables** – Sorted by – **Row**.
- **Table Schema** – Defines – Only Column families (which are Key –Value Pairs).
- Table – Collection of Rows
- Rows – Collection of Column Families
- Column Families – Collection of Column
- Column – Collection of key value pairs.

# HBASE – Storage Mechanism

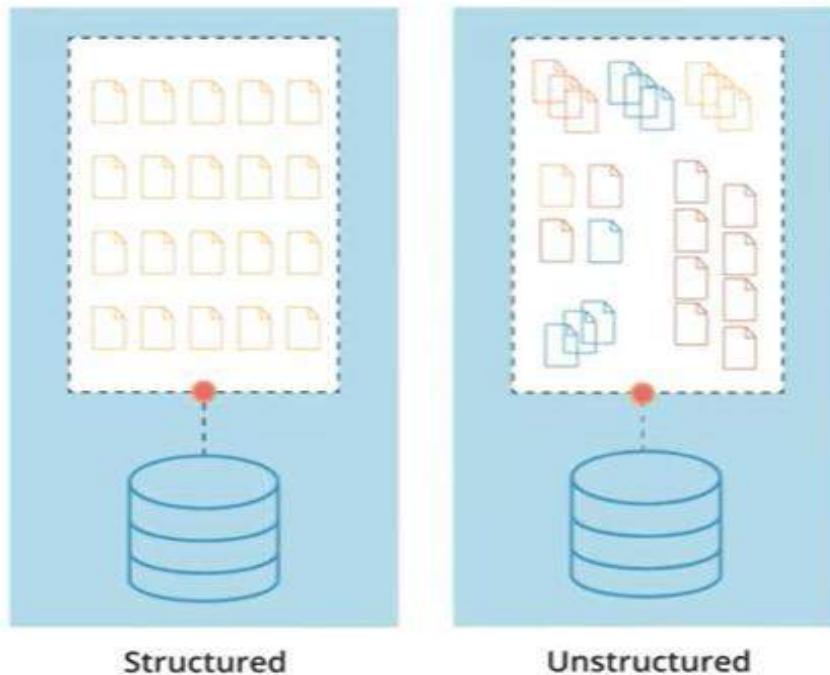


- HBase is a column-oriented database and data is stored in tables.
- The tables are sorted by RowId. As shown, HBase has RowId, which is the collection of several column families that are present in the table.
- The column families that are present in the schema are key-value pairs.
- If we observe in detail each column family having multiple numbers of columns.
- The column values stored into disk memory.
- Each cell of the table has its own **Metadata** like timestamp and other information.

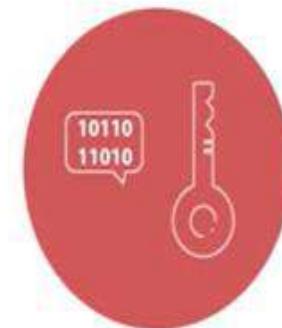
# NoSQL:

- NoSQL databases are databases that **store data in a format other than relational tables**.
- Types of NoSQL databases include pure **document databases, key-value stores, wide-column databases, and graph databases**.

NoSQL is a form of unstructured storage.



With the explosion of social media sites, such as Facebook and Twitter, the demand to manage large data has grown tremendously.



Key-value pair  
databases



Document  
databases



Column-based  
data stores

# Types of NoSQL:

## Key Value



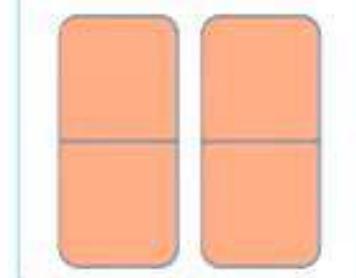
Examples:  
Oracle NoSQL, Redis  
server, Scalaris

## Document-based



Examples:  
MongoDB, CouchDB,  
OrientDB, RavenDB

## Column-based



Examples:  
BigTable, Cassandra,  
HBase, Hypertable

## Graph-based



Examples:  
Neo4J, InfoGrid, Infinite  
Graph, FlockDB

# Types of NoSQL:

## Key value :

- Every data element in the database is stored as a **key value pair** consisting of an attribute name (or "key") and a value. In a sense, a key-value store is **like a relational database with only two columns**: the key or attribute name and the value.
- The data can be **retrieved** by using a unique key allotted to each element in the database. The values can be simple data types like strings and numbers or complex objects.
- key-value store is more **flexible** because each user's data can have different attributes without requiring changes to the database schema.

## Example

Key: "user:1"

Value:{

  "name": "Alice",

  "email": "alice@email.com",

  "age": 30

}

# Types of NoSQL

## document database:

- It stores data in **JSON, BSON, or XML** documents (not Word documents or Google Docs, of course). In a document database, **documents can be nested**. Particular elements can be indexed for faster querying.
- Documents can be **stored and retrieved in a form that is much closer to the data objects** used in applications which means **less translation is required to use these data in the applications**.
- In the Document database, the **particular elements can be accessed by using the index value** that is assigned for faster querying.
- [Example](#)

```
{"employees": [
 {"name": "Shyam", "email": "shyamjaiswal@gmail.com"},
 {"name": "Bob", "email": "bob32@gmail.com"},
 {"name": "Jai", "email": "jai87@gmail.com"}
]}
```

# Types of NoSQL

## column-oriented database

- **non-relational database** that **stores the data in columns** instead of rows. That means when we want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data.
- Columnar databases are designed **to read data more efficiently** and **retrieve the data with greater speed**. A columnar database is used to store a large amount of data. **Key features** of columnar oriented database:
  - Scalability.
  - Compression.
  - Very responsive.

# Types of NoSQL

## A graph database

- focuses on the relationship between data elements.
- Each element is stored as a node (such as a person in a social media graph). The connections between elements are called links or relationships.
- In a graph database, connections are first-class elements of the database, stored directly.

## Key features of graph database:

- In a graph-based database, it is **easy to identify the relationship between the data by using the links**.
- The **Query's output is real-time results**.
- The **speed depends upon the number of relationships among the database elements**.

# HBASE :Major Components

- Client library:
  - Connecting to the HBase Cluster
  - Table Operations
  - Data Manipulation
  - Batch Operations
  - Scanning
  - Filtering
- Master Server
- Region Server



# 1.Master Server:

- Assigns regions to the region servers and takes the help of Apache Zookeeper.
- Handles load balancing of the regions across region servers.
- Maintains the state of the cluster by negotiating the load balancing.

Primary responsibilities of the master server:

## 1.Cluster Coordination:

1. The master server acts as the central coordinator for the HBase cluster. It monitors the health and status of all the RegionServers in the cluster, ensuring that they are functioning correctly.

## 2.Assignment of Regions:

1. One of the key responsibilities of the master server is to assign regions of HBase tables to different RegionServers in the cluster. It determines the distribution of regions across RegionServers based on factors such as load balancing, data locality, and cluster capacity.

## 3.Table Management:

1. The master server is responsible for managing HBase tables and their schemas. It handles tasks such as creating, deleting, enabling, and disabling tables. When a new table is created, the master server coordinates the initial assignment of regions to RegionServers.

# Master Server:

- **Metadata Management:**
  - The master server **maintains** important **metadata about HBase tables, regions, and RegionServers**. This metadata includes information about **table schemas, region locations, and server assignments**. **Clients use this metadata to locate and access data in the cluster efficiently.**
- **Cluster Operations:**
  - The master server provides APIs and interfaces for performing administrative tasks and cluster management operations. This includes tasks such as **starting and stopping RegionServers, compacting and splitting regions, and monitoring cluster health and performance**.
- **Schema Changes:**
  - If schema modifications are requested (such as adding or removing columns from a table), the master server **coordinates these changes across the cluster**. It ensures that schema changes are propagated to all relevant RegionServers and that they are applied correctly.
- **Failover Handling:**
  - In the event of a **RegionServer failure**, the master server detects the failure and coordinates the reassignment of regions hosted by the failed RegionServer to other healthy RegionServers. This ensures continuous availability and fault tolerance of HBase tables.
- **Load Balancing:**
  - The master server is responsible for load balancing across RegionServers to ensure even distribution of data and workload. It monitors the resource usage and data distribution across the cluster and may trigger region reassessments to achieve better load balancing.

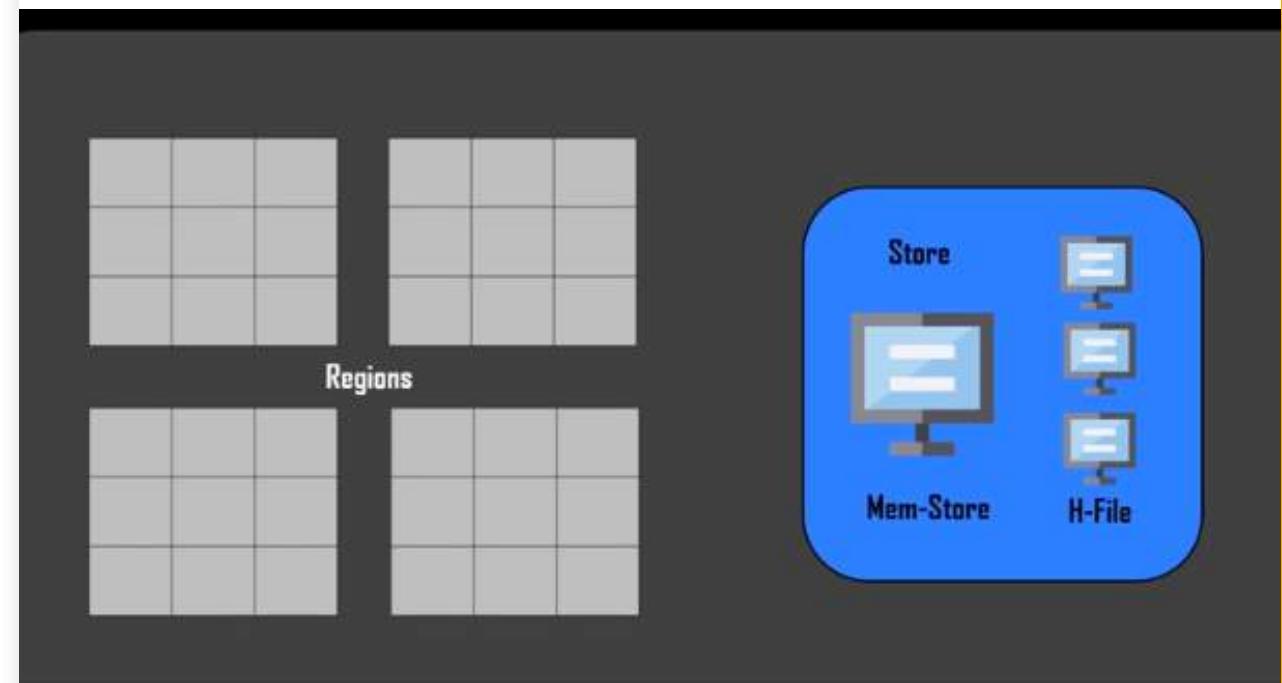
## 2. Region :

- Communicate with the client and handle data-related operations.
- Handle read and write requests for all the regions under it.
- Decide the size of the region by following the region size thresholds.

- Regions – Nothing but “Tables”
- Tables – split up across the “region servers”.

# 3. Region Server

- **Store** - Contains “memory file” & “H-File” .
- **Mem- store** – Just like cache memory.
- Anything that is entered in “HBASE” is automatically **stored initially**.
- Later the data is transferred and saved in H-Files as “**blocks**”.
- Later the mem-store is **erased out** .
- Region Management
- Data Storage and Retrieval



# Zookeeper

- Zookeeper is an open-source project that provides services like maintaining configuration.
- Zookeeper has ephemeral nodes representing different region servers.
- Clients communicate with region servers via zookeeper.



- Zookeeper: Providing Distributed synchronization, Naming etc
- It has **ephemeral node**: Master servers use these nodes to **discover available nodes**.
- Based on availability: Nodes are also used to track server failure or network partitions.
- In sudo / stand alone mode: HBASE itself take care of Zookeeper

# HBase Architecture:

HBase has two types of Nodes—Master and RegionServer. Following are the characteristics of the two nodes.

## Master

- Only one Master node runs at a time. Its high availability is maintained with ZooKeeper.
- It manages cluster operations like assignment, load balancing, and splitting.
- It is not a part of the read/write path.

## RegionServer

- One or more RegionServers can exist at a time.
- It hosts tables, performs reads, and buffers writes.
- Clients communicate with RegionServers for read/write operation.

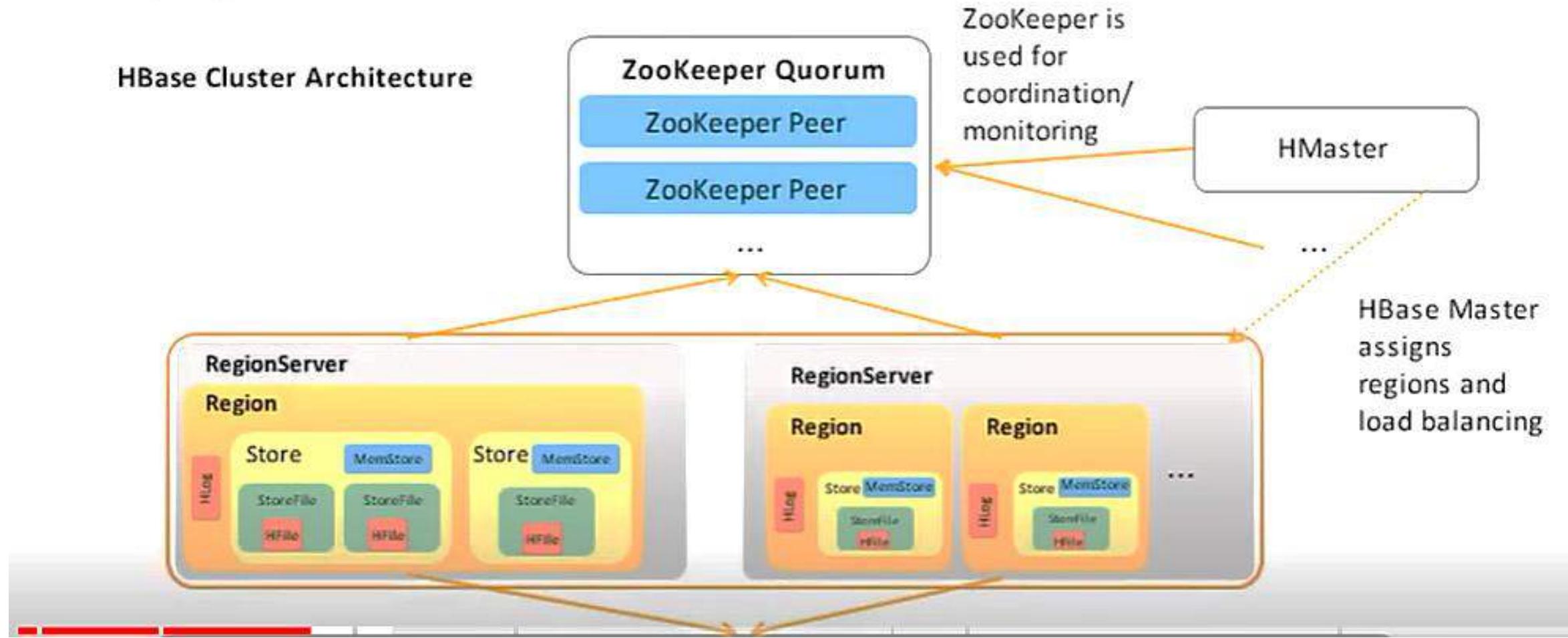
# HBASE - ARCHIZTECTURE



- Inside HBASE: **Tables are split into “Regions”.**
- Regions – Served by – Server Regions.
- Regions Servers– Vertically Divided by “column families” into stores.
- Stores – are saved as files in HDFS.
- HBASE – 3 components
- Client library
- Master Server
- Region Server

# HBase Architecture:

The image represents the components of HBase—HBase Master and RegionServers.



HDFS

# HBase Architecture:



ZooKeeper is used for monitoring



HMaster

HBase Master assigns regions and load balancing

Region server serves data for read and write

Region Server

Region

Store

MemStore

StoreFile

StoreFile

HLog

HFile

HFile

Region Server

Region

Store

MemStore

StoreFile

StoreFile

HLog

HFile

HFile

Region Server

Region

Store

MemStore

StoreFile

StoreFile

HLog

HFile

HFile

HDFS

# Column – Oriented v/s Row Oriented

## Column-oriented Database

- When the situation comes to process and analytics we use this approach. Such as **Online Analytical Processing** and it's applications.
- The amount of data that can able to store in this model is very huge like in terms of petabytes

## Row oriented Database

- **Online Transactional process** such as banking and finance domains use this approach.
- It is designed for a small number of rows and columns.

# Storage Model of HBase:

The two major components of the storage model are as follows:



## Partitioning:

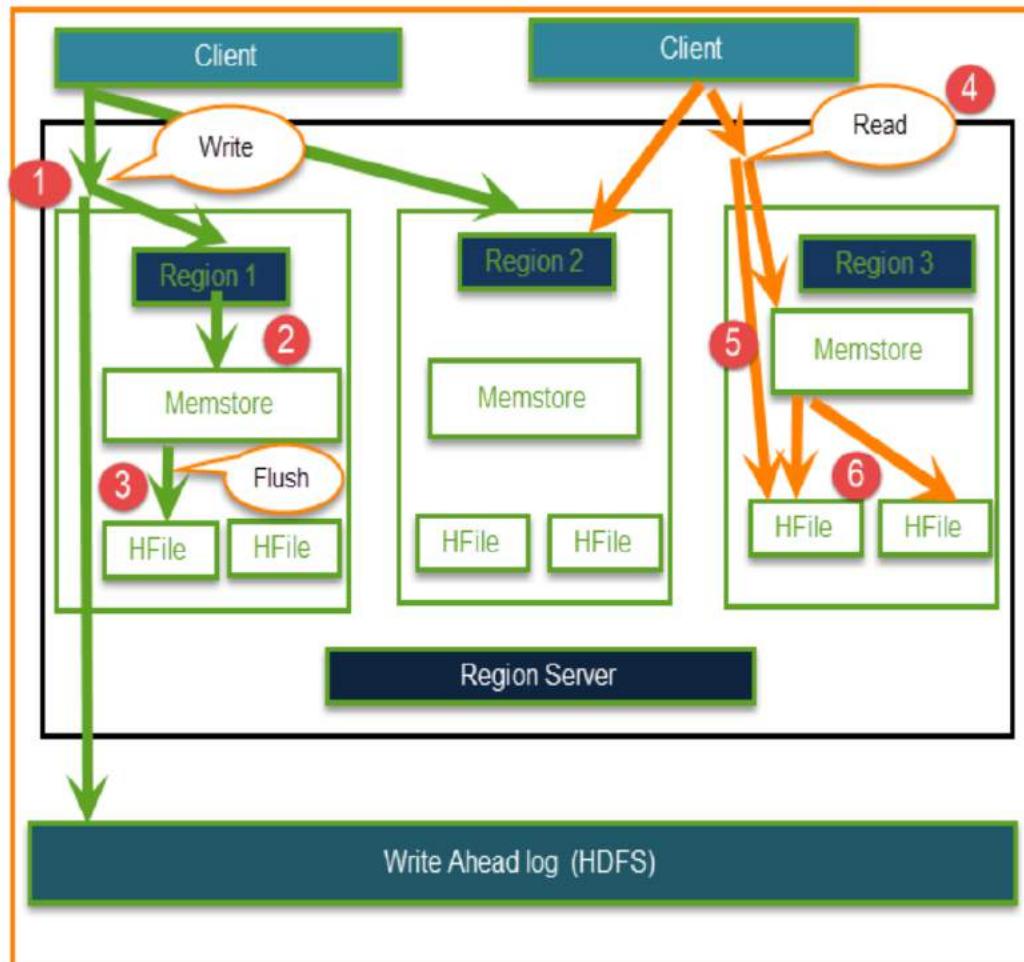
- A table is horizontally partitioned into regions.
- Each region is managed by a RegionServer.
- A RegionServer may hold multiple regions.



## Persistence and data availability:

- HBase stores its data in HDFS, does not replicate RegionServers, and relies on HDFS replication for data availability.
- Updates and reads are served from the in-memory cache called MemStore.

# HDFS – Read & Write Data



Step 1) Client wants to write data and in turn first communicates with Regions server and then regions

Step 2) Regions contacting memstore for storing associated with the column family

Step 3) First data stores into Memstore, where the data is sorted and after that, it flushes into HFile. The main reason for using Memstore is to store data in a Distributed file system based on Row Key. Memstore will be placed in Region server main memory while HFiles are written into HDFS.

Step 4) Client wants to read data from Regions

Step 5) In turn Client can have direct access to Mem store, and it can request for data.

Step 6) Client approaches HFiles to get the data. The data are fetched and retrieved by the Client.

## HBase Architecture: HBase Write Mechanism

**Step 1:** Whenever the client has a write request, the client writes the data to the WAL (Write Ahead Log).

- The edits are then appended at the end of the WAL file.
- This WAL file is maintained in every Region Server and Region Server uses it to recover data which is not committed to the disk.

**Step 2:** Once data is written to the WAL, then it is copied to the MemStore.

**Step 3:** Once the data is placed in MemStore, then the client receives the acknowledgment.

**Step 4:** When the MemStore reaches the threshold, it dumps or commits the data into a HFile.

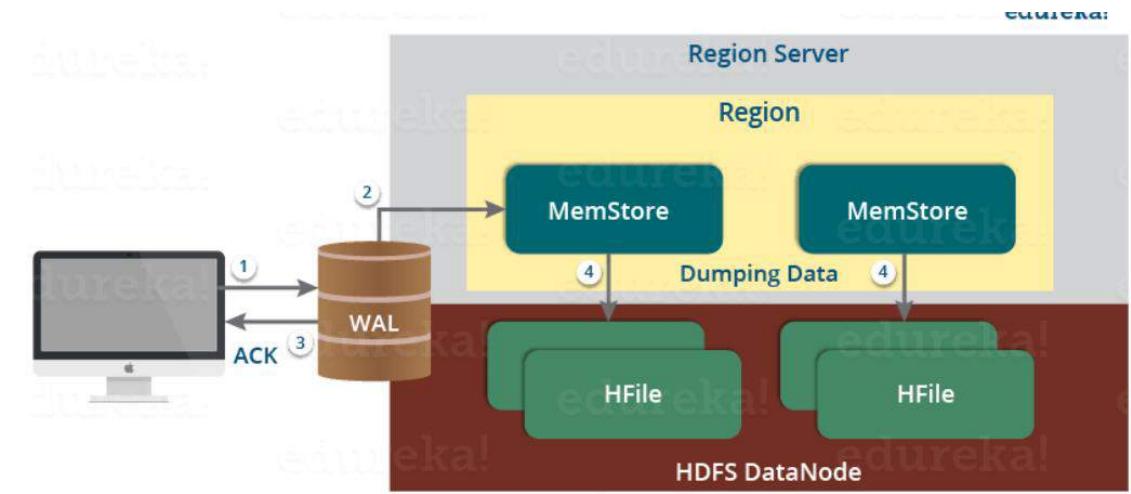


Figure: Write Mechanism in HBase

## HBase Write Mechanism- MemStore

- The MemStore always updates the data stored in it, in a lexicographical order (sequentially in a dictionary manner) as **sorted KeyValues**. There is **one MemStore for each column family**, and thus the **updates are stored in a sorted manner for each column family**.
- When the MemStore **reaches the threshold**, it dumps all the data into a **new HFile** in a sorted manner. This HFile is stored in HDFS. HBase contains **multiple HFiles for each Column Family**.
- Over time, the number of **HFile grows as MemStore dumps the data**.
- MemStore also saves the last written sequence number**, so Master Server and MemStore both know, that **what is committed so far and where to start from**. When region starts up, the **last sequence number is read**, and from that number, **new edits start**.

## HBase Architecture: HBase Write Mechanism- Hfile

- The writes are placed sequentially on the disk. Therefore, the movement of the disk's read-write head is very less. This makes write and search mechanism very fast.
- The HFile indexes are loaded in memory whenever an HFile is opened. This helps in finding a record in a single seek.
- The **trailer is a pointer which points to the HFile's meta block**. It is written at the end of the committed file. It contains information about timestamp and bloom filters.
- Bloom Filter helps in searching key value pairs, it **skips the file which does not contain the required rowkey**. Timestamp also helps in searching a **version of the file**, it helps in skipping the data.

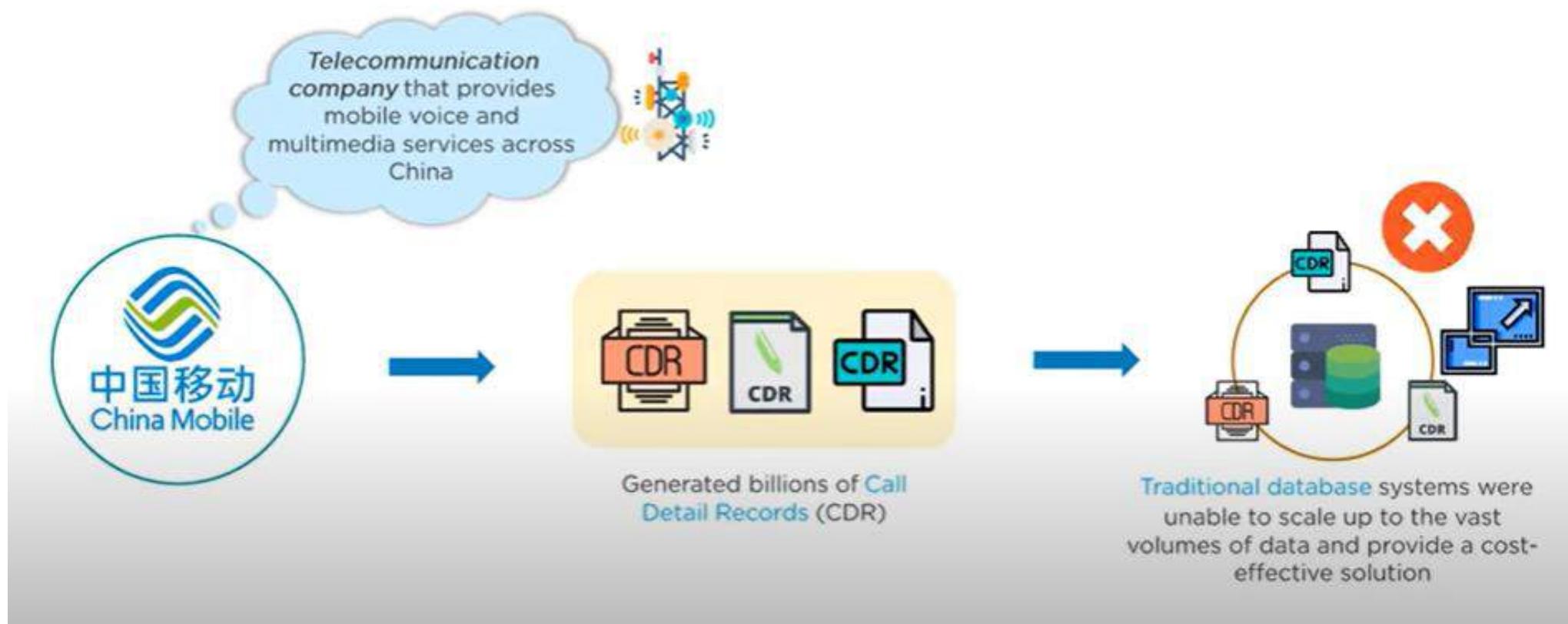
# Hierarchy of Objects:

|           |                                                                                                                                                                                                                                   |                                                                                                                                                                                                           |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Table     | HBase table present in the HBase cluster                                                                                                                                                                                          |                                                                                                                                                                                                           |
| Region    | HRegions for the presented tables                                                                                                                                                                                                 |                                                                                                                                                                                                           |
| Store     | It stores per ColumnFamily for each region for the table                                                                                                                                                                          |                                                                                                                                                                                                           |
| Memstore  | <ul style="list-style-type: none"><li>• Memstore for each store for each region for the table</li><li>• It sorts data before flushing into HFiles</li><li>• Write and read performance will increase because of sorting</li></ul> | <ul style="list-style-type: none"><li>• Memstore holds in-memory modifications to the store.</li><li>• The hierarchy of objects in HBase Regions is as shown from top to bottom in below table.</li></ul> |
| StoreFile | StoreFiles for each store for each region for the table                                                                                                                                                                           |                                                                                                                                                                                                           |
| Block     | Blocks present inside StoreFiles                                                                                                                                                                                                  |                                                                                                                                                                                                           |

# Case Study : Telecom

| Problem Statement                                                                                                                                                                                                                                                                                                                                                                         | Solution                                                                                                                                                                                                                                                                                                    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Telecom Industry</b> faces following Technical challenges</p> <ul style="list-style-type: none"><li>• Storing billions of CDR (Call detailed recording) log records generated by telecom domain</li><li>• Providing real-time access to CDR logs and billing information of customers</li><li>• Provide cost-effective solution comparing to traditional database systems</li></ul> | <p>HBase is used to store billions of rows of detailed call records. If 20TB of data is added per month to the existing RDBMS database, performance will deteriorate. To handle a large amount of data in this use case, HBase is the best solution. HBase performs fast querying and displays records.</p> |

# HBase Use Case:



# Case Study : Banking

The **Banking industry** generates millions of records on a daily basis. In addition to this, the banking industry also needs an analytics solution that can detect Fraud in money transactions

To store, process and update vast volumes of data and performing analytics, an ideal solution is – HBase integrated with several Hadoop ecosystem components.

## **When to Use Hbase:**

- Whenever there is a need to write heavy applications.
- Performing online log analytics and to generate compliance reports.

# HBase Commands

- **Create:** Creates a new table identified by 'table1' and Column Family identified by 'colf'.

create 'table1', 'colf'

- **Put:** Inserts a new record into the table with row identified by 'row.'

list 'table1'

- **Scan:** returns the data stored in table\

put 'table1', 'row1', 'colf:a',  
'value1'

- **Get:** Returns the records matching the row identifier provided in the table

put 'table1', 'row1', 'colf:b',  
'value2'

- **Help:** Get a list of commands

put 'table1', 'row2', 'colf:a',  
'value3'

scan 'table1'

get 'table1', 'row1'

# Summary – HBASE:

- HBase architecture components: HMaster, HRegion Server, HRegions, ZooKeeper, HDFS
- HMaster in HBase is the implementation of a Master server in HBase architecture.
- When HBase Region Server receives writes and read requests from the client, it assigns the request to a specific region, where the actual column family resides
- HRegions are the basic building elements of HBase cluster that consists of the distribution of tables and are comprised of Column families.
- HBase Zookeeper is a centralized monitoring server which maintains configuration information and provides distributed synchronization.
- HDFS provides a high degree of fault-tolerance and runs on cheap commodity hardware.
- HBase Data Model is a set of components that consists of Tables, Rows, Column families, Cells, Columns, and Versions.
- Column and Row-oriented storages differ in their storage mechanism.

# Commands: HBASE

```
File Edit View Search Terminal Help
hbase(main):005:0> create 'employee', 'Master: quickstart.cloudera' 'Designation', 'Salary', 'Department'
0 row(s) in 1.4460 seconds

=> Hbase::Table - employee
hbase(main):006:0> list
TABLE
employee
1 row(s) in 0.0300 seconds

=> ["employee"]
hbase(main):007:0> disable 'employee'
0 row(s) in 2.4640 seconds

hbase(main):008:0> scan 'employee'■
```

```
cloudera@quickstart:~
File Edit View Search Terminal Help
hbase(main):009:0> create 'employee2', 'Designation', 'Salary', 'Department'
0 row(s) in 1.2580 seconds

=> Hbase::Table - employee2
hbase(main):010:0> disable_all 'e.*'
employee
employee2

Disable the above 2 tables (y/n)?
n
```

- disable ‘table’
- drop ‘table’

# Adding Data – Table

```
File Edit View Search Terminal Help
hbase(main):011:0> create 'student', 'name', 'age', 'course'
0 row(s) in 1.2410 seconds

=> Hbase::Table - student
hbase(main):012:0> put 'student','sharath', 'name:fullname', 'sharath kumar'
0 row(s) in 0.2560 seconds

hbase(main):013:0> put 'student','sharath', 'age:presentage', '24'
0 row(s) in 0.0130 seconds

hbase(main):014:0> put 'student','sharath', 'course:pursuing', 'Hadoop'
0 row(s) in 0.0170 seconds

hbase(main):015:0> put 'student','shashank', 'name:fullname', 'shashank R'
0 row(s) in 0.0140 seconds

hbase(main):016:0> put 'student','shashank', 'age:presentage', '23'
0 row(s) in 0.0150 seconds

hbase(main):017:0> put 'student','shashank', 'course:pursuing', 'Java'■
```

# Entire Record / Specific Record

```
File Edit View Search Terminal Help
hbase(main):018:0> get 'student','shashank'
COLUMN CELL
 age:presentage timestamp=1583476124493, value=23
 course:pursuing timestamp=1583476133671, value=Java
 name:fullname timestamp=1583476115741, value=shashank R
3 row(s) in 0.0290 seconds

hbase(main):019:0> ■ I

hbase(main):020:0> get 'student','sharath','course'
COLUMN CELL
 course:pursuing timestamp=1583476107762, value=Hadoop
1 row(s) in 0.0540 seconds

hbase(main):021:0> get 'student','sharath','name'
COLUMN CELL
 name:fullname timestamp=1583476045340, value=sharath kumar
1 row(s) in 0.0210 seconds
```

```
File Edit View Search Terminal Help
hbase(main):022:0> scan 'student'
ROW
sharath
sharath
sharath
shashank
shashank
shashank
2 row(s) in 0.0840 seconds
```

COLLECTION+CELL  
column=age:presentage, timestamp=1583476082211, value=24  
column=course:pursuing, timestamp=1583476107762, value=Hadoop  
column=name:fullname, timestamp=1583476045340, value=sharath kumar  
column=age:presentage, timestamp=1583476124493, value=23  
column=course:pursuing, timestamp=1583476133671, value=Java  
column=name:fullname, timestamp=1583476115741, value=shashank R

```
hbase(main):016:0> scan 'guru99'
ROW
r1
r2
r3
3 row(s) in 0.0340 seconds
```

COLLECTION+CELL  
column=c1:, timestamp=30, value=value  
column=c1:, timestamp=15, value=value  
column=c1:, timestamp=15, value=value

scanning 'guru99' table records

```
hbase(main):023:0> count 'student'
2 row(s) in 0.0320 seconds
```

=> 2

```
hbase(main):024:0> alter 'student',NAME=>'age',VERSIONS=>5
```

Updating all regions with the new schema...

0/1 regions updated.

1/1 regions updated.

Done.

0 row(s) in 3.1190 seconds

## show\_filters

Syntax: show\_filters

```
hbase(main):013:0> show_filters
ColumnPrefixFilter
TimestampsFilter
PageFilter
MultipleColumnPrefixFilter
FamilyFilter
```

- **Filtering** – Reading data from HBASE using get / scan operations.
- Return **subset** of results to **clients**.
- Inorder to use these **filters** – import “java” classes to Hbase shell

This command displays all the filters present in HBase like ColumnPrefix Filter, TimestampsFilter, PageFilter, FamilyFilter, etc.



THANK  
YOU.



PART 6

# Apache Spark



# AGENDA

- Apache Spark
- Pre- Requisites of Spark
- Uses of Spark
- Features / Why Spark
- Cluster Managers
- Storage layers of Spark
- Quick Introduction
- RDD
- RDD Operations
- Types of RDD
- Spark Architecture
- Spark Program execution
- Case Study
- Sample Transformations- Scala
- Actions Execution - Scala
- Defining Schema/ DF
- Non \_-Defining Schema/DF
- Parquet files

# What is Spark

- Open source software tool
- Distributed cluster computing
- Batch/Stream processing
- Integrated with various big data tools
- Interactive querying
- Cluster management system

# What is APACHE SPARK?



Apache Spark is an open-source data processing engine to store and process data in real-time across various clusters of computers using simple programming constructs

Support various programming languages



Developers and data scientists incorporate Spark into their applications to rapidly query, analyze, and transform data at scale



Query



Analyze



Transform

# Limitations of MapReduce

- Use only Java for application building.
- Opt **only for batch processing**. Does not support stream processing.
- Hadoop MapReduce **uses disk-based processing**.

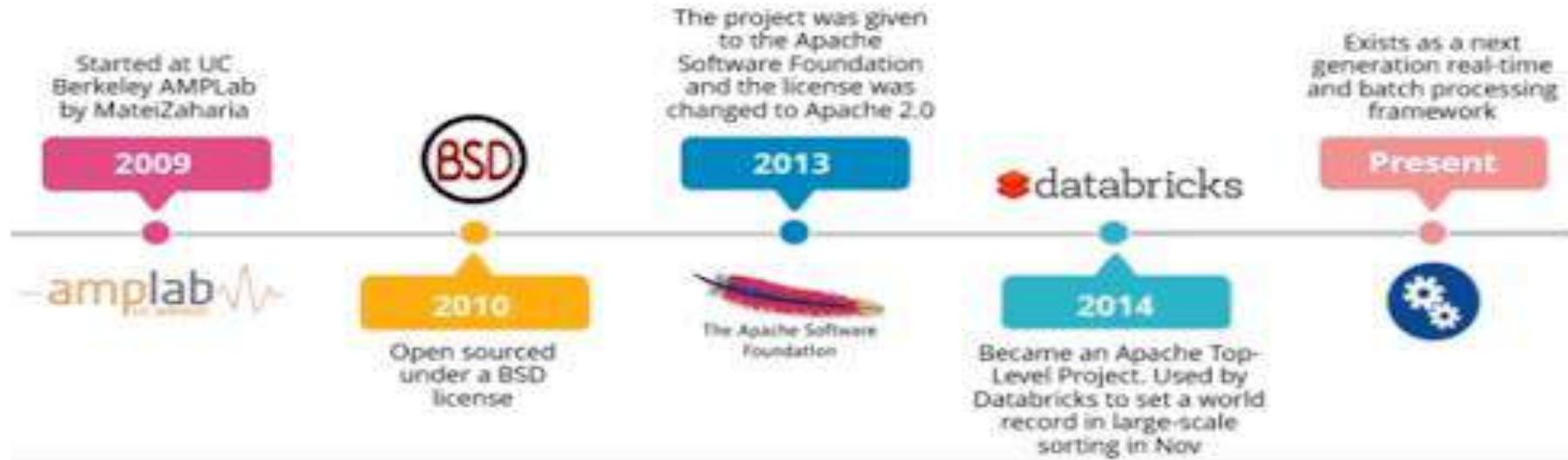
# Why Spark?

In the industry, there is a need for a **general-purpose cluster** computing tool as:

- Hadoop **MapReduce** can only perform batch processing.
- Apache Storm / S4 can only perform stream processing.
- Apache Impala / Apache Tez can only perform interactive processing
- Neo4j / Apache Giraph can only perform graph processing
- There was a big demand for a powerful engine that can process the **data in real-time (streaming) as well as in batch mode.**
- There was a need for an engine that can respond in sub-second and perform **in-memory processing**.

# History of Spark?

The history of Spark is explained below:

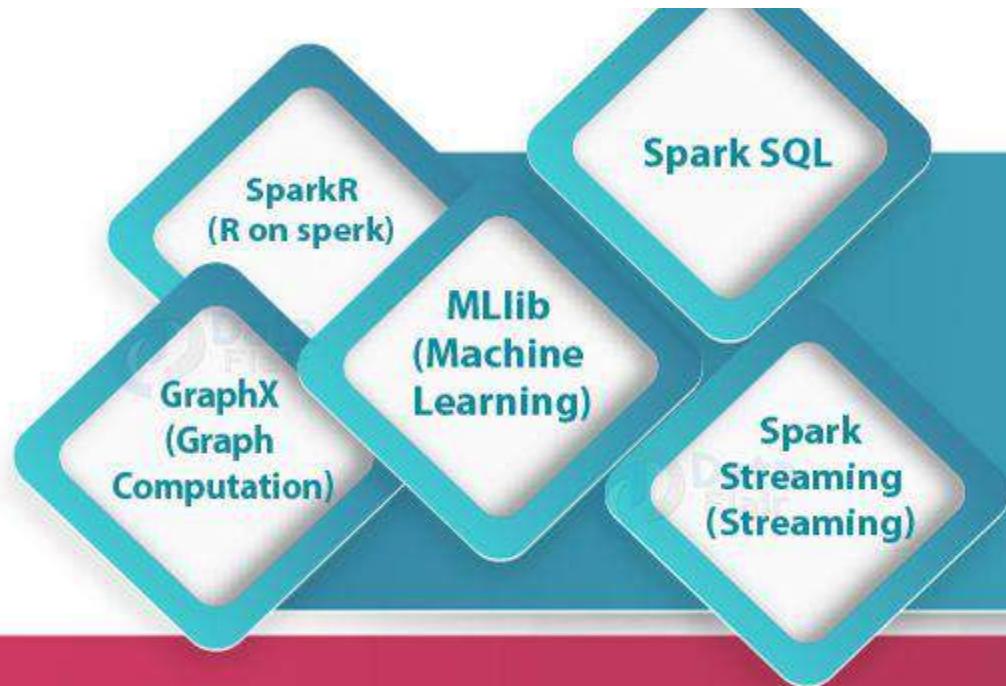


- The Spark was initiated by Matei Zaharia at UC Berkeley's AMPLab in 2009.
- It was open sourced in 2010 under a BSD license.
- In 2013, the project was acquired by Apache Software Foundation.
- In 2014, the Spark emerged as a Top-Level Apache Project.

## Apache Spark:

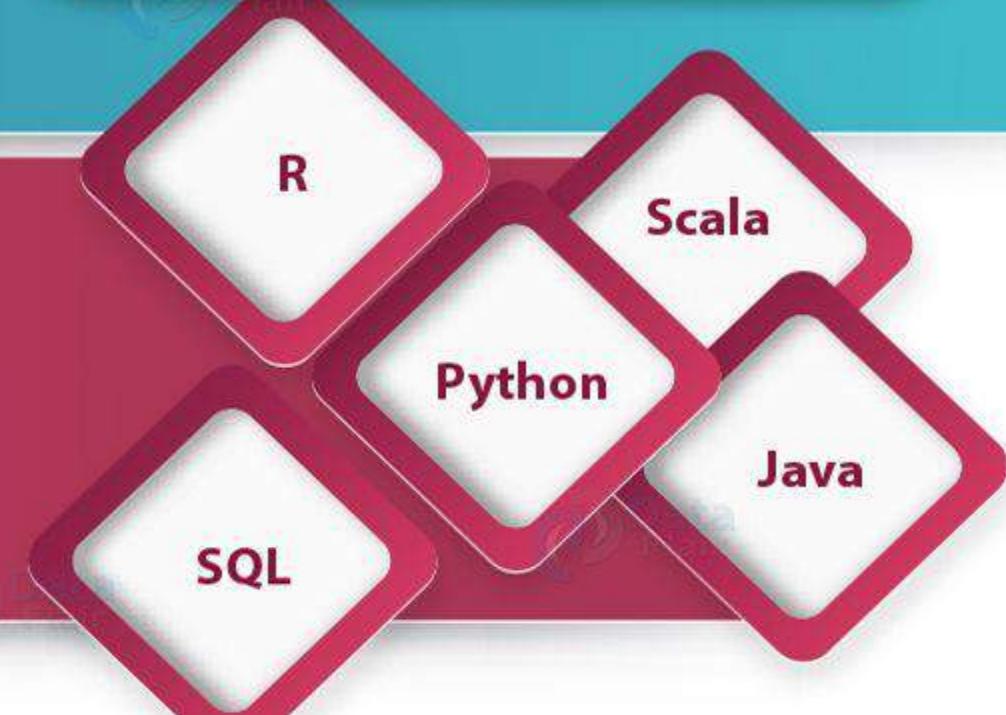
- ▶ Cluster computing platform designed to be fast and general purpose.
- ▶ In memory computation.
- ▶ Designed to cover wide range of workloads(batch applications, machine learning, interactive queries, streaming).
- ▶ Combines different processing types.
- ▶ Integrates closely with other Big Data tools.

# Apache Spark Ecosystem



Apache Spark Core API

Apache Spark Ecosystem



# Hadoop V/S Spark

| Basis                                     | Hadoop                                                                                         | Spark                                                                                                                        |
|-------------------------------------------|------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <b>Processing Speed &amp; Performance</b> | Hadoop's MapReduce model reads and writes from a disk, thus slowing down the processing speed. | Spark reduces the number of read/write cycles to disk and stores intermediate data in memory, hence faster-processing speed. |
| <b>Usage</b>                              | Hadoop is designed to handle batch processing efficiently.                                     | Spark is designed to handle real-time data efficiently.                                                                      |
| <b>Latency</b>                            | Hadoop is a high latency computing framework, which does not have an interactive mode.         | Spark is a low latency computing and can process data interactively.                                                         |
| <b>Data</b>                               | With Hadoop MapReduce, a developer can only process data in batch mode only.                   | Spark can process real-time data, from real-time events like Twitter, and Facebook.                                          |
| <b>Cost</b>                               | Hadoop is a cheaper option available while comparing it in terms of cost                       | Spark requires a lot of RAM to run in-memory, thus increasing the cluster and hence cost.                                    |
| <b>Algorithm Used</b>                     | The PageRank algorithm is used in Hadoop.                                                      | Graph computation library called GraphX is used by Spark.                                                                    |

# Hadoop V/S Spark:

|                            |                                                                                                                                                                      |                                                                                                                                                      |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Fault Tolerance</b>     | Hadoop is a highly fault-tolerant system where Fault-tolerance achieved by replicating blocks of data.<br>If a node goes down, the data can be found on another node | Fault-tolerance achieved by storing chain of transformations<br>If data is lost, the chain of transformations can be recomputed on the original data |
| <b>Security</b>            | Hadoop supports LDAP, ACLs, SLAs, etc and hence it is extremely secure.                                                                                              | Spark is not secure, it relies on the integration with Hadoop to achieve the necessary security level.                                               |
| <b>Machine Learning</b>    | Data fragments in Hadoop can be too large and can create bottlenecks. Thus, it is slower than Spark.                                                                 | Spark is much faster as it uses MLlib for computations and has in-memory processing.                                                                 |
| <b>Scalability</b>         | Hadoop is easily scalable by adding nodes and disk for storage.<br>It supports tens of thousands of nodes.                                                           | It is quite difficult to scale as it relies on RAM for computations. It supports thousands of nodes in a cluster.                                    |
| <b>Language support</b>    | It uses Java or Python for MapReduce apps.                                                                                                                           | It uses Java, R, Scala, Python, or Spark SQL for the APIs.                                                                                           |
| <b>User-friendliness</b>   | It is more difficult to use.                                                                                                                                         | It is more user-friendly.                                                                                                                            |
| <b>Resource Management</b> | YARN is the most common option for resource management.                                                                                                              | It has built-in tools for resource management.                                                                                                       |

# Hadoop V/S Apache Spark

| Features                  | Hadoop                                                                                          | Apache Spark                                                                                                                |
|---------------------------|-------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Data Processing           | Apache Hadoop provides batch processing                                                         | Apache Spark provides both batch processing and stream processing                                                           |
| Memory usage              | Hadoop is disk-bound                                                                            | Spark uses large amounts of RAM                                                                                             |
| Security                  | Better security features                                                                        | Its security is currently in its infancy                                                                                    |
| Fault Tolerance           | Replication is used for fault tolerance.                                                        | RDD and various data storage models are used for fault tolerance.                                                           |
| Graph Processing          | Algorithms like PageRank is used.                                                               | Spark comes with a graph computation library called GraphX.                                                                 |
| Ease of Use               | Difficult to use.                                                                               | Easier to use.                                                                                                              |
| Real-time data processing | It fails when it comes to real-time data processing.                                            | It can process real-time data.                                                                                              |
| Speed                     | Hadoop's MapReduce model reads and writes from a disk, thus it slows down the processing speed. | Spark reduces the number of read/write cycles to disk and store intermediate data in memory, hence faster-processing speed. |
| Latency                   | It is high latency computing framework.                                                         | It is a low latency computing and can process data interactively                                                            |

# Pre-Requisites For Spark

- ▶ Basic knowledge of Python, Scala, SQL, Linux, Hadoop, Statistics.

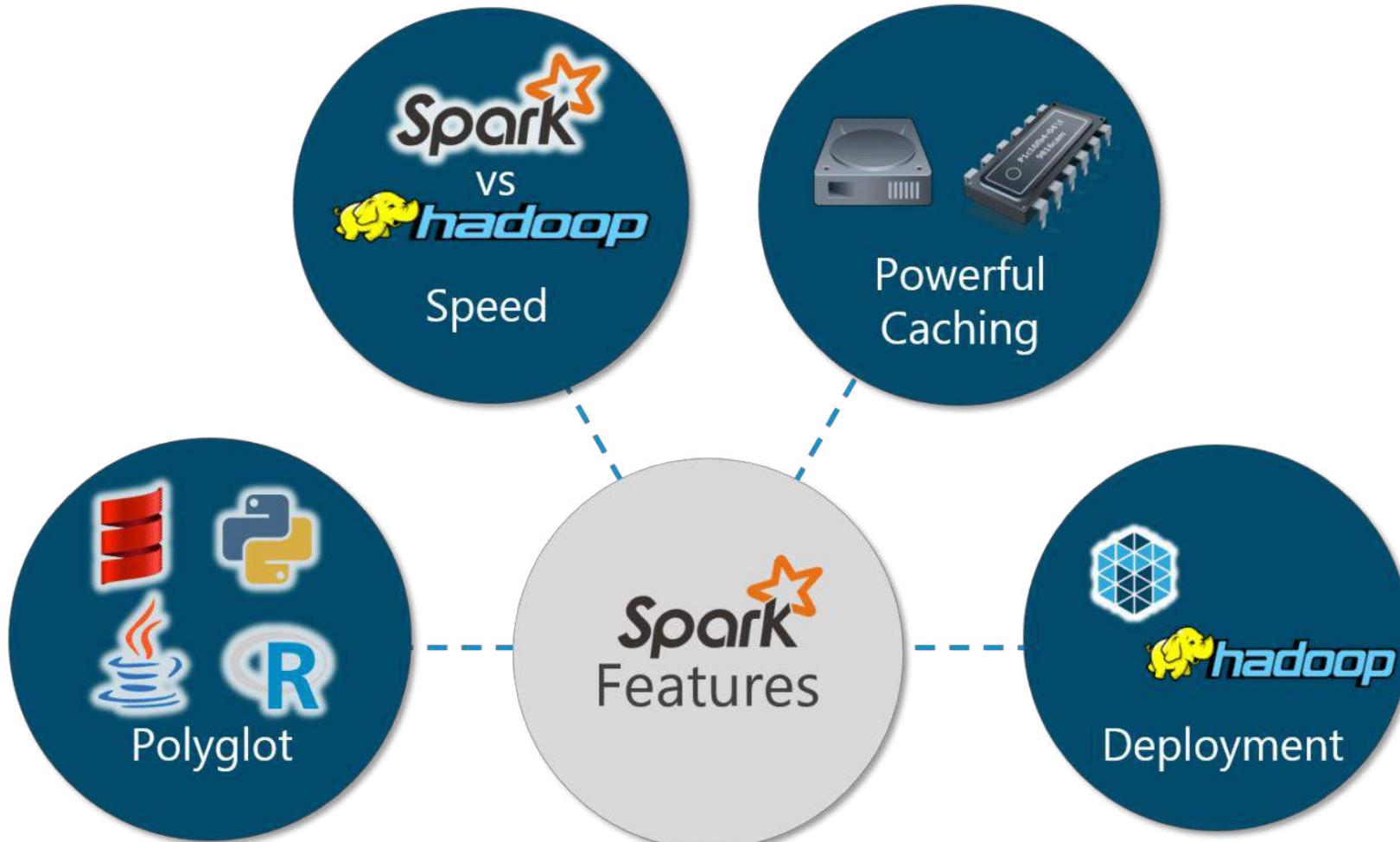
# Uses of Spark:

- ▶ Data processing applications
  - Batch Applications
  - SQL
  - Machine Learning
  - Streaming data processing
  - Graph data processing

# Feature of Spark / Why Spark?

- ▶ In memory processing.
- ▶ Tight integration of components.
- ▶ Easy and inexpensive.

# Features of Spark



## Features of Apache Spark

**Swift Processing:** swift processing speed of up to 100x faster in memory and 10x faster than Hadoop even when running on disk.

**Dynamic:** 80 high-level operators, Scala being defaulted language for Spark. We can also work with Java, Python, R.

**In – Memory Processing:** Disk seeks is becoming very costly, Spark keeps data in memory for faster access. Spark owns advanced DAG execution engine.

**Reusability:** Apache Spark provides the provision of code reusability for batch processing, join streams against historical data, or run adhoc queries on stream state.

**Fault Tolerance:** Spark RDD (Resilient Distributed Dataset), abstraction are designed to seamlessly handle failures of any worker nodes in the cluster.  
Real-Time Stream Processing

# Features of Spark

## Polyglot:

Spark provides high-level APIs in Java, Scala, Python and R. Spark code can be written in any of these four languages. It provides a shell in Scala and Python. The Scala shell can be accessed through **./bin/spark-shell** and Python shell through **./bin/pyspark** from the installed directory.



## Speed:



Spark runs up to 100 times faster than Hadoop MapReduce for large-scale data processing. Spark is able to achieve this speed through controlled partitioning. It manages data using partitions that help parallelize distributed data processing with minimal network traffic.

# Features of Spark

## Multiple Formats:

Spark supports multiple data sources such as Parquet, JSON, Hive and Cassandra apart from the usual formats such as text files, CSV and RDBMS tables. The Data Source API provides a pluggable mechanism for accessing structured data through Spark SQL. Data sources can be more than just simple pipes that convert data and pull it into Spark.



## Real Time Computation:

Spark's computation is real-time and has low latency because of its in-memory computation. Spark is designed for massive scalability and the Spark team has documented users of the system running production clusters with thousands of nodes and supports several computational models.



# Features of Spark



- 1  + 
- 2  + 

## Hadoop Integration:

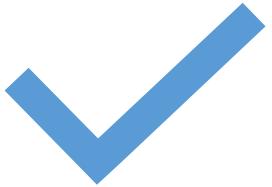
Apache Spark provides smooth compatibility with Hadoop. This is a boon for all the Big Data engineers who started their careers with Hadoop. Spark is a potential replacement for the MapReduce functions of Hadoop, while Spark has the ability to run on top of an existing Hadoop cluster using YARN for resource scheduling.

## Machine Learning:

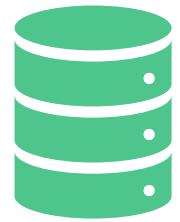
Spark's MLlib is the machine learning component which is handy when it comes to big data processing. It eradicates the need to use multiple tools, one for processing and one for machine learning. Spark provides data engineers and data scientists with a powerful, unified engine that is both fast and easy to use.



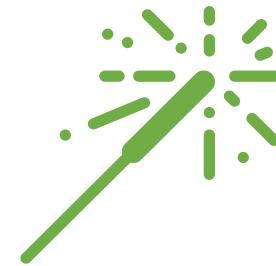
# Cluster Managers:



Hadoop YARN



Apache Mesos



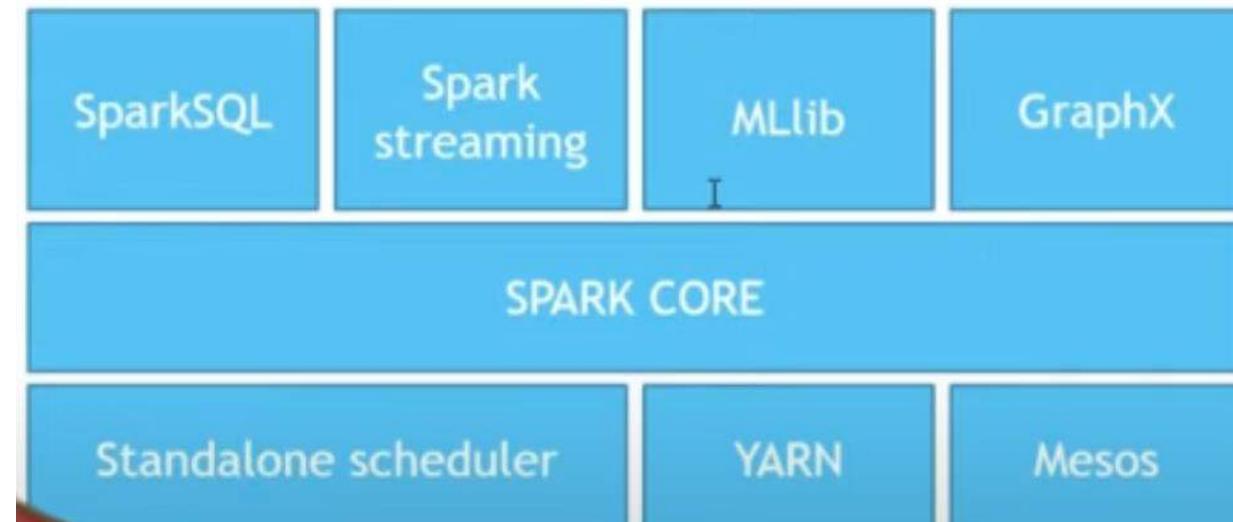
Standalone Scheduler : Install  
Spark on empty set of  
machines.

# Storage of Sp

The screenshot shows a YouTube video player with the following details:

- Title:** [HINDI] INTRODUCTION TO APACHE SPARK
- Views:** 138K
- Published:** 5 years ago
- Description:** Apache Spark Tutorial Hindi
- Thumbnail:** A man speaking into a microphone.
- Player Controls:** Back, forward, play/pause, volume, and a progress bar showing 5:26 / 13:05.
- Sidebar:** A sidebar titled "Apache Spark Tutorial Hindi" lists four related videos:
  - [HINDI] INTRODUCTION TO APACHE SPARK (TG117 Hindi)
  - [Hindi] Spark RDD, DAG, Spark Architecture | Part - I (TG117 Hindi)
  - [Hindi] Spark RDD, DAG, Spark Architecture | Part II (TG117 Hindi)
  - Hindi | Apache spark 2.3.0 Installation on Ubuntu (TG117 Hindi)
- Browser Tab:** The browser tab shows multiple tabs, including "spark sc [Hindi]", "[Hindi] X", "[Hindi] X", "Apache", "Home", "MU SLC", "Mail - P", "Questio", "Questio", "Questio", and "Questio".
- OS Taskbar:** The taskbar at the bottom shows various application icons and system status.

# Unified Stack:



- **Base layer** = Cluster Managers
- **Middle layer** = Spark Core (Engine of spark)
  - **Spark Core** = It will manage:
  - Task Scheduling, Distributing, Monitoring applications
- **Top Layer** = Own components of spark to perform various operations

| SL.NO | Hadoop                                                                                                    | Apache Spark                                                                                                                                           |
|-------|-----------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1     | It does not have its own components to handle : structured data (Hive), machine learning (Install Mahout) | It has its own components to perform various processing types.                                                                                         |
| 2     | Lot of time required to install, deploy & maintain                                                        | Installation Time, Deployment time, Maintenance time is reduced.                                                                                       |
| 3     | Not much suitable                                                                                         | If new application to be developed – it should perform various workloads like query streaming, ML, graph data processing etc.<br>So spark is suitable. |



## Spark Components

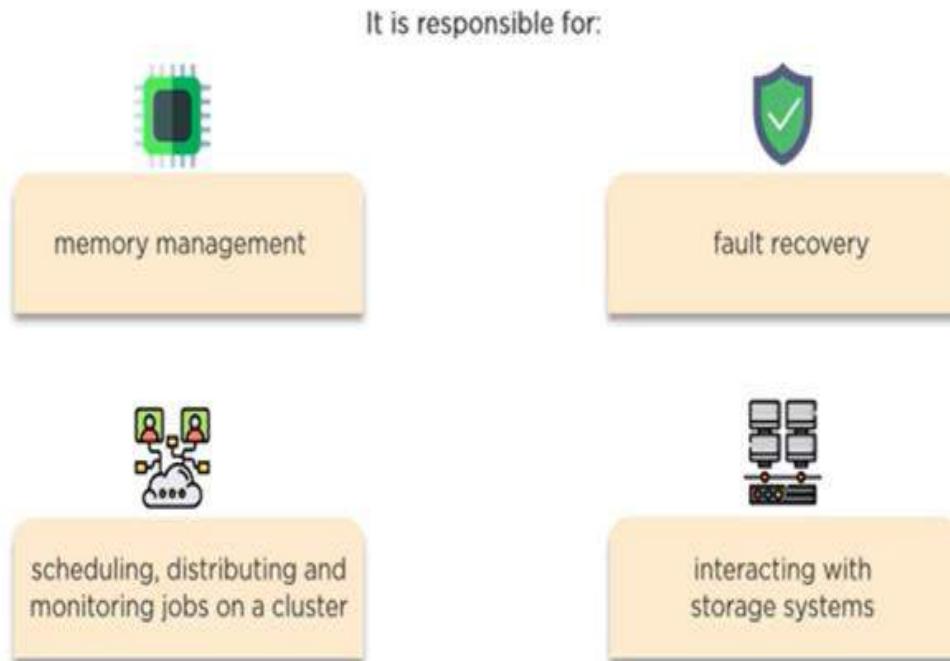
- Spark components are what make Apache Spark fast and reliable. A lot of these Spark components were built to resolve the issues that cropped up while using Hadoop MapReduce. Apache Spark has the following components:
  - **Spark Core**
  - **Spark SQL**
  - **Spark Streaming**
  - **Spark GraphX**
  - **Mlib (Machine Learning)**

# Spark Core:

- ▶ Contains basic functionality of Spark. (task scheduling, memory management, fault recovery, interacting with storage systems)
- ▶ Home to API that defines RDDs

# Spark Core:

Spark Core is the base engine for large-scale parallel and distributed data processing



## 1. Spark Core

- *Spark Core* is the base engine for large<sup>27</sup>-scale **parallel** and **distributed** data processing.
- The core is the distributed execution engine and the **Java, Scala, and Python APIs** offer a platform for distributed ETL application development.
- Further, additional libraries which are built on the core allow **diverse workloads** for streaming, SQL, and machine learning.
- It is responsible for:
  - ✓ Memory management and fault recovery
  - ✓ Scheduling, distributing and
  - ✓ monitoring jobs on a cluster
  - ✓ Interacting with storage systems

# Spark Core:

28

Spark Core is embedded with RDDs (Resilient Distributed Datasets), an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel



These are operations (such as map, filter, join, union) that are performed on an RDD that yields a new RDD containing the result

These are operations (such as reduce, first, count) that return a value after running a computation on an RDD

# What are RDDs?

- RDD or (**Resilient Distributed Data set**) is a fundamental **data structure** in Spark.
- The term **Resilient** defines the ability that generates the data automatically or data **rolling back** to the **original state** when an unexpected calamity occurs with a probability of data loss.
- RDD does **not need** something like **Hard Disks** or any other secondary storage. **It needs only RAM**.
- *RDD is not a Distributed File System instead it is Distributed Memory System.*
- Data can be loaded from any source to Apache Spark like Hadoop, HBase, Hive, SQL, S3
- They can process any type of data such as Structured, Unstructured, Semi-Structured data.



# Operations performed on RDDs

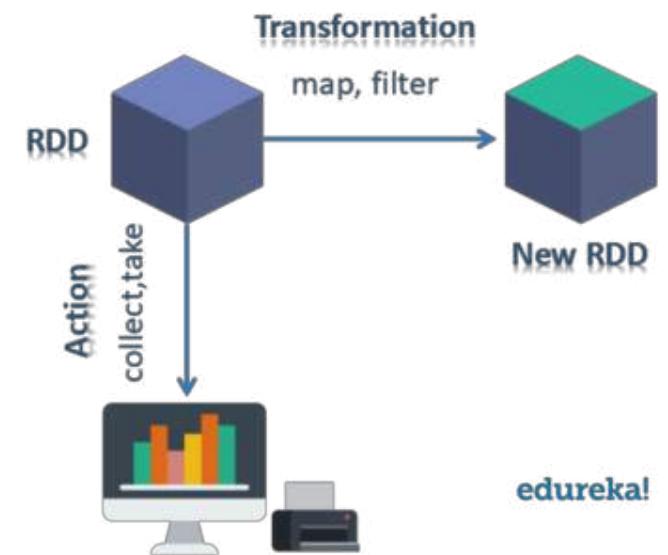
## Transformations:

- **filter, access** and **modify** the data in parent RDD to generate a **successive RDD**
- The new RDD returns a pointer to the previous RDD ensuring the dependency between them.
- Transformations are **Lazy Evaluations**
- **1. Narrow Transformations**

- `map()` , `filter()`
- `flatMap()`, `partition()`
- `mapPartitions()`

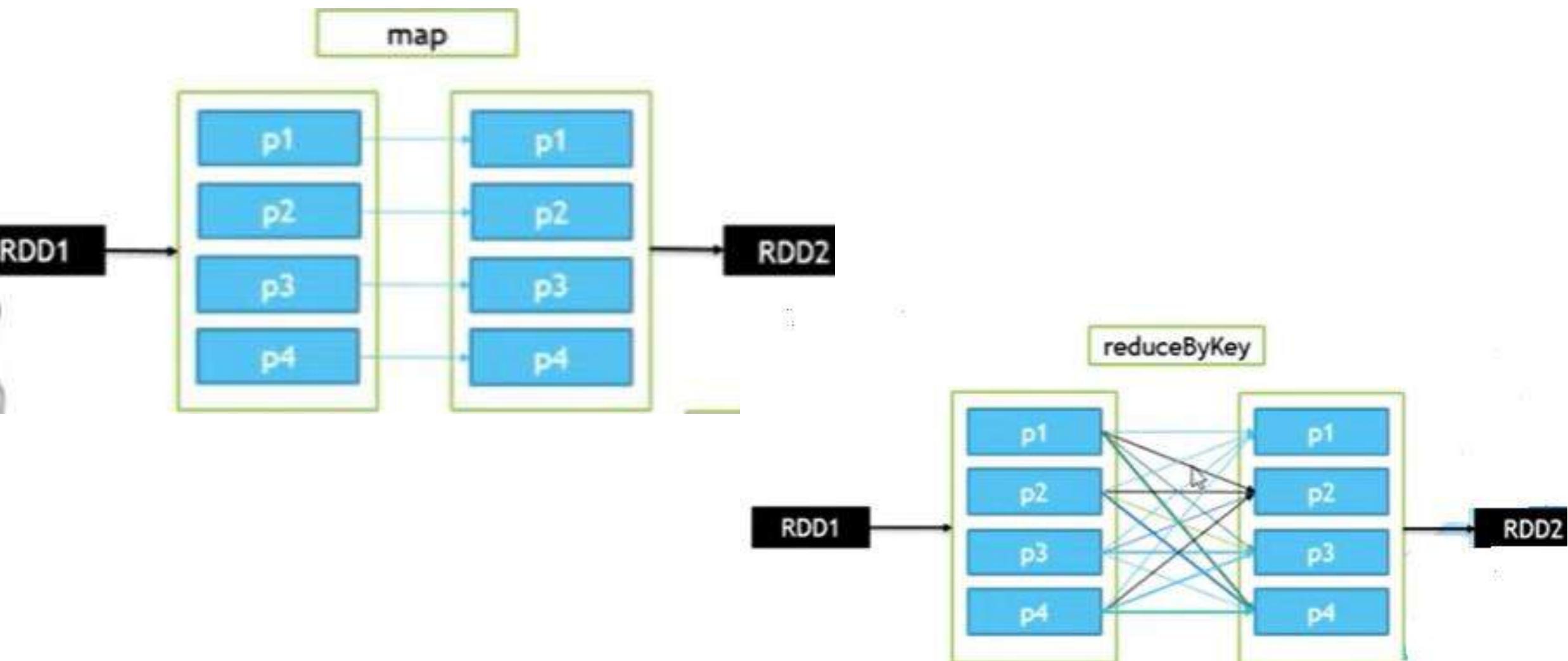
## 2. Wide Transformations

- `reduceBy()`
- `union()`
- `Intersection()`
- `Join()`



**Actions:** Actions instruct Apache Spark to apply **computation** and pass the result or an exception back to the driver RDD. Few of the actions include:**collect()**, **count()**, **take()**, **first()**

# Narrow & Wide Transformations



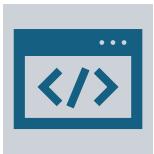
# RDD: Resilient Distributed Dataset



RDD Once done – It is **immutable** but can overridden

## Transformations :

```
var a = sc.textFile("hdfs://.....")
var b = a.flatMap(....)
var c = b.distinct(....)
```



If u want to **modify “data”** which is already there in specific RDD it cant be done , instead need to make **“new RDD”**



If data to be overridden –it can be successfully performed (means data from another file can be called in to current RDD, once data is overridden the content of current RDD will be updated with new called file data)

- Here **a** is “**RDD**”
- **Transformation** = Process of making RDD's
  - In above example; b & c are new RDD.
  - Flatmap = Is Transformation
- **Actions** = Processing the data inside the cluster

# RDD:

- ▶ Fault tolerant distributed dataset.
- ▶ Lazy Evaluation
- ▶ Caching
- ▶ In memory computation
- ▶ Immutability
- ▶ Partitioning

# RDD

- Unless Actions are not called no Transformations get executed – **lazy Evaluation**.
- **RDD (2 operations)**
  - **Transformations** = process of **making chain of RDD's**
  - Ex: From a RDD making b RDD
  - During Transformations = **No executions take place** (in cluster)
  - **Actions** = Actual Executions take place
  - During this time of transformations – **spark makes (RDD's Map) using DAG** (which RDD made – sequence of RDD's).

Transformations :

```
var a = sc.textFile("hdfs://...../abc.txt") --> RDD0
var b = a.filter(....) --> RDD1
var c = b.distinct(....) --> RDD2
```

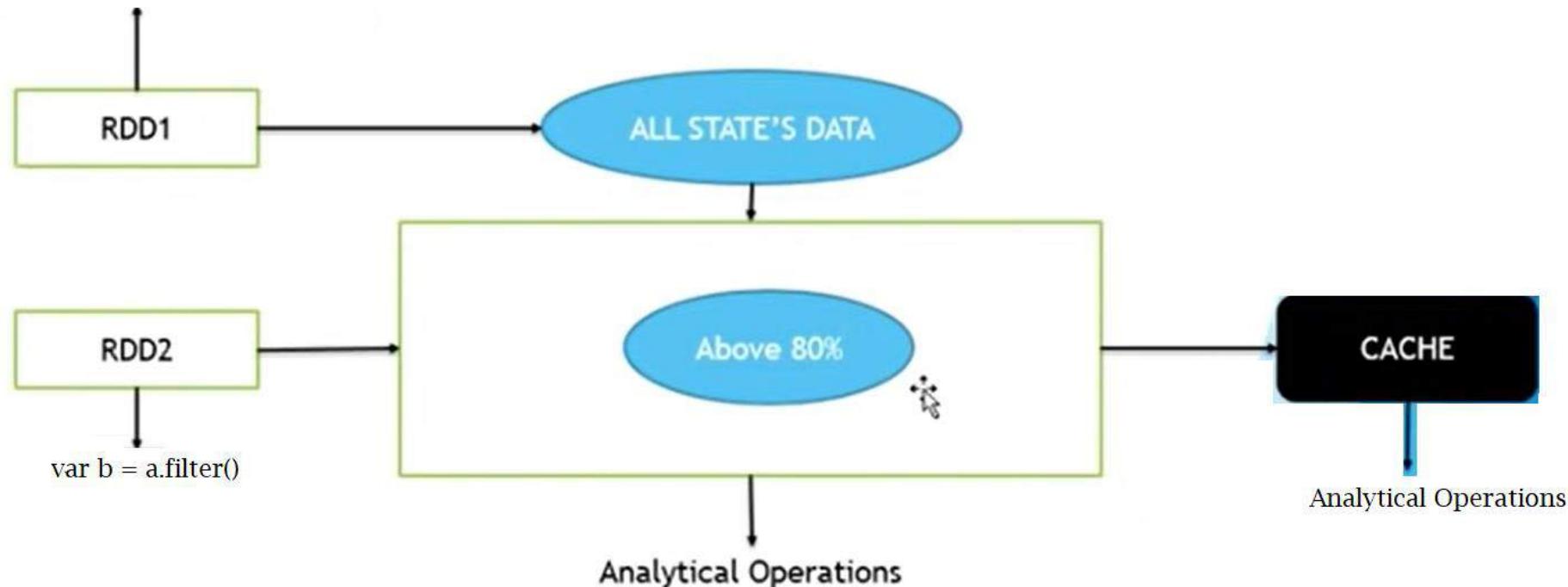
Actions

```
c.collect()
```

# Caching:

- If any modifications to be made in data we need to make new RDD always .
- Dataset used frequently is transferred to cache.

```
Var a = sc.textFile(hdfs://...)
```

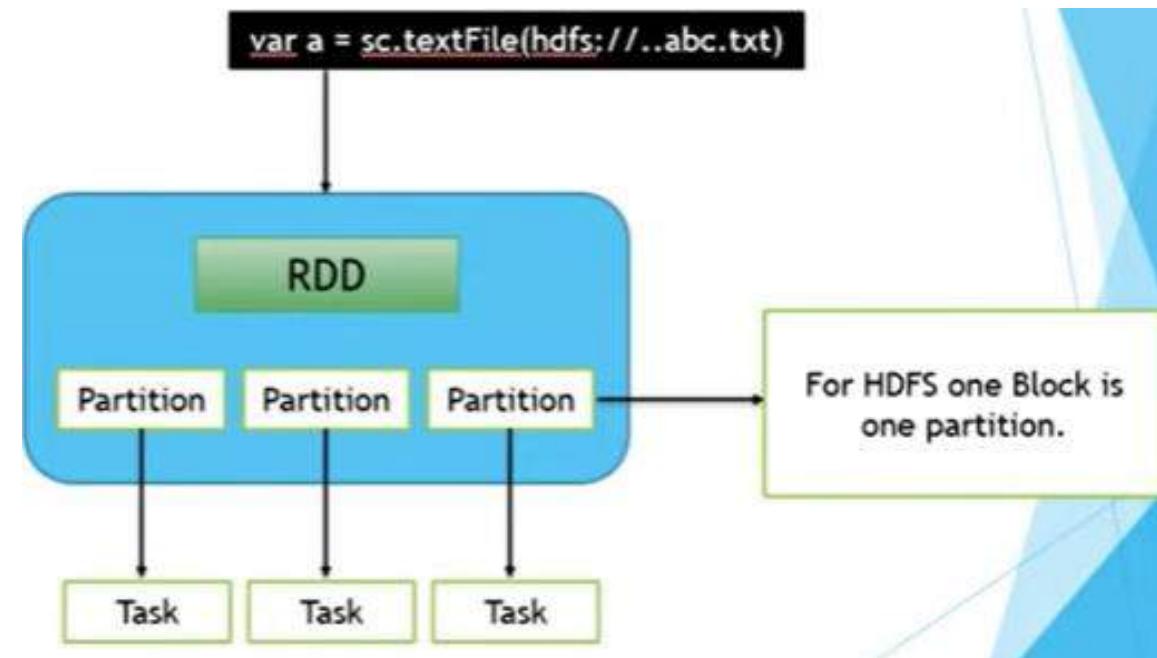


## Immutable:

- If any modifications to be made in data we need to make new RDD always (Immutable).
- Means RDD's are immutable(we cant change RDD).

# Partitioning

- ▶ Data is split up into partitions.
- ▶ Partition size depends on data source you are using.
- ▶ For HDFS one Block is one Partition.
- ▶ Single partition -- > Single Task

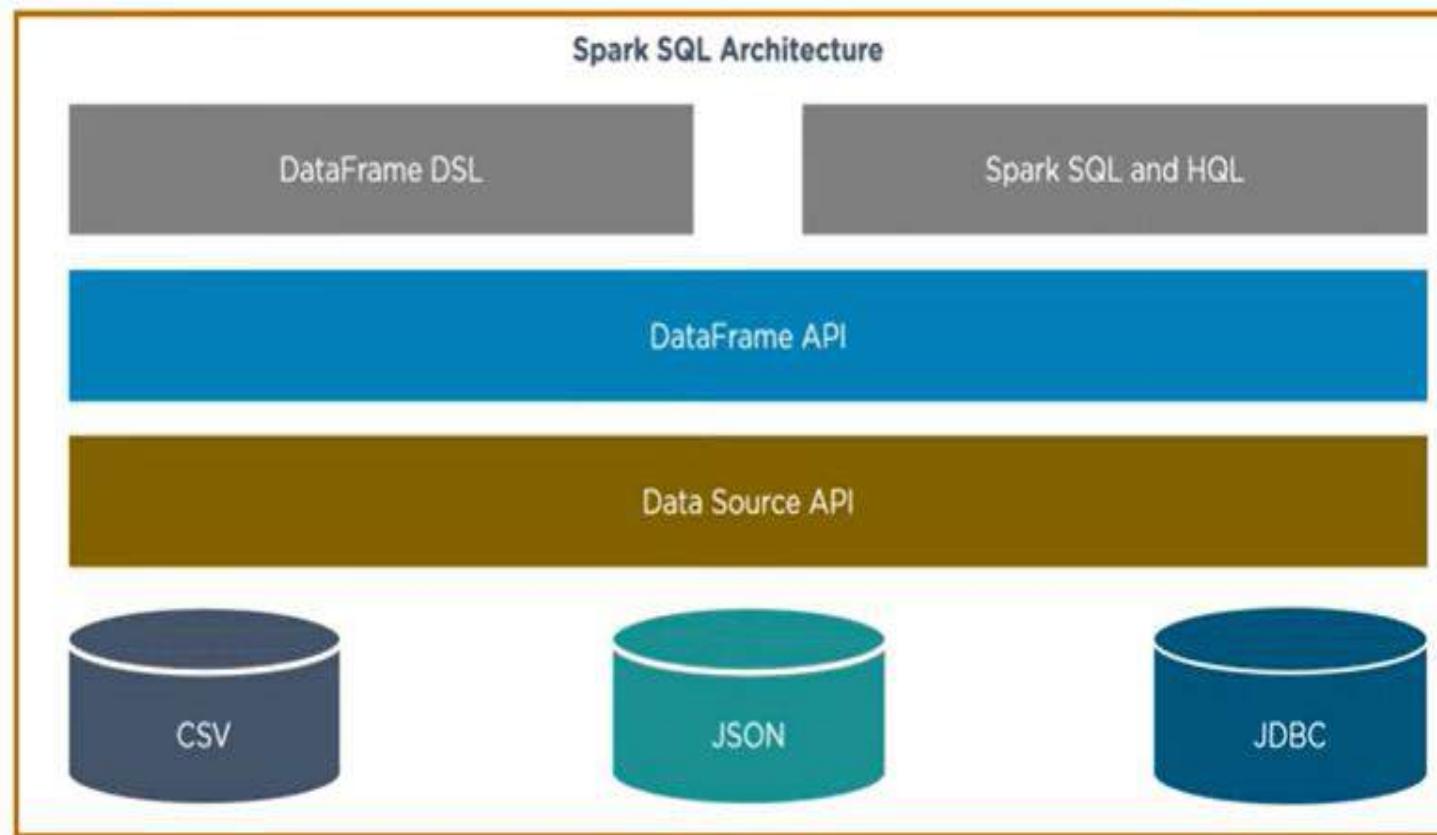


# Spark SQL – Structured Data

- ▶ Spark's package for working with structured data.
- ▶ Supports many sources of data including Hive tables, parquet, JSON.
- ▶ Allows developers to intermix SQL with programmatic data manipulations supported by RDDs in Python, Scala and Java.
- ▶ Shark was older SQL on Spark project.

# Spark SQL:

Spark SQL framework component is used for structured and semi-structured data processing



## 2. Spark SQL

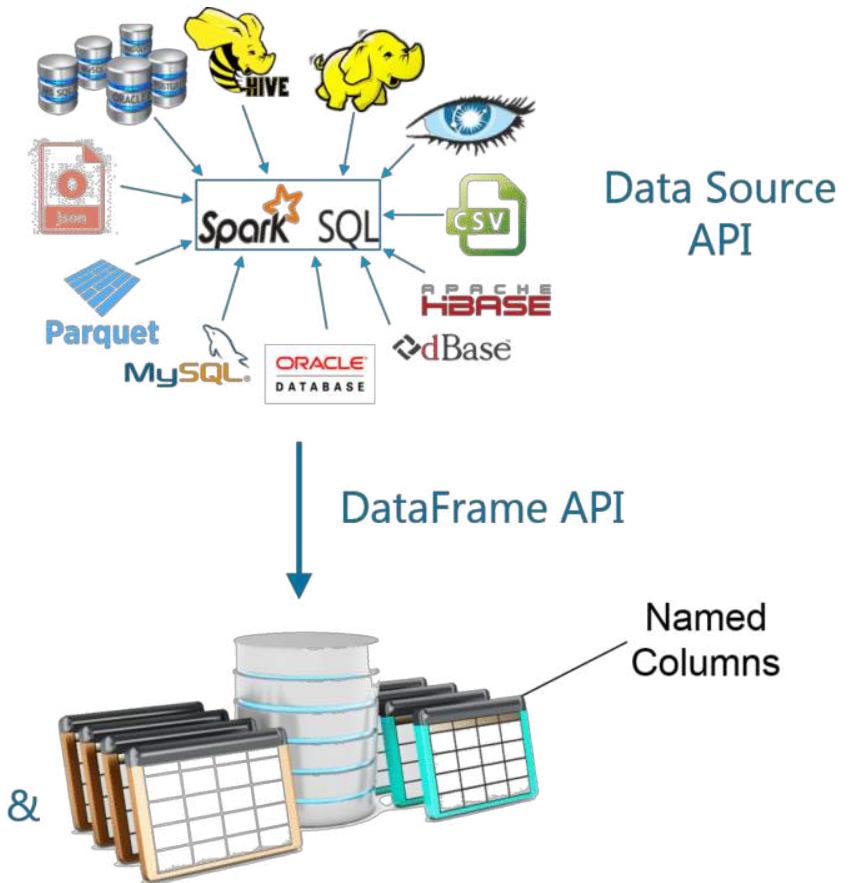
Spark SQL is a new module in Spark which integrates relational processing with Spark's functional programming API. It supports querying data either via SQL or via the Hive Query Language.

Spark SQL integrates relational processing with Spark's functional programming. Further, it provides support for various data sources and makes it possible to weave SQL queries with code transformations thus resulting in a very powerful tool.

The following are the four libraries of Spark SQL.

1. Data Source API
2. DataFrame API
3. Interpreter & Optimizer
4. SQL Service

# Spark SQL



**Figure:** The flow diagram represents a Spark SQL process using all the four libraries in sequence

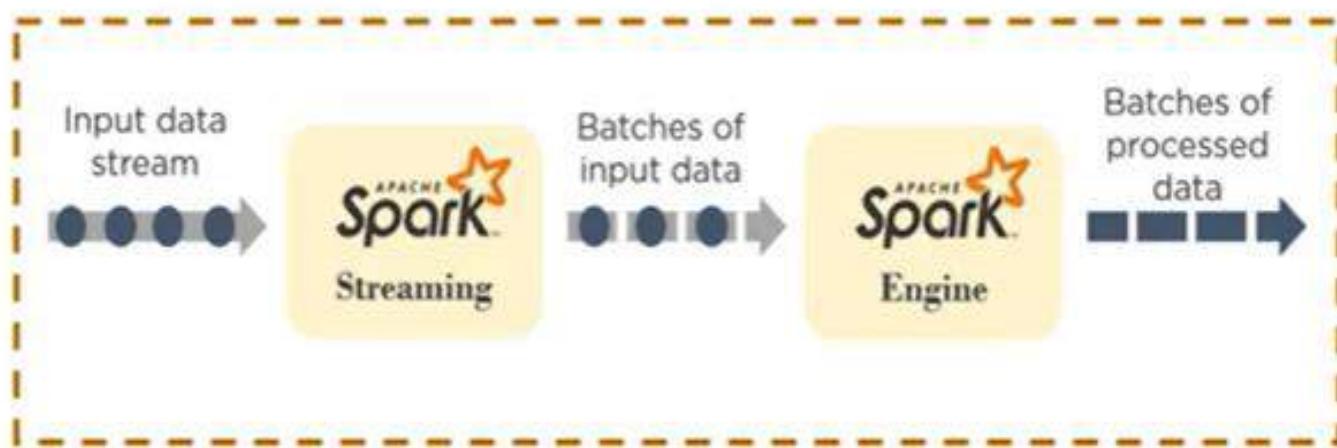
## Spark Streaming:

- ▶ Enables processing of live streams of data e.g. log files generated by production web servers.
- ▶ APIs are quite similar to spark core's RDD APIs.

# Spark Streaming:

Spark Streaming is a lightweight API that allows developers to perform batch processing and real-time streaming of data with ease

Provides secure, reliable, and fast processing of live data streams



### 3. Spark Streaming

*Spark Streaming* is the component of Spark which is used to process real-time streaming data. Thus, it is a useful addition to the core Spark API. It enables high-throughput and fault-tolerant stream processing of live data streams.

# MLLib

- ▶ Provides multiple types of machine learning algorithms.

# Spark MLlib:

MLlib is a low-level machine learning library that is simple to use, is scalable, and compatible with various programming languages

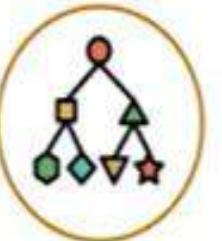
MLlib eases the deployment and development of scalable machine learning algorithms



It contains machine learning libraries that have an implementation of various machine learning algorithms



Clustering



Classification



Collaborative Filtering

## 4. MLlib (Machine Learning)

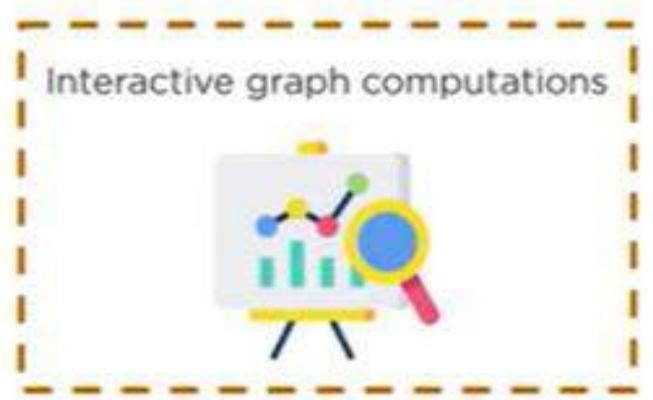
MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark.

# Graphx

- ▶ Library for manipulating Graphs.
- ▶ Extends Spark RDD API.
- ▶ Provides various operators for manipulating Graphs.

# Spark GraphX:

GraphX is Spark's own Graph Computation Engine and data store



## 5. GraphX

- *GraphX* is the Spark API for graphs and graph-parallel computation. Thus, it extends the Spark RDD with a Resilient Distributed Property Graph.
- The property graph is a directed multigraph which can have multiple edges in parallel. Every edge and vertex have user defined properties associated with it.
- Here, the parallel edges allow multiple relationships between the same vertices. At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge.



# Transformations

| Function       | Description                                                                                           |
|----------------|-------------------------------------------------------------------------------------------------------|
| map()          | Returns a new RDD by applying the function on each data element                                       |
| filter()       | Returns a new RDD formed by selecting those elements of the source on which the function returns true |
| reduceByKey()  | Aggregates the values of a key using a function                                                       |
| groupByKey()   | Converts a (key, value) pair into a (key, <iterable value>) pair                                      |
| union()        | Returns a new RDD that contains all elements and arguments from the source RDD                        |
| intersection() | Returns a new RDD that contains an intersection of the elements in the datasets                       |



# Actions

| Function           | Description                                                                    |
|--------------------|--------------------------------------------------------------------------------|
| count()            | Gets the number of data elements in an RDD                                     |
| collect()          | Gets all the data elements in an RDD as an array                               |
| reduce()           | Aggregates data elements into an RDD by taking two arguments and returning one |
| take(n)            | Fetches the first $n$ elements of an RDD                                       |
| foreach(operation) | Executes the operation for each data element in an RDD                         |
| first()            | Retrieves the first data element of an RDD                                     |

# Spark Context

- 1. Entry Point:** SparkContext is the entry point for Spark functionality in any Spark application.
- 2. Connection to Cluster:** It establishes a connection to the Spark cluster.
- 3. Resource Management:** SparkContext manages the resources allocated to the Spark application on the cluster.
- 4. RDD Creation:** It is used to create RDDs (Resilient Distributed Datasets), which are the fundamental data structures in Spark.
- 5. Driver Program Coordination:** The SparkContext is created by the driver program to coordinate the execution of Spark jobs on the cluster.
- 6. Access to Services:** It provides access to various Spark services like broadcast variables, accumulators, and shared variables.
- 7. Configuration:** SparkContext allows setting various configurations for Spark application execution.
- 8. Lifecycle Management:** It manages the lifecycle of the Spark application, including starting, running, and stopping the application.
- 9. Fault Tolerance:** SparkContext ensures fault tolerance by monitoring the execution of tasks and recovering from failures.
- 10. Parallel Operations:** SparkContext enables parallel operations on distributed data by leveraging the capabilities of the Spark cluster.



# How to Create RDD in Spark?

There are following three processes to create Spark RDD.

1. Using **parallelized collections**
2. From **external datasets** (viz . other external storage systems like a shared file system, HBase or HDFS)
3. From existing **Apache Spark RDDs**

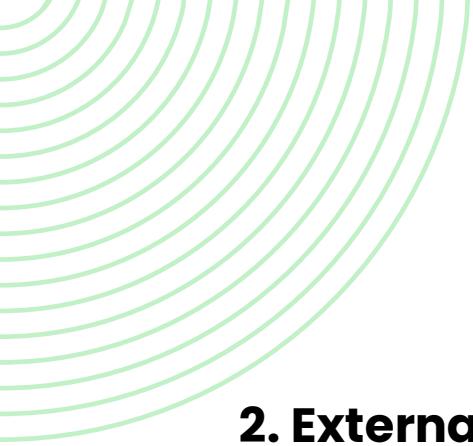
## 1. Parallelized Collections

create parallelized collections by calling **parallelize method of SparkContext interface** on the existing collection of driver program in Java, Scala or Python.

### Example:

To hold the numbers 2 to 6 as parallelized collection

```
val collection = Array(2, 3, 4, 5, 6)
val prData =
 spark.sparkContext.parallelize(collection)
```



# How to Create RDD in Spark?

## 2. External Datasets

Apache Spark can create distributed datasets from any **Hadoop supported file storage** which may include:

- Local file system
- HDFS
- Cassandra
- HBase
- Amazon S3

**Spark supports file formats like**

- Text files
- Sequence Files
- CSV
- JSON
- Any Hadoop Input Format

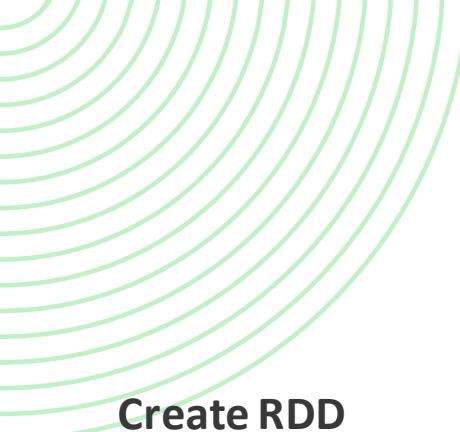


# How to Create RDD in Spark?

## 3. From Existing RDDs

RDD is immutable; hence you can't change it. However, using transformation, you can create a new RDD from an existing RDD. As no change takes place due to mutation, it maintains the consistency over the cluster. Few of the operations used for this purpose are:

- map
- filter
- count
- distinct
- flatmap



## Practical demo of RDD operations

### Create RDD

```
scala> val rdd1 = sc.parallelize(List(23, 45, 67, 86, 78, 27, 82, 45, 67, 86))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:27
```

Here, sc denotes SparkContext and each element is copied to form RDD.

### Read result

We can read the result generated by RDD by using the collect operation.

```
scala> rdd1.collect
res0: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86)

scala> ■
```

## Count

The count action is used to get the total number of elements present in the particular RDD.

```
scala> rdd1.count
res1: Long = 10
```

```
scala> █
```

## Distinct

Distinct is a type of transformation that is used to get the unique elements in the RDD.

```
scala> rdd1.distinct.collect
res3: Array[Int] = Array(82, 86, 78, 27, 23, 45, 67)
```

```
scala> █
```

## Filter

Filter transformation creates a new dataset by selecting the elements according to the given

```
scala> rdd1.filter(x => x < 50).collect
res5: Array[Int] = Array(23, 45, 27, 45)
```

```
scala> █
```

## **sortBy**

sortBy operation is used to arrange the elements in ascending order when the condition is true and in descending order when the condition is false.

```
scala> rdd1.sortBy(x => x, true).collect
res6: Array[Int] = Array(23, 27, 45, 45, 67, 67, 78, 82, 86, 86)
```

```
scala> rdd1.sortBy(x => x, false).collect
res7: Array[Int] = Array(86, 86, 82, 78, 67, 67, 45, 45, 27, 23)
```

```
scala> ■
```

## **Reduce**

Reduce action is used to summarize the RDD based on the given formula.

```
scala> rdd1.reduce((x, y) => x + y)
res8: Int = 606
```

```
scala> ■
```

## **Map**

Map transformation processes each element in the RDD according to the given condition and creates a new RDD.

```
scala> rdd1.map(x => x + 1).collect
res9: Array[Int] = Array(24, 46, 68, 87, 79, 28, 83, 46, 68, 87)
```

```
scala> ■
```

## Union, intersection, and cartesian

Let's create another RDD.

```
val rdd2 = sc.parallelize(List(25,73, 97, 78, 27, 82))
```

- Union operation combines all the elements of the given two RDDs.
- Intersection operation forms a new RDD by taking the common elements in the given RDDs.
- Cartesian operation is used to create a cartesian product of the required RDDs.

```
scala> rdd1.union(rdd2).collect
res12: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86, 25, 73, 97, 78
, 27, 82)
```

```
scala> rdd1.intersection(rdd2).collect
res13: Array[Int] = Array(82, 78, 27)
```

```
scala> rdd1.cartesian(rdd2).collect
res14: Array[(Int, Int)] = Array((23,25), (23,73), (23,97), (45,25), (45,73), (4
5,97), (67,25), (67,73), (67,97), (86,25), (86,73), (86,97), (78,25), (78,73),
(78,97), (23,78), (23,27), (23,82), (45,78), (45,27), (45,82), (67,78), (67,27),
(67,82), (86,78), (86,27), (86,82), (78,78), (78,27), (78,82), (27,25), (27,73),
(27,97), (82,25), (82,73), (82,97), (45,25), (45,73), (45,97), (67,25), (67,73)
, (67,97), (86,25), (86,73), (86,97), (27,78), (27,27), (27,82), (82,78), (82,27
), (82,82), (45,78), (45,27), (45,82), (67,78), (67,27), (67,82), (86,78), (86,2
7), (86,82))
```

```
scala> ■
```

## First

First is a type of action that always returns the first element of the RDD.

```
scala> rdd1.first()
res15: Int = 23
```

```
scala> █
```

## Take

Take action returns the first n elements in the RDD.

```
scala> rdd1.take(5)
res16: Array[Int] = Array(23, 45, 67, 86, 78)
scala> █
```



# Pair RDD Operations

- Pair RDDs are a unique class of data structure in Spark that take the form **of key-value pairs**
- most **real-world data** is in the form **of Key/Value pairs**, Pair RDDs are practically employed more frequently.
- The terms "key" and "value" are different by the Pair RDDs. The **value is data**, whereas the **key is an identifier**.

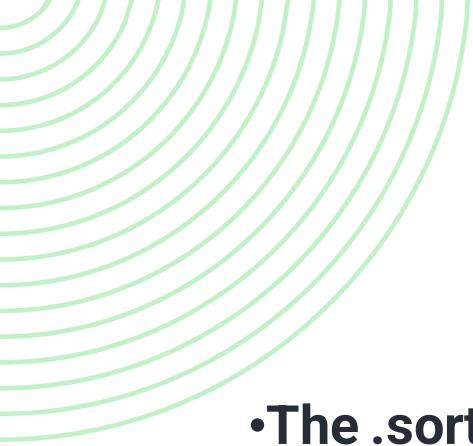
## Transformations in Pair RDDs

We must utilize operations that use keys and values since Pair RDDs are built from many tuples.

Following are the widely used Transformation on a Pair RDD:

### 1. The `.reduceByKey()` Transformation

For each key in the data, the `.reduceByKey()` transformation runs multiple parallel operations, **combining the results for the same keys**. **The task is carried out using a lambda or anonymous function**. Since it is a transformation, the outcome is an RDD.



# Pair RDD Operations

- **The .sortByKey() Transformation**

Using the keys from key-value pairs, the.sortByKey() transformation **sorts the input data in ascending or descending order**. It returns a unique RDD as a result.

- **The .groupByKey() Transformation**

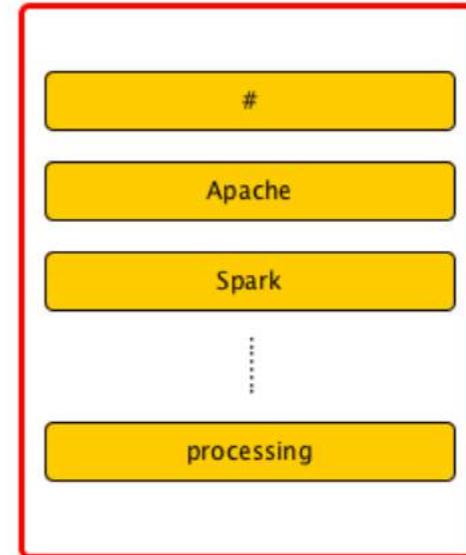
The.groupByKey() transformation **groups all values in the given data with the same key**. As a result, a **new RDD is returned**. For instance, the.groupByKey() function will be useful if we need to extract all the Cultural Members from a list of committee members.

# Actions in Pair RDDs

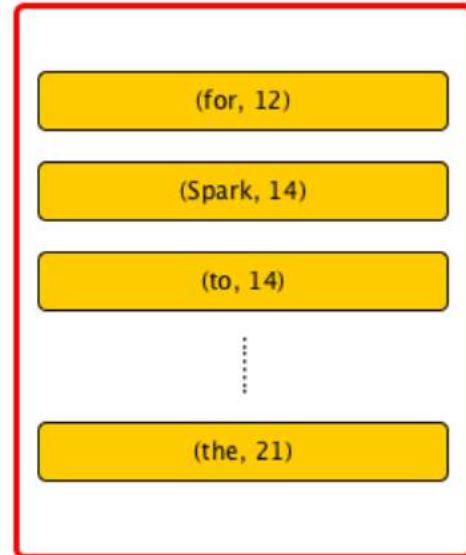
## The countByKey() Action

The number of values linked with each key in the provided data is counted using the.`countByKey()` action. This operation returns a dictionary, which can be iterated using loops to retrieve the keys and values. We can also utilize dictionary methods like`.keys()`,`.values()`, and`.items` because the result is a dictionary ()�.

RDD of Strings



RDD of Pairs



## **DataFrame:**

- A Data Frame is used for storing data in tables.
- It is equivalent to a table in a relational database but with richer optimization.
- It is a data abstraction and domain-specific language (DSL) applicable to a structure and semi-structured data.
- It is a distributed collection of data in the form of named column and row.
- It has a matrix-like structure whose column may be different types (numeric, logical, factor, or character ).
- We can say the data frame has a two-dimensional array-like structure where each column contains the value of one variable and row contains one set of values for each column.
- It combines features of lists and matrices.

## **RDD:**

- It is the representation of a set of records, an immutable collection of objects with distributed computing.
- RDD is a large collection of data or RDD is an array of references for partitioned objects.
- Each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster.
- RDDs are fault-tolerant i.e. self-recovered/recomputed in the case of failure.
- The dataset could be data loaded externally by the users which can be in the form of JSON file, CSV file, text file or database via JDBC with no specific data structure.

# Differentiation: RDD vs Datasets vs DataFrame

| Basis               | RDD                                                                                    | Datasets                                                                                           | DataFrame                                                                      |
|---------------------|----------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| Inception Year      | RDD came into existence in the year 2011.                                              | Datasets entered the market in the year 2013.                                                      | DataFrame came into existence in the year 2015.                                |
| Meaning             | RDD is a collection of data where the data elements are distributed without any schema | Datasets are distributed collections where the data elements are organized into the named columns. | Datasets are basically the extension of DataFrames with added features         |
| Optimization        | In case of RDDs, the developers need to manually write the optimization codes.         | Datasets use catalyst optimizers for optimization.                                                 | Even in the case of DataFrames, catalyst optimizers are used for optimization. |
| Defining the Schema | In RDDs, the schema needs to be defined manually.                                      | The schema is automatically defined in case of Datasets                                            | The schema is automatically defined in DataFrame                               |



## Advantages of RDD

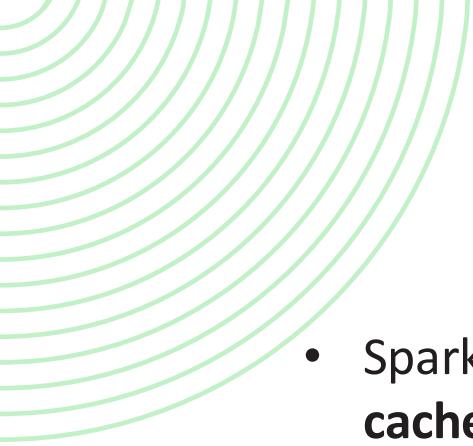
There are multiple advantages of RDD in Spark. We have covered few of the important ones here

- RDD aids in increasing the **execution speed** of Spark.
- RDDs are the **basic unit of parallelism** and hence help in achieving the **consistency of data**.
- RDDs help in **performing and saving the actions** separately
- They are **persistent** as they can be used repeatedly.



# Limitation of RDD

- There is **no input optimization** available in RDDs
- One of the biggest limitations of RDDs is that the **execution process does not start instantly.**
- **No changes can be made** in RDD once it is **created**.
- **RDD lacks enough storage memory.**
- The **run-time type safety** is absent in RDDs.



# RDD Persistence

- Spark RDD persistence is an **optimization technique** which **saves** the result of RDD evaluation in **cache memory**.
- Spark provides a convenient way to work on the dataset by **persisting it in memory across operations**.
- we can **also reuse** them in other tasks on that dataset.
- We can use either **persist()** or **cache()** method to mark an RDD **to be persisted**.
- In any case, if the partition of an RDD is lost, it will automatically be **recomputed** using the **transformations that originally created it**.
- There is an **availability of different storage levels** which are used to **store persisted RDDs**.
- Use these levels by passing a **StorageLevel** object (Scala, Java, Python) to persist().
- The **cache()** method is used for the **default storage level**, which is **StorageLevel.MEMORY\_ONLY**.



## Example

RDD1: Contains a list of numbers [1, 2, 3, 4, 5].

RDD2: Created by doubling each element of RDD1.

RDD3: Created by squaring each element of RDD2.

```
Create RDD1
```

```
rdd1 = sc.parallelize([1, 2, 3, 4, 5])
```

```
Create RDD2 by performing a transformation on RDD1
```

(doubling each element)

```
rdd2 = rdd1.map(lambda x: x * 2)
```

```
Create RDD3 by performing a transformation on RDD2
```

(squaring each element)

```
rdd3 = rdd2.map(lambda x: x ** 2)
```

Now, let's say we perform an action on RDD3, such as counting the number of elements:

```
Perform an action on RDD3 (e.g., count the number of elements)
```

```
count = rdd3.count()
```



## Example

Here's an example of how you can cache RDD3:

```
Assuming RDD3 is already created
rdd3.persist()
```

```
Perform actions on RDD3
count = rdd3.count()
Other operations on RDD3...
```

- Intermediate results are stored in memory or disk, leading to **faster and more efficient** computations when accessing RDD3 multiple times.

**Note: rdd.cache() is same as rdd.persist()**



# Need of Persistence in Apache Spark

- In Spark, we can use some RDD's **multiple times**.
- We repeat the same process of **RDD evaluation** each time it required or brought into action.
- This task can be **time and memory consuming**, especially for iterative algorithms that look at data multiple times.
- **To solve** the problem of **repeated computation** the technique of persistence came into the picture.
- There are some **advantages of RDD caching** and persistence mechanism in spark. It makes the whole system
  - 1. Time efficient
  - 2. Cost efficient
  - 3. Lessen the execution time.

# The set of storage levels:

| Storage Level                             | Description                                                                                                                                                                                         |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MEMORY_ONLY                               | It stores the RDD as deserialized Java objects in the JVM. This is the default level. If the RDD doesn't fit in memory, some partitions will not be cached and recomputed each time they're needed. |
| MEMORY_AND_DISK                           | It stores the RDD as deserialized Java objects in the JVM. If the RDD doesn't fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.             |
| MEMORY_ONLY_SER<br>(Java and Scala)       | It stores RDD as serialized Java objects ( i.e. one-byte array per partition). This is generally more space-efficient than deserialized objects.                                                    |
| MEMORY_AND_DISK_SER<br>(Java and Scala)   | It is similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them.                                                                                |
| DISK_ONLY                                 | It stores the RDD partitions only on disk.                                                                                                                                                          |
| MEMORY_ONLY_2,<br>MEMORY_AND_DISK_2, etc. | It is the same as the levels above, but replicate each partition on two cluster nodes.                                                                                                              |
| OFF_HEAP (experimental)                   | It is similar to MEMORY_ONLY_SER, but store the data in off-heap memory. The off-heap memory must be enabled.                                                                                       |



# Persistence Storage Levels

- MEMORY\_ONLY (default) – same as cache

**rdd.persist(StorageLevel.MEMORY\_ONLY) or rdd.persist()**

- MEMORY\_AND\_DISK – Stores partitions on disk which do not fit in memory (This is also called Spilling)

**rdd.persist(StorageLevel.MEMORY\_AND\_DISK)**

- DISK\_ONLY – Stores all partitions on the disk

**rdd.persist(StorageLevel.DISK\_ONLY)**

## -MEMORY\_ONLY\_SER and MEMORY\_AND\_DISK\_SER

- Persisting the **RDD in a serialized** (binary) form helps to **reduce the size of the RDD**, thus making space for more RDD to be persisted in the cache memory. So these two memory formats are **space-efficient**.
- They are **not time-efficient** because we need **to incur the cost of time** involved in **deserializing the data**.



# Partition Replication

Stores the partition on two nodes.

**DISK\_ONLY\_2**

**MEMORY\_AND\_DISK\_2**

**MEMORY\_ONLY\_2**

**MEMORY\_AND\_DISK\_SER\_2**

**MEMORY\_ONLY\_SER\_2**

- These options stores a replicated copy of the RDD into some other Worker Node's cache memory as well.
- it helps to recompute the RDD if the other worker node goes down.

## Unpersisting the RDD

- To stop persisting and remove from memory or disk
- To change an RDD's persistence level

**rdd.unpersist()**

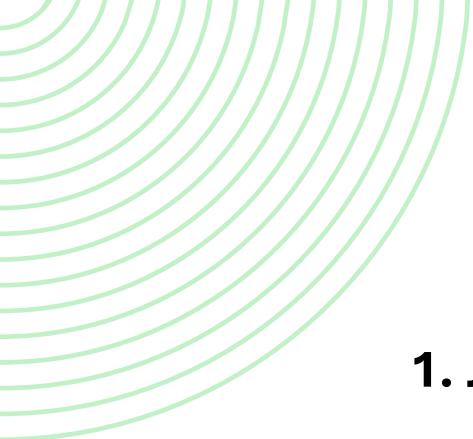
# Representation of the Storage level

| Storage Level       | Space used | CPU time | In memory | On-disk | Serialized | Recompute some partitions |
|---------------------|------------|----------|-----------|---------|------------|---------------------------|
| MEMORY_ONLY         | High       | Low      | Y         | N       | N          | Y                         |
| MEMORY_ONLY_SER     | Low        | High     | Y         | N       | Y          | Y                         |
| MEMORY_AND_DISK     | High       | Medium   | Some      | Some    | Some       | N                         |
| MEMORY_AND_DISK_SER | Low        | High     | Some      | Some    | Y          | N                         |
| DISK_ONLY           | Low        | High     | N         | Y       | Y          | N                         |



# Serialization

- In distributed systems, **data transfer over the network is the most common task.**
- If this is **not handled efficiently**, you may end up facing numerous problems, like **high memory usage, network bottlenecks, and performance issues.**
- **Serialization** refers to **converting objects into a stream of bytes** and vice-versa (de-serialization) in an optimal way to transfer it over nodes of network or store it in a file/memory buffer.
- Spark provides **two serialization libraries** and modes are supported and configured through **spark.serializer** property.



# Two serialization libraries

## 1. Java serialization (default)

The serialization of a class is enabled by the class implementing the [java.io.Serializable](#) interface.

Java serialization is slow and leads to large serialized formats for many classes. We can fine-tune the performance by extending [java.io.Externalizable](#).

## 2. Kryo serialization (recommended by Spark)

- Kryo is a Java serialization framework that focuses on **speed, efficiency, and a user-friendly API**.
- Kryo has **less memory footprint**, which becomes very important when you are shuffling and caching a large amount of data.
- It is **not natively supported to serialize to the disk**.
- A class is never serialized **only object of a class is serialized**.



```
from pyspark import SparkConf, SparkContext
from pyspark.serializers import KryoSerializer

conf = SparkConf().setAppName("my_app").setMaster("local")

conf.set("spark.serializer", KryoSerializer.getName())

sc = SparkContext(conf=conf)

Now we can create and process RDDs using Kryo serialization

Create RDD1
rdd1 = sc.parallelize([1, 2, 3, 4, 5])
```



# Shared Variable in Spark

- Variables those we want to share **throughout our cluster**
  - Transformation functions passed on variable, it executes on the specific remote cluster node.
  - Usually, it works on separate copies of all the variables those we use in functions.
  - These specific variables are precisely copied to each machine.
  - on the remote machine, no updates to the variables sent back to the driver program.
  - Therefore, it would be inefficient to support general, read-write shared variables across tasks.

**Two types of shared variables, such as:**

1. Broadcast Variables
2. Accumulators



# Broadcast Variables:

- Broadcast variables allow you to efficiently **distribute read-only data to all the worker nodes** in a Spark cluster.
- They are useful when you have large datasets or lookup tables that need to be shared across all nodes but don't change frequently.
- Broadcast variables are **broadcasted to all the worker nodes** once and then cached locally on each node for future reference, which **reduces network overhead**.
- They are **immutable** and can be safely used in parallel operations.

```
Creating a broadcast variable
broadcast_var = sc.broadcast([1, 2, 3, 4, 5])

Accessing the broadcast variable value on each worker node
def process_data(element):
 broadcast_value = broadcast_var.value

 processed_data = [x * 2 for x in element if x in
 broadcast_value]
 return processed_data

Applying a transformation operation using the broadcast
variable
rdd = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
processed_rdd = rdd.map(process_data)
```



# Accumulators:

- Accumulators are variables that are only "added" to through **an associative and commutative operation** and are only "read" **by the driver program**.
- They are primarily used for **aggregating information across all the worker nodes** during parallel computations, such as **counting occurrences or summing values**.
- Accumulators are useful for tasks like **collecting statistics** or **monitoring the progress of a job**.

```
Creating an accumulator for counting occurrences
accum = sc.accumulator(0)
```

```
Defining a function to increment the accumulator
def process_data(element):
 global accum
 if 3 in element:
 accum += 1
 return element
```

```
Applying a transformation operation using the accumulator
rdd = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
processed_rdd = rdd.map(process_data)
```

```
Performing an action to trigger the accumulation
processed_rdd.count()
```

```
Accessing the value of the accumulator in the driver
program
print("Occurrences of 3:", accum.value)
```

## DAG: Directed Acyclic Graph

Spark creates a graph when you enter code in spark console.

When an action is called on Spark RDD, Spark submits graph to DAG scheduler.

Operators are divided into stages of task in DAG Scheduler.

The Stages are passed on Task Scheduler, Which launches task through CM.

# Importance of DAG in Spark

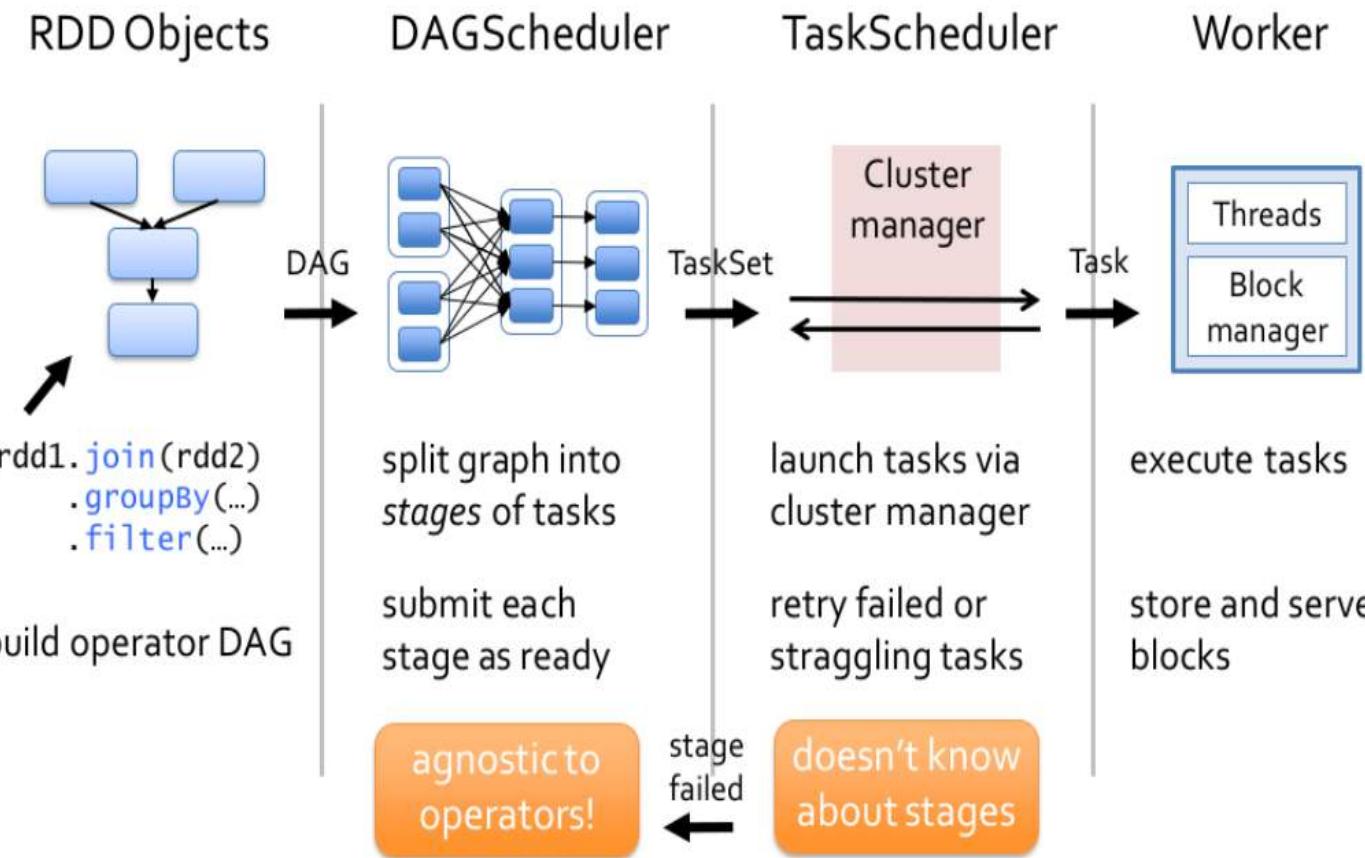
The DAG plays a critical role in this process by providing **a logical execution plan for the job.**

- The DAG breaks the job down into a **sequence of stages**, where **each stage** represents a **group of tasks** that can be **executed independently** of each other. The **tasks within each stage can be executed in parallel** across the machines.
- The DAG allows Spark to perform various optimizations, such as **pipelining, task reordering, and pruning unnecessary operations**, to improve the efficiency of the job execution.
- By breaking down the job into smaller stages and tasks, Spark can execute them in parallel and distribute them across a cluster of machines for faster processing.**

# DAG Scheduler

DAG Scheduler is responsible for **transforming a sequence of RDD transformations and actions** into a directed acyclic graph (DAG) of **stages and tasks**

- Stages will be created.
- Which task need to put in which stage is decided by DAG Scheduler.
- Stage list will be created by DAG scheduler and submitted to Task scheduler.
- Each task will be executed via Cluster manager and cluster manager (CM) will launch it.
- To perform particular task, it needs data and resources.
- Cluster manager will help task scheduler to provide information from where the data need to be fetched in order to execute the tasks list and required resources check will be done by cluster manager.
- Later on task scheduler requests the CM to provide some slot to execute task .
- Executer run on slave machine. TO execute particular task executer has required resources



DAGScheduler transforms a **logical execution plan** (RDD lineage of dependencies built using RDD transformations) to a **physical execution plan** (using stages).

# Terminologies in DAG: Stages, Tasks, Dependencies

To work with the DAG Scheduler in Spark, you need to understand the following concepts:

## Stages:

- A stage represents a set of tasks that can be executed in parallel.
- There are **two types of stages** in Spark: **shuffle stages** and **non-shuffle stages**.
- **Shuffle** stages involve the **exchange of data between nodes**, while non-shuffle stages do not.

## Tasks:

- A task represents a **single unit of work** that can be executed on a **single partition** of an **RDD**.
- Tasks are the smallest units of parallelism in Spark.

## Dependencies:

- The dependencies between RDDs determine **the order in which tasks are executed**.
- There are two types of dependencies in Spark: **narrow dependencies** and **wide dependencies**.
- **Narrow** dependencies indicate that **each partition of the parent RDD** is used by **at most one partition of the child RDD**.
- while **wide dependencies** indicate **that each partition of the parent RDD** can be used by **multiple partitions of the child RDD**.

# Example: DAG diagram



- By visualizing the DAG diagram, developers can better understand the logical execution plan of a Spark job and identify any **potential bottlenecks or performance issues**.

- The DAG diagram consists of five stages: Text RDD, Filter RDD, Map RDD, Reduce RDD, and Output RDD.
- The arrows indicate the dependencies between the stages, and each stage is made up of multiple tasks that can be executed in parallel.
- The Text RDD stage represents the initial loading of the data from a text file, and the subsequent stages involve applying transformations to the data to produce the final output.
- The Filter RDD stage applies a filter transformation to remove any unwanted data.
- The map RDD stage applies a map transformation to transform the remaining data.
- The reduce rdd stage applies a reduce transformation to aggregate the data and
- The output RDD stage writes the final output to a file.

# Fault tolerance with the help of DAG

- Spark achieves **fault tolerance** using the DAG by using a technique called **lineage**, which is the **record of the transformations that were used to create an RDD**.
- When a partition of an **RDD is lost** due to a node failure, Spark can **use the lineage to rebuild the lost partition**.
- The lineage is built up as the DAG is constructed, and Spark uses it to recover from any failures during the job execution.
- Spark uses the lineage to recompute the lost partitions.

To achieve fault tolerance, Spark uses **two mechanisms**:

1. RDD Persistence
2. Checkpointing

## Advantages of DAG in spark

### Efficient execution:

- The DAG allows Spark to break down a large-scale data processing job into smaller, independent tasks that can be executed in parallel.
- By executing the tasks in parallel, Spark can distribute the workload across multiple machines and perform the job much faster than if it was executed sequentially.

### Optimization:

- The DAG allows Spark to optimize the job execution by performing various optimizations, such as **pipelining, task reordering, and pruning unnecessary operations**.
- This helps to reduce the overall execution time of the job and improve performance.

### Fault tolerance:

- The DAG allows Spark to achieve fault tolerance by using the lineage to recover from node failures during the job execution.
- This ensures that the job can continue running even if a node fails, without losing any data.

### Reusability:

- The DAG allows Spark to reuse the intermediate results generated by a job.
- This means that if a portion of the data is processed once, it can be reused in subsequent jobs, thereby reducing the processing time and improving performance.

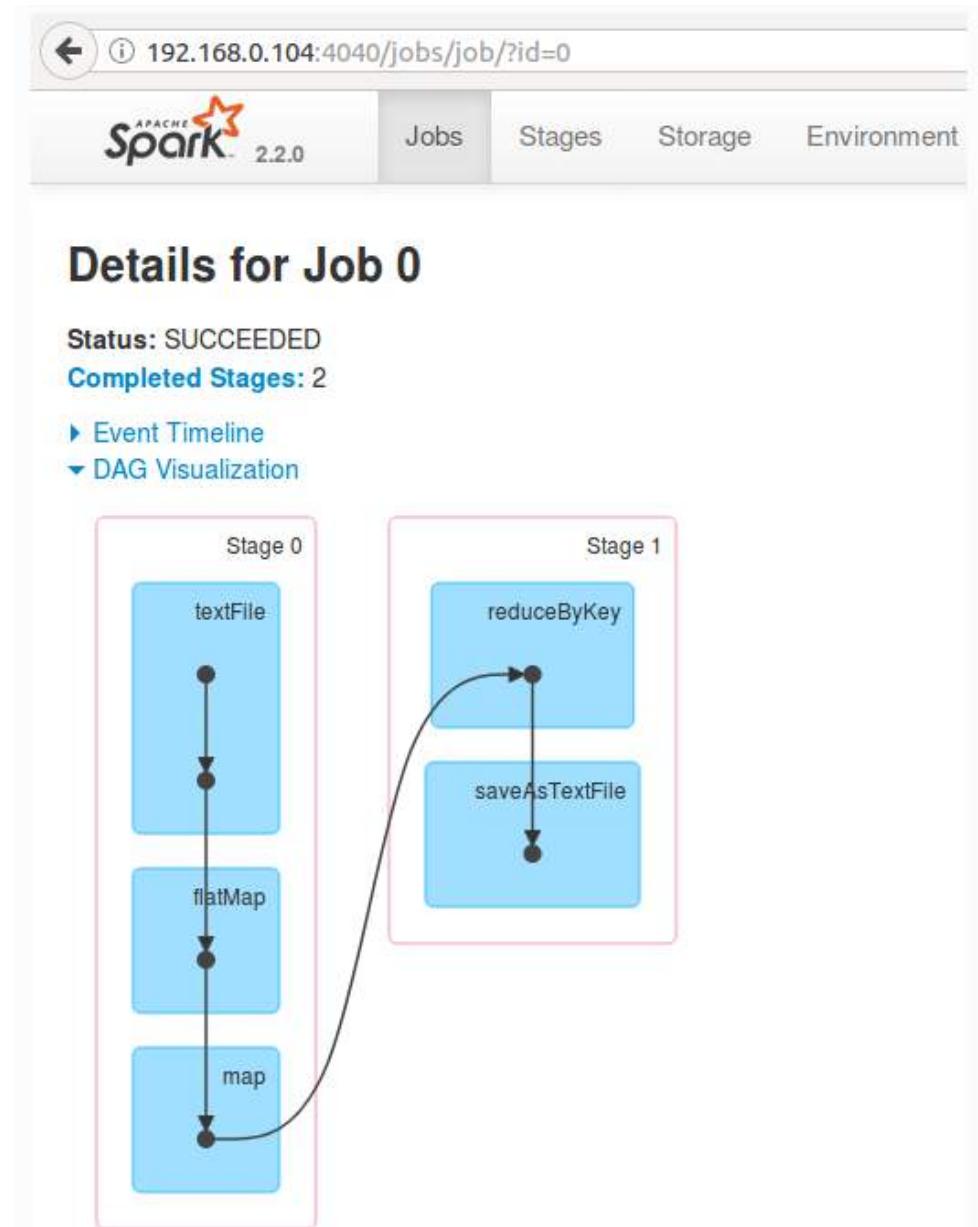
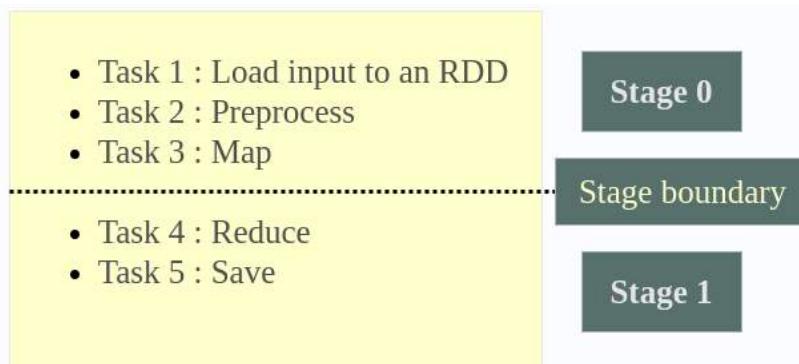
# Advantages of DAG in spark

## Visualization:

The DAG provides a visual representation of the **logical execution plan** of the job, which can help users to better understand the job and **identify any potential bottlenecks or performance issues**.

This could be visualized in **Spark Web UI**, once you run the WordCount example.

1. Hit the url `192.168.0.104:4040/jobs/`
2. Click on the link under Job Description.
3. Expand ‘DAG Visualization’



To work with the DAG Scheduler, you can use the following approaches:

**Visualize the DAG:**

- You can use the Spark UI to visualize the DAG of a Spark job.
- This allows you to see the different stages and tasks that make up the job and identify any potential bottlenecks or performance issues.

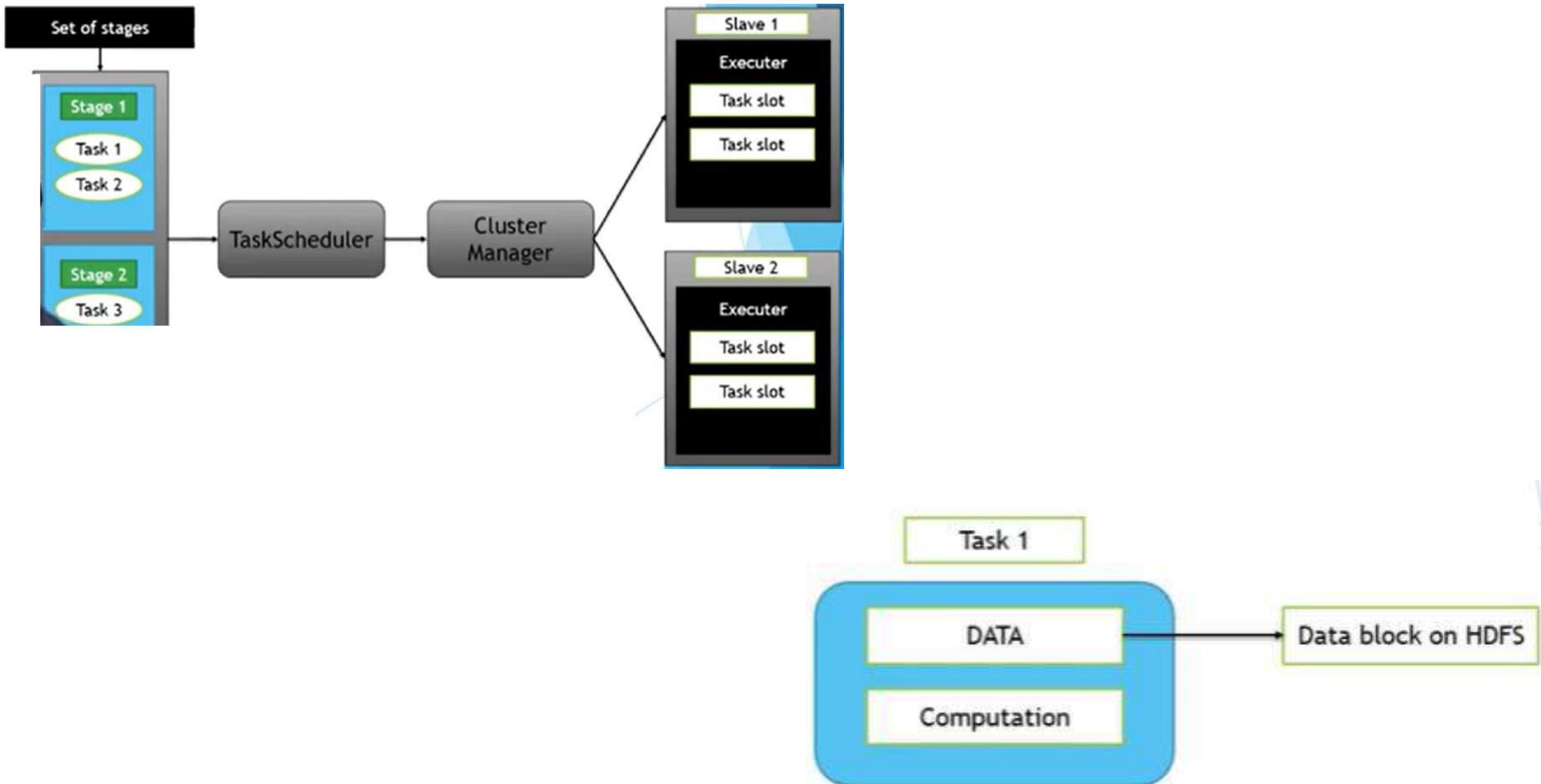
**Optimize the DAG:**

- You can optimize the DAG by using techniques such as pipelining, caching, and reordering of tasks to improve the performance of the job.

**Debug issues:**

- If you encounter issues with a Spark job, you can use the DAG Scheduler to **identify the root cause of the problem**.
- For example, you can use the Spark UI to identify any slow or failed stages and use this information to troubleshoot the issue.

# DAG Scheduler:



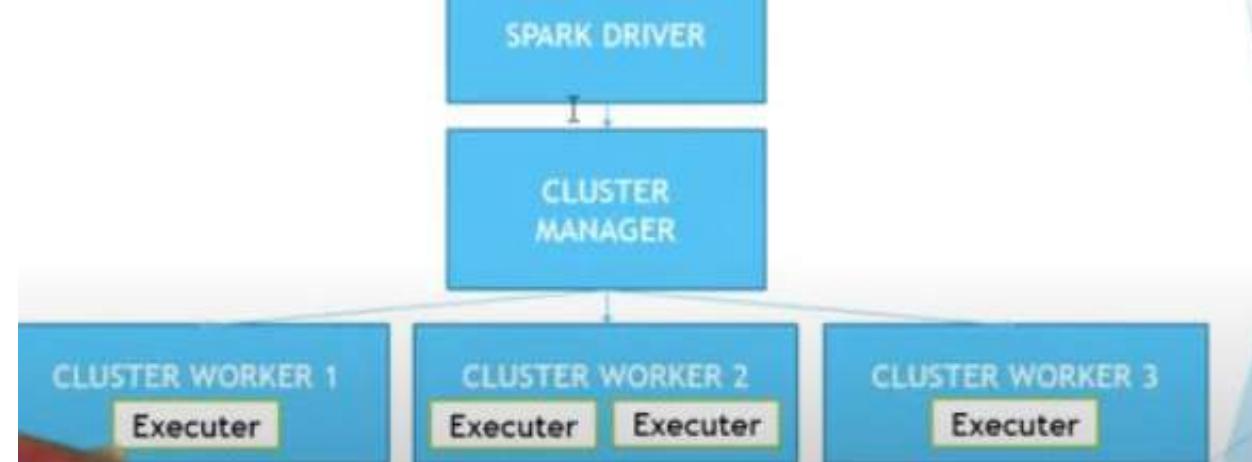
## Spark Context (Ticket To Park)

- ▶ Establish a connection to spark execution environment.
- ▶ It can be used to create RDDs, accumulators and broadcast variables.

# Spark Architecture

- **Spark Driver**

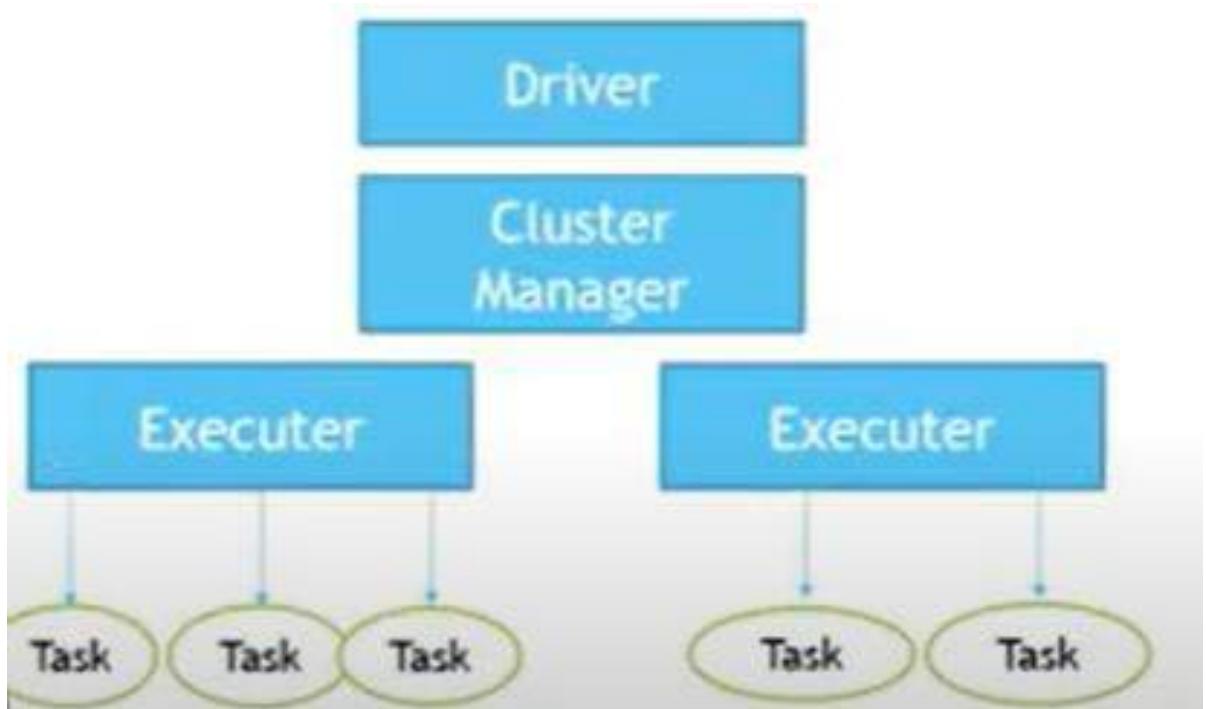
- It works on master slave concept.
- Master = Spark Driver
- Slave = Executer
- Spark Driver
- Runs on top of **master system**
- When we submit spark program , the **main method execution** is done by driver- (driver execution will be inside jvm)
- Creation of **Context** also takes place in driver
- Indirectly **Dag scheduler & task scheduler** are created at driver end itself.
- User **spark program (application)** will be converted into **actual spark job** by “Spark driver”
- **Cluster Manager- (CM)**
- It's a **pluggable component**
- Use – Yarn , Mesos, or Spark scheduler
- Jobs: **Resource Allocate or Deallocate**
- Driver – **send Tasks** – CM- To execute Task – **CM checks on which system data is present** – CM checks required resources are there or no (CM will check for availability of RAM on slave)- **If RAM available** – CM will assign the task to **Specific Executer Present** in worker Machine
- After the specific Task is executed –CM will **Deallocate Resources**



- **Executers:**

- Are **distributed**
- Run on **slave machines**
- Whenever particular **task need to be executed** it always executed inside executer.
- **Executer $\geq$ 1** can be present inside slave machine
- 1 Executer can have  $\geq$ 1 Task
- **Jobs:**
  - Data Read
  - Data Process
  - Results Writing  
(memory/disk/cache)

# Spark Program Execution:



- User - Spark application code is submitted to spark
- Spark Driver internal components - convert that program (user code) in to DAG (Map – execution graph)
- DAG will be give to DAG Scheduler
- DAG Scheduler – output (stages) – Task scheduler – Task list execution – Negotiate with CM – CM checks availability of resources & RAM - CM assigns Task list to Executer -Inside executer Task will be executed- Data read- data process and write results – Results will be sent from executer to driver .
- After completion of spark application- CM deallocates resources
- Before execution of Task in Executer- Executer has to Register in Driver (Driver will have information of running executers, quantity of executers present)

## Working Process :

1. Let's say a user submits a job using “spark-submit”.
2. “spark-submit” will in-turn launch the Driver which will execute the main() method of our code.
3. Driver contacts the cluster manager and requests for resources to launch the Executors.
4. The cluster manager launches the Executors on behalf of the Driver.
5. Once the Executors are launched, they establish a direct connection with the Driver.
6. The driver determines the total number of Tasks by checking the Lineage.
7. The driver creates the Logical and Physical Plan.
8. Once the Physical Plan is generated, Spark allocates the Tasks to the Executors.
9. Task runs on Executor and each Task upon completion returns the result to the Driver.
10. Finally, when all Task is completed, the main() method running in the Driver exits, i.e. main() method invokes sparkContext.stop().
11. Finally, Spark releases all the resources from the Cluster Manager.

# How Apache Spark builds a DAG and Physical Execution Plan ?

1. User **submits a spark application** to the Apache Spark.
2. Driver is the module that takes in the **application from Spark side**.
3. Driver **identifies transformations and actions present** in the spark application. These identifications are the tasks.
4. Based on the flow of program, these **tasks are arranged in a graph like structure** with directed flow of execution from task to task forming **no loops** in the graph (also called DAG).  
DAG is pure logical.
5. This **logical DAG is converted to Physical Execution Plan**. Physical Execution Plan contains stages.
6. **DAG Scheduler** creates a **Physical Execution Plan from the logical DAG**. Physical Execution Plan contains tasks and are bundled to be sent to nodes of cluster.

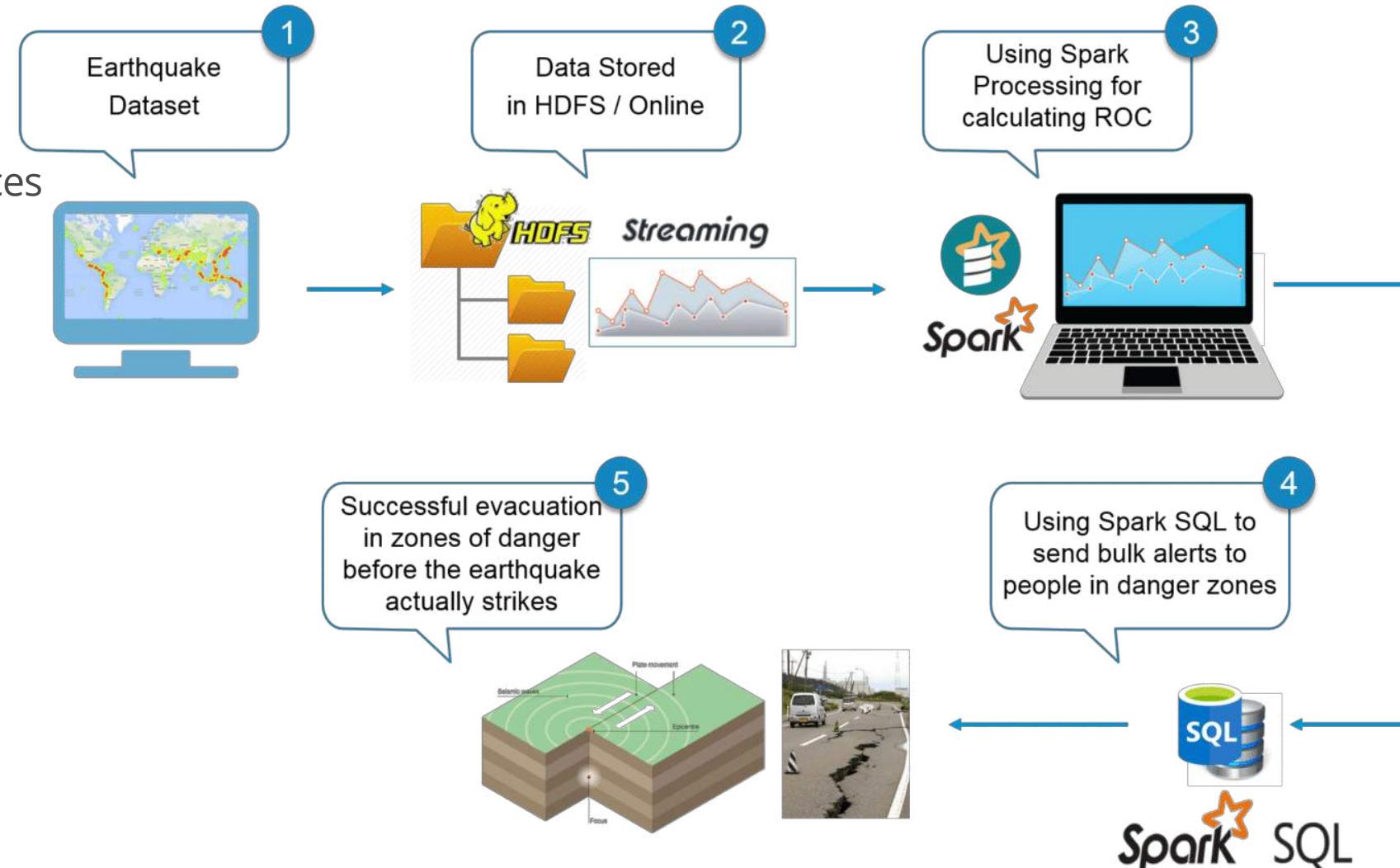
# Use Case: Earthquake Detection using Spark

94

**Problem Statement:** To design a Real Time Earthquake Detection Model to send life saving alerts, which should improve its machine learning to provide near real-time computation results.

## Use Case - Requirements:

1. Process data in real-time
2. Handle input from multiple sources
3. Easy to use system
4. Bulk transmission of alerts



# Common Transformations & Actions

```
scala> val rdd1 = sc.parallelize(List(1,2,3,4))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at p

scala> val maprdd1 = rdd1.map(x => x+5)
maprdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at map

scala> maprdd1.collect
res0: Array[Int] = Array(6, 7, 8, 9)

scala> val filterrdd1 = rdd1.filter(x => x!=3)
filterrdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at f

scala> filterrdd1.collect
res1: Array[Int] = Array(1, 2, 4)

scala> val rdd2 = sc.parallelize(List(1,2))
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:24

scala> val maprdd2 = rdd2.map(x=>x.to(3))
maprdd2: org.apache.spark.rdd.RDD[scala.collection.immutable.Range.Inclusive] = MapPartitionsRDD[4] at map at <console>:25

scala> maprdd2.collect
res2: Array[scala.collection.immutable.Range.Inclusive] = Array(Range(1, 2, 3), Range(2, 3))

scala> val flatmaprdd2 = rdd2.flatMap(x=>x.to(3))
flatmaprdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[5]

scala> flatmaprdd2.collect
res3: Array[Int] = Array(1, 2, 3, 2, 3)

scala> val rdd3 = sc.parallelize(List(1,1,2,3,2,4))
rdd3: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[6] at p

scala> val distinctrdd3 = rdd3.distinct
distinctrdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[9] at d

scala> distinctrdd3.collect
res4: Array[Int] = Array(4, 1, 2, 3)
```



```
scala> val rdd4 = sc.parallelize(List(1,2,3,4))
rdd4: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at parallelized
scala>
scala> val rdd5 = sc.parallelize(List(4,5))
rdd5: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at parallelized
scala> val unionrdd = rdd4.union(rdd5)
unionrdd: org.apache.spark.rdd.RDD[Int] = UnionRDD[12] at union at <console>
scala> unionrdd.collect
res5: Array[Int] = Array(1, 2, 3, 4, 4, 5)
```

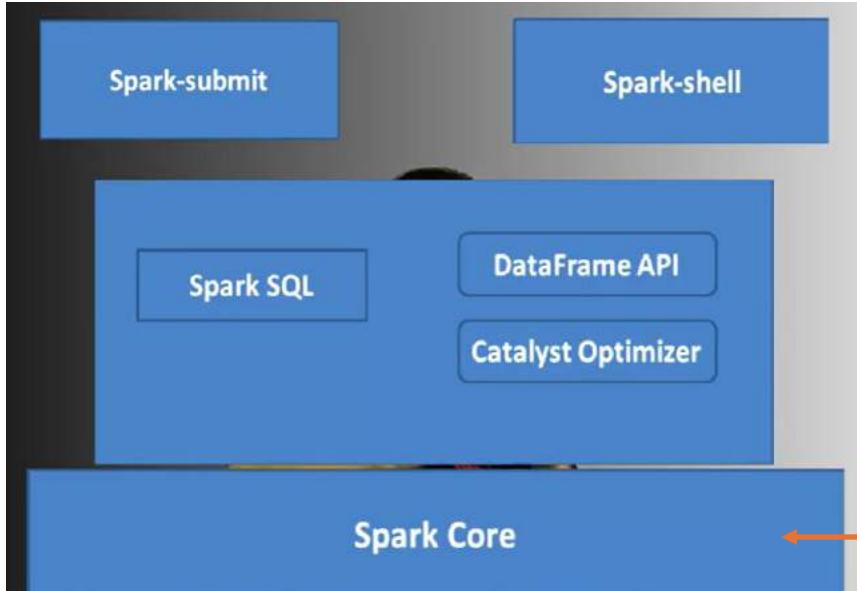
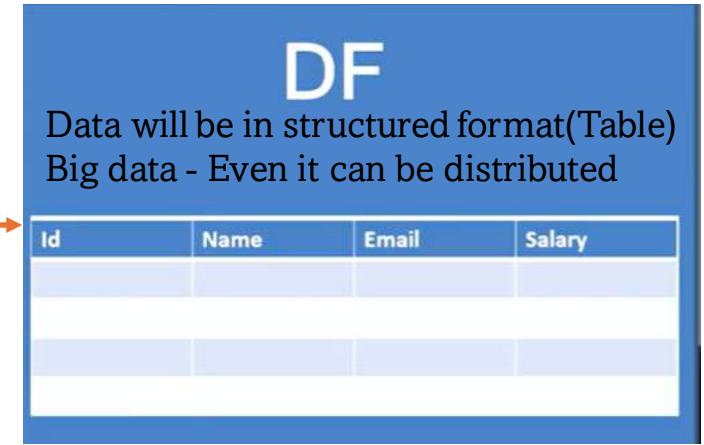
```
scala> subtractrdd.collect
res7: Array[Int] = Array(1, 2, 3)
scala> rdd4.reduce((x, y) => x + y)
res8: Int = 10
scala> rdd4.take(2)
res9: Array[Int] = Array(1, 2)
scala> rdd4.top(3)
res10: Array[Int] = Array(4, 3, 2)
```

```
scala> val intersectionrdd = rdd4.intersection(rdd5)
intersectionrdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[13] at intersection at <console>
scala> intersectionrdd.collect
res6: Array[Int] = Array(4)
scala> val subtractrdd = rdd4.subtract(rdd5)
```

# DATAFRAME (DF):

DF = Is RDD in which SQL Table is inserted

- RDD (Limitations)
  - Not Easy as SQL
  - No support of catalyst optimizer (CO – Helps to prioritize the operations = Increase performance, arranges the flow of work)

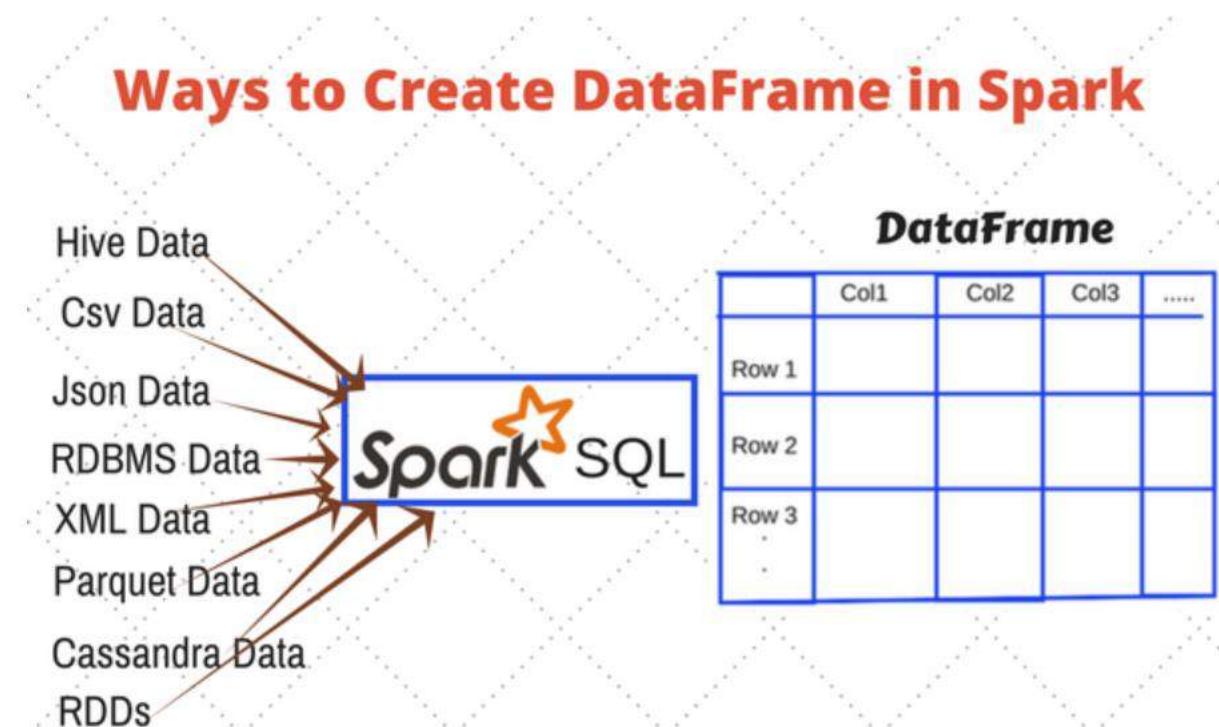


Execution of  
DF/RDD

# How to Create a DataFrame?

A DataFrame in Apache Spark can be created in multiple ways:

- It can be created using different data formats. For example, loading the data from JSON, CSV.
- Loading data from Existing RDD.
- Programmatically specifying schema



# Creating DataFrame from RDD

Steps for creating a DataFrame from list of tuples:

- Create a list of tuples. Each tuple contains name of a person with age.
- Create a RDD from the list above.
- Convert each tuple to a row.
- Create a DataFrame by applying **createDataFrame** on RDD with the help of **sqlContext**.

```
from pyspark.sql import Row
l = [('Ankit',25),('Jalfaizy',22),('saurabh',20),('Bala',26)]
rdd = sc.parallelize(l)
people = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
schemaPeople = sqlContext.createDataFrame(people)
```

# RDD To DF

```
hduser@ubuntu:~$ spark-shell --conf spark.sql.catalogImplementation=in-memory */
20/05/28 13:28:31 WARN Utils: Your hostname, ubuntu resolves to a loopback address:
```

```
scala> val pehlaRdd = sc.parallelize(1 to 10).map(x => (x,"df ka data"))
pehlaRdd: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[1] at map at <console>:24

scala> pehlaRdd.collect
20/05/28 13:31:11 WARN SizeEstimator: Failed to check whether UseCompressedOoops is set; assuming yes
res0: Array[(Int, String)] = Array((1,df ka data), (2,df ka data), (3,df ka data), (4,df ka data), (5,df
ka data), (6,df ka data), (7,df ka data), (8,df ka data), (9,df ka data), (10,df ka data))
```

```
scala> val pehlaDf = pehlaRdd.toDF("id","simple_string")
pehlaDf: org.apache.spark.sql.DataFrame = [id: int, simple_string: string]

scala> pehlaRdd.collect.foreach(println)
(1,df ka data)
(2,df ka data)
(3,df ka data)
(4,df ka data)
(5,df ka data)
(6,df ka data)
(7,df ka data)
(8,df ka data)
(9,df ka data)
(10,df ka data)
```

- We **didn't provide any schema**
- Spark **detected the data types by itself** – Because of RDD.
- Assume if you were suppose to read data file from Hadoop (HDFS) or from local system – define RDD – then we **need to define schema**.
- AS we cannot use **parallelize method** to create RDD always.

```
scala> pehlaDf.show
+---+-----+
| id|simple_string|
+---+-----+
1	df ka data
2	df ka data
3	df ka data
4	df ka data
5	df ka data
6	df ka data
7	df ka data
8	df ka data
9	df ka data
10	df ka data
+---+-----+
```

# Creating DF – Providing Schema

```
scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row
```

```
scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._
```

```
scala> val vidyarthiRdd = sc.parallelize(Array(Row(1,"sdp",25),(Row(2,"xyz",31))))
vidyarthiRdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = ParallelCollectionRD
e at <console>:28
```

```
scala> vidyarthiRdd
val vidyarthiRdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
```

```
scala> vidyarthiRdd.collect
20/05/28 15:39:27 WARN SizeEstimator: Failed to check whether UseCompressedOoops is set;
res0: Array[org.apache.spark.sql.Row] = Array([1,sdp,25], [2,xyz,31])
```

```
scala> val schema009 = StructType(
 | Array(
 | StructField("rollNo",IntegerType,true),
 | StructField("name",StringType,true),
 | StructField("marks",IntegerType,true)
 |)
schema009: org.apache.spark.sql.types.StructType = StructType(StructField(rollNo,IntegerType,true)
tField(name,StringType,true), StructField(marks,IntegerType,true))

scala> val vidyarthiDf = spark.createDataFrame(vidyarthiRdd,schema009);
vidyarthiDf: org.apache.spark.sql.DataFrame = [rollNo: int, name: string ... 1 more field]
```

```
scala> vidyarthiDf.printSchema
root
|-- rollNo: integer (nullable = true)
|-- name: string (nullable = true)
|-- marks: integer (nullable = true)
```

```
scala> vidyarthi
vidyarthiDf vidyarthiRdd

scala> vidyarthiDf.show
+---+---+---+
|rollNo|name|marks|
+---+---+---+
| 1| sdp| 25|
| 2| xyz| 31|
+---+---+---+
```

# Creating DF using CSV & Parquet files

\*spark\_df\_video

```
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2012
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,javed jafree,2007|
```

- Transfer the file to spark (read from HDFS / Local system)
- Create DF & Define Schema
  - Create Class
  - Create RDD
  - Create DF

```
scala> case class Movies(rank:Int,movie:String,actor:String,release_year:Int)
```

Class

```
scala> val FavouriteMoviesRdd = sc.textFile("/home/hduser/Desktop/data/fav_movies")
FavouriteMoviesRdd: org.apache.spark.rdd.RDD[String] = /home/hduser/Desktop/data/fav_movies
RDD[7] at textFile at <console>:28
```

RDD (Read file from local)

```
scala> FavouriteMoviesRdd.collect
collect collectAsync
```

```
scala> FavouriteMoviesRdd.collect.foreach(println)
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2012
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,iafree,2007
```

```
scala> FavouriteMoviesRdd.collect.foreach(println)
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2012
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,javed jafree,2007
```

```
scala> val FavouriteMoviesDf = FavouriteMoviesRdd.map(x => x.split(',')).map(x => Movies(x(0).toInt,x(1),
x(2),x(3).toInt)).toDF
```

$x(0) = 1$   
 $x(1) = \text{logan}$   
 $x(2) = \text{hugh jackman}$   
 $x(3) = 2017$

→ Create DF

```
case class Movies(rank:Int,movie:String,
actor:String,release_year:Int)
```

```
scala> val FavouriteMoviesDf = FavouriteMoviesRdd.map(x => x.split(',')).
map(x => Movies(x(0).toInt,x(1),
x(2),x(3).toInt)).toDF
FavouriteMoviesDf: org.apache.spark.sql.DataFrame = [rank: int, movie: string ... 2 more fields]

scala> FavouriteMovies
FavouriteMoviesDf FavouriteMoviesRdd

scala> FavouriteMoviesDf.printSchema
root
 |-- rank: integer (nullable = false)
 |-- movie: string (nullable = true)
 |-- actor: string (nullable = true)
 |-- release_year: integer (nullable = false)
```

```
er@ubunuc:~
scala> FavouriteMovies
FavouriteMoviesDf FavouriteMoviesRdd

scala> FavouriteMoviesDf.printSchema
root
|-- rank: integer (nullable = false)
|-- movie: string (nullable = true)
|-- actor: string (nullable = true)
|-- release_year: integer (nullable = false)
```

```
scala> FavouriteMovies
FavouriteMoviesDf FavouriteMoviesRdd
```

```
scala> FavouriteMoviesDf.show
```

| rank | movie                | actor              | release_year |
|------|----------------------|--------------------|--------------|
| 1    | logan                | hugh jackman       | 2017         |
| 2    | The Dark Knight      | Heath ledger       | 2008         |
| 3    | Wolverine            | hugh jackman       | 2013         |
| 4    | Deadpool             | ryan reynolds      | 2016         |
| 5    | Rocky                | Sylvester Stallone | 1976         |
| 6    | go goa gone          | saif               | 2013         |
| 7    | Joker                | joaquin phoenix    | 2019         |
| 8    | Avengers             | rdj                | 2012         |
| 9    | Sherlock Holmes      | rdj                | 2009         |
| 10   | pirates of the ca... | johnny depp        | 2003         |
| 11   | The Expendables      | sylvester stallone | 2010         |
| 12   | dhamaal              | javed jafree       | 2007         |

# Direct Load CSV file – No need to define Schema- No class creation

```
*spark_df_video
rank, movie, actor, release_year
1, logan, hugh jackman, 2017
2, The Dark Knight, Heath ledger, 2008
3, Wolverine, hugh jackman, 2013
4, Deadpool, ryan reynolds, 2016
5, Rocky, Sylvester Stallone, 1976
6, go goa gone, saif, 2013
7, Joker, joaquin phoenix, 2019
8, Avengers, rdj, 2008
9, sherlock Holmes, rdj, 2009
10, pirates of the caribbean, johnny depp, 2003
11, The Expendables, sylvester stallone, 2010
12, dhamaal, javed jafree, 2007
```

Define Column name in File itself  
Detect type (schema by spark)

```
scala> val moviesDf1 = spark.read.option("header","true").option("inferSchema","true").csv("/home/hduser/Desktop/data/fav_movies_with_header")
moviesDf1: org.apache.spark.sql.DataFrame = [rank: int, movie: string ... 2 more fields]

scala> moviesDf1.printSchema
root
|-- rank: integer (nullable = true)
|-- movie: string (nullable = true)
|-- actor: string (nullable = true)
|-- release_year: integer (nullable = true)

scala> moviesDf1.show
+---+-----+-----+-----+
|rank| movie| actor|release_year|
+---+-----+-----+-----+
1	logan	hugh jackman	2017
2	The Dark Knight	Heath ledger	2008
3	Wolverine	hugh jackman	2013
4	Deadpool	ryan reynolds	2016
5	Rocky	Sylvester Stallone	1976
6	go goa gone	saif	2013
7	Joker	joaquin phoenix	2019
8	Avengers	rdj	2008
9	sherlock Holmes	rdj	2009
10	pirates of the ca...	johnny depp	2003
11	The Expendables	sylvester stallone	2010
12	dhamaal	javed jafree	2007
+---+-----+-----+-----+
```

# Parquet File- No column Names, Schema

- Parquet File – Inside it will have information of “Column names, data types, Schema”- No need to define them.
- But file need to be with parquet extension (read from local/hdfs system)

```
scala> val moviesDf2 = spark.read.load("/home/hduser/Desktop/data/movies/movies.parquet")
moviesDf2: org.apache.spark.sql.DataFrame = [rank: int, movie: string ... 2 more fields]
```

```
scala> moviesDf2.show
+---+-----+-----+-----+
|rank| movie| actor|release_year|
+---+-----+-----+-----+
1	logan	hugh jackman	2017
2	The Dark Knight	Heath ledger	2008
3	Wolverine	hugh jackman	2013
4	Deadpool	ryan reynolds	2016
5	Rocky	Sylvester Stallone	1976
6	go goa gone	saif	2013
7	Joker	joaquin phoenix	2019
8	Avengers	rdj	2008
9	sherlock Holmes	rdj	2009
10	pirates of the ca...	johnny depp	2003
11	The Expendables	sylvester stallone	2010
12	dhamaal	javed jafree	2007
+---+-----+-----+-----+
```

# Word Count Program using Spark

- Create a text file in your local machine and write some text into it.

```
$ nano sparkdata.txt
```

Create a directory in HDFS, where to kept text file.

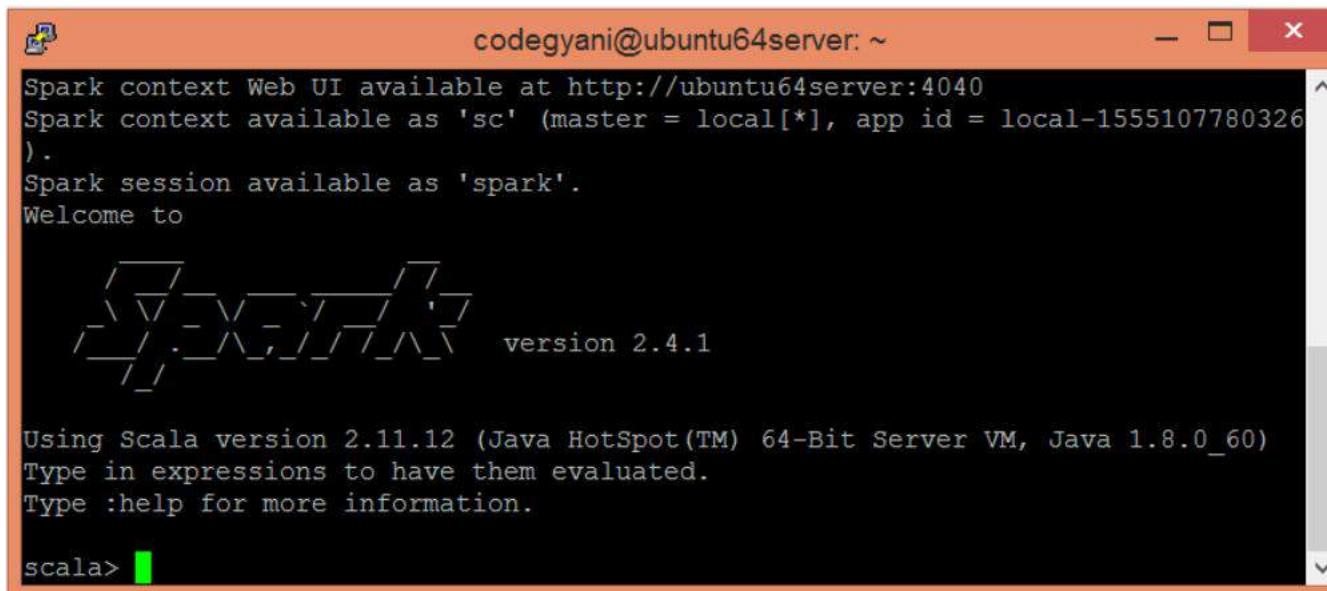
```
$ hdfs dfs -mkdir /spark
```

Upload the sparkdata.txt file on HDFS in the specific directory.

```
$ hdfs dfs -put /home/codegyani/sparkdata.txt /spark
```

Now, follow the below command to open the spark in Scala mode.

```
$ spark-shell
```



```
codegyani@ubuntu64server: ~
Spark context Web UI available at http://ubuntu64server:4040
Spark context available as 'sc' (master = local[*], app id = local-1555107780326
).
Spark session available as 'spark'.
Welcome to
 __\ \
 / \ _ _ _ _ _ _ _ \
version 2.4.1

Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

- Let's create an RDD by using the following command.

```
scala> val data=sc.textFile("sparkdata.txt")
```

Now, we can read the generated result by using the following command.

```
scala> data.collect;
```

- Here, we split the existing data in the form of individual words by using the following command.

```
scala> val splitdata = data.flatMap(line => line.split(" "));
```

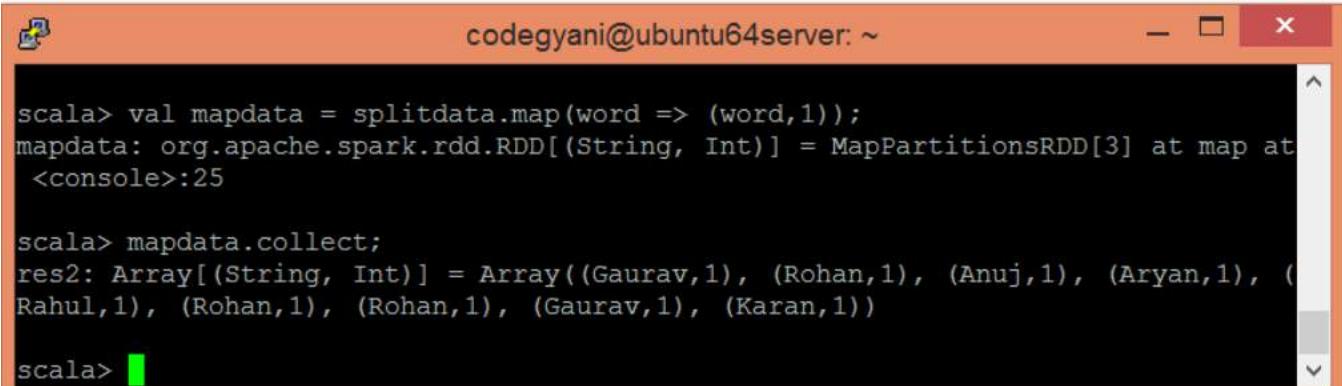
- Now, we can read the generated result by using the following command.

```
scala> splitdata.collect;
```

- Now, perform the map operation.

```
scala> val mapdata = splitdata.map(word => (word,1));
```

```
scala> mapdata.collect;
```



A screenshot of a terminal window titled 'codegyani@ubuntu64server: ~'. The window shows Scala code being run. The first two lines of code define an RDD 'splitdata' from a file and then collect its elements. The third line defines a new RDD 'mapdata' by mapping each word to a tuple (word, 1). The fourth line collects the elements of 'mapdata' into an array 'res2', which contains tuples where each word appears once. A blue arrow points from the 'mapdata.collect;' line in the text above to the corresponding line in the terminal screenshot.

```
scala> val mapdata = splitdata.map(word => (word,1));
mapdata: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:25

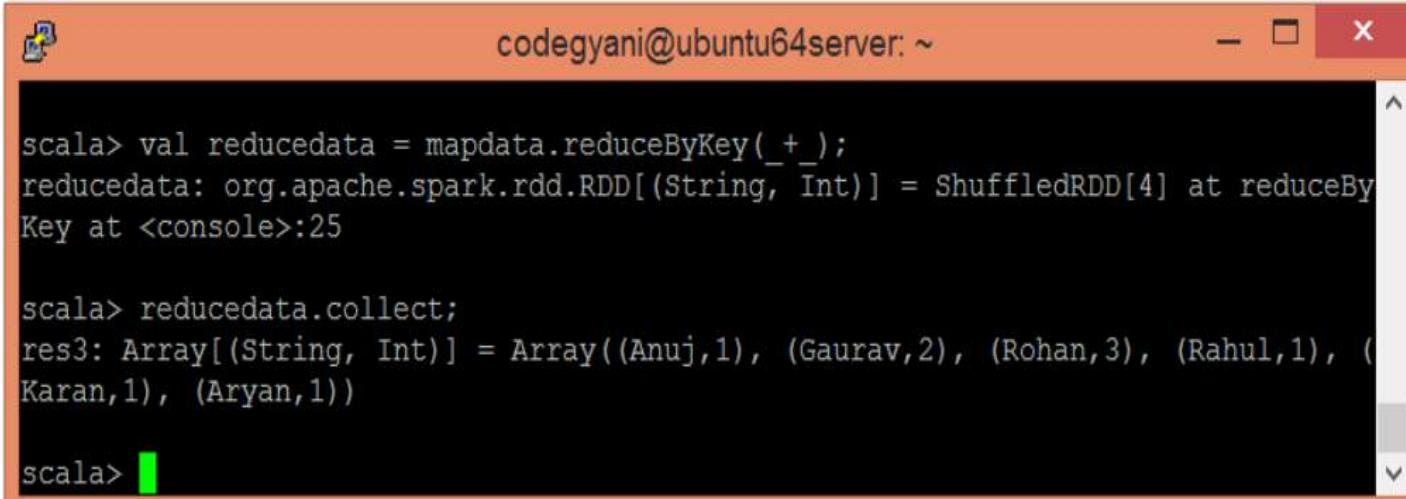
scala> mapdata.collect;
res2: Array[(String, Int)] = Array((Gaurav,1), (Rohan,1), (Anuj,1), (Aryan,1), (Rahul,1), (Rohan,1), (Rohan,1), (Gaurav,1), (Karan,1))

scala>
```

- Now, perform the reduce operation

```
scala> val reducedata = mapdata.reduceByKey(_+_);
```

```
scala> reducedata.collect;
```



A screenshot of a terminal window titled "codegyani@ubuntu64server: ~". The window shows Scala code being run. The code defines a variable `reducedata` as the result of a `reduceByKey` operation on `mapdata`, which is a ShuffledRDD[4]. It then collects the data, resulting in an array of tuples where each tuple contains a string and an integer. The output is as follows:

```
scala> val reducedata = mapdata.reduceByKey(_+_);
reducedata: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:25

scala> reducedata.collect;
res3: Array[(String, Int)] = Array((Anuj,1), (Gaurav,2), (Rohan,3), (Rahul,1), (Karan,1), (Aryan,1))

scala>
```

Now, on to the WordCount script. For local testing, we will use a file from our file system.

```
val text = sc.textFile("mytextfield.txt")
val counts = text.flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey(_+_).collect
```

The next step is to run the script.

```
spark-shell -i WordCountscala.scala
```

A dark green background featuring a repeating pattern of tropical leaves, including palm fronds and smaller greenery.

*Thank you*



# PART 6

# Apache Spark

BY: Dr.Rashmi L Malghan



# AGENDA

- Apache Spark
- Pre- Requisites of Spark
- Uses of Spark
- Features / Why Spark
- Cluster Managers
- Storage layers of Spark
- Quick Introduction
- RDD
- RDD Operations
- Types of RDD
- Spark Architecture
- Spark Program execution
- Case Study
- Sample Transformations- Scala
- Actions Execution - Scala
- Defining Schema/ DF
- Non \_-Defining Schema/DF
- Parquet files

# What is Spark

- Open source software tool
- Distributed cluster computing
- Batch/Stream processing
- Integrated with various big data tools
- Interactive querying
- Cluster management system

# What is APACHE SPARK?



Apache Spark is an open-source data processing engine to store and process data in real-time across various clusters of computers using simple programming constructs

Support various programming languages



Developers and data scientists incorporate Spark into their applications to rapidly query, analyze, and transform data at scale



Query



Analyze



Transform

# Limitations of MapReduce

- Use only Java for application building.
- Opt **only for batch processing**. Does not support stream processing.
- Hadoop MapReduce **uses disk-based processing**.

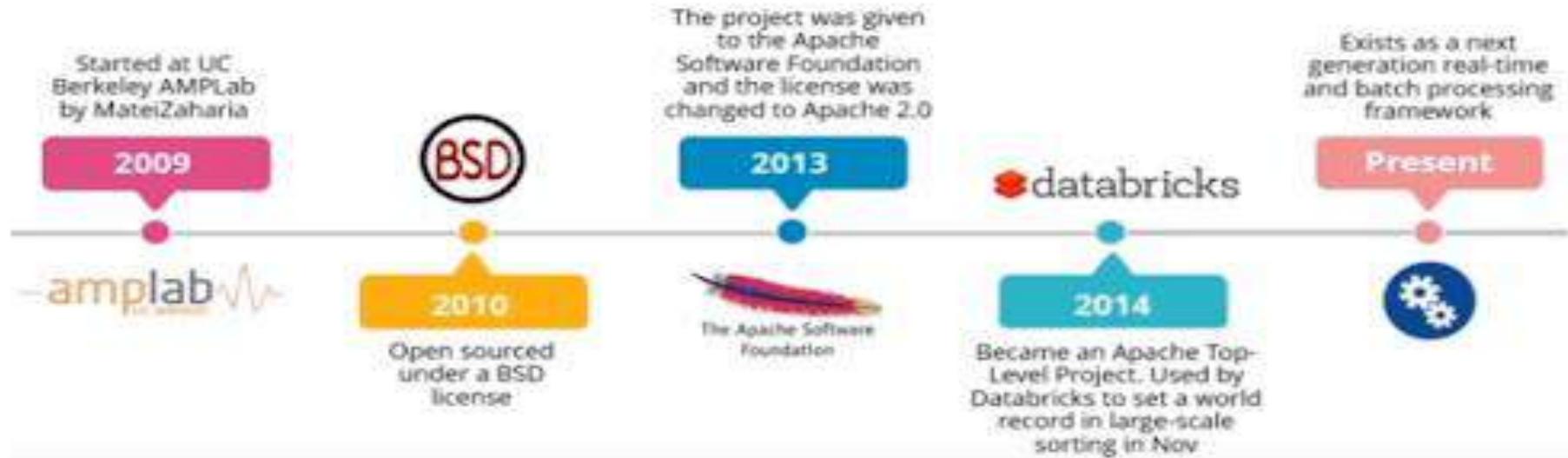
# Why Spark?

In the industry, there is a need for a **general-purpose cluster** computing tool as:

- Hadoop **MapReduce** can only perform batch processing.
- Apache Storm / S4 can only perform stream processing.
- Apache Impala / Apache Tez can only perform interactive processing
- Neo4j / Apache Giraph can only perform graph processing
- There was a big demand for a powerful engine that can process the **data in real-time (streaming) as well as in batch mode.**
- There was a need for an engine that can respond in sub-second and perform **in-memory processing**.

# History of Spark?

The history of Spark is explained below:

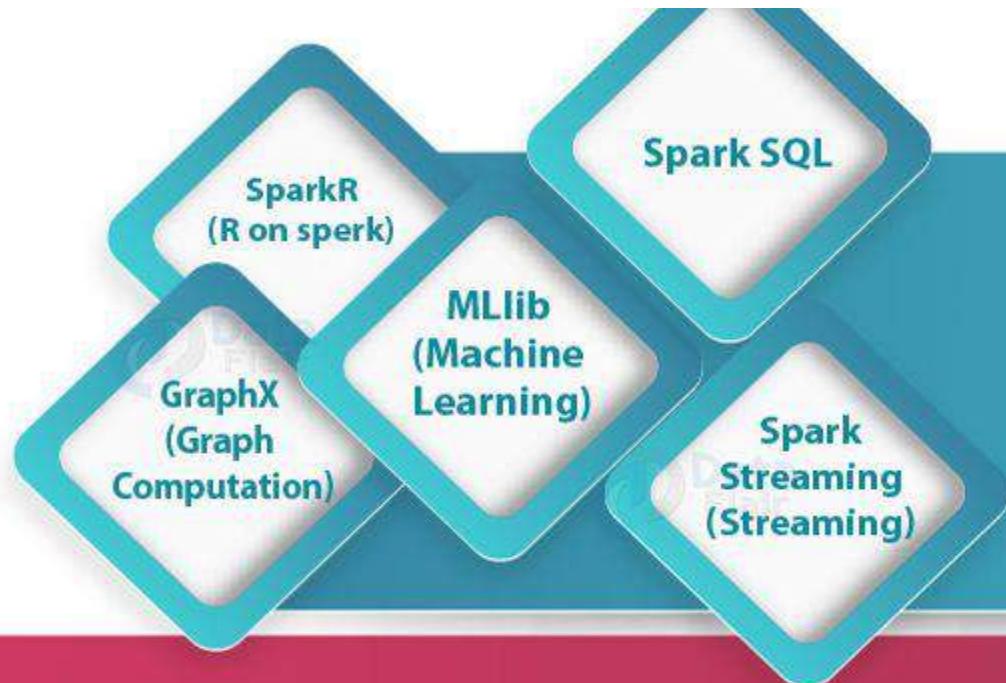


- The Spark was initiated by Matei Zaharia at UC Berkeley's AMPLab in 2009.
- It was open sourced in 2010 under a BSD license.
- In 2013, the project was acquired by Apache Software Foundation.
- In 2014, the Spark emerged as a Top-Level Apache Project.

## Apache Spark:

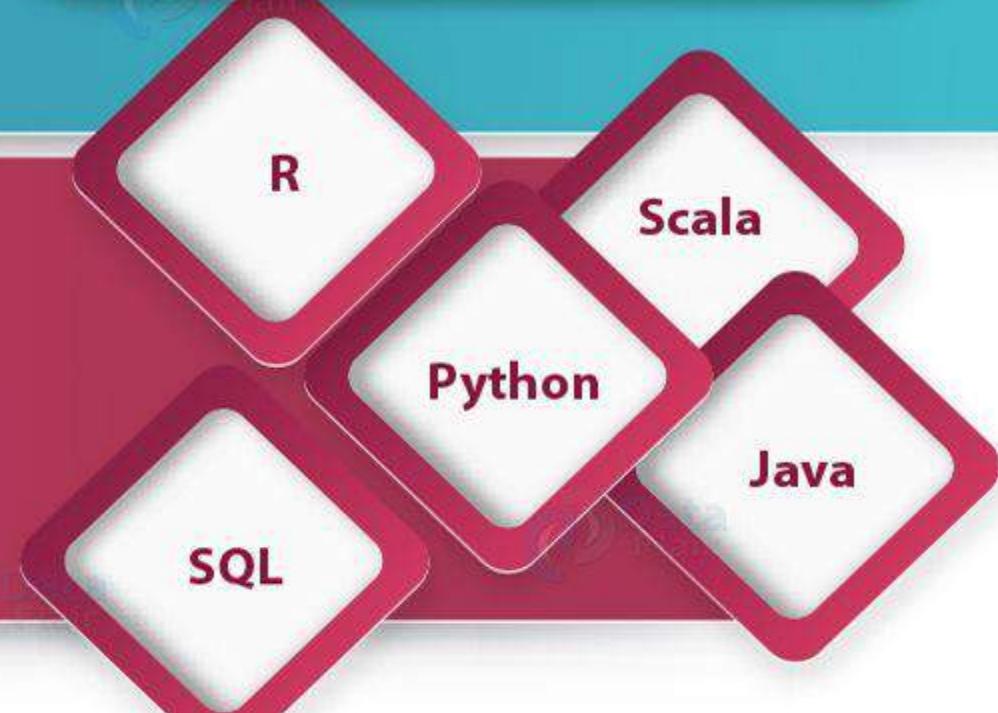
- ▶ Cluster computing platform designed to be fast and general purpose.
- ▶ In memory computation.
- ▶ Designed to cover wide range of workloads(batch applications, machine learning, interactive queries, streaming).
- ▶ Combines different processing types.
- ▶ Integrates closely with other Big Data tools.

# Apache Spark Ecosystem



**Apache Spark Core API**

**Apache Spark Ecosystem**



# Hadoop V/S Spark

| Basis                                     | Hadoop                                                                                         | Spark                                                                                                                        |
|-------------------------------------------|------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <b>Processing Speed &amp; Performance</b> | Hadoop's MapReduce model reads and writes from a disk, thus slowing down the processing speed. | Spark reduces the number of read/write cycles to disk and stores intermediate data in memory, hence faster-processing speed. |
| <b>Usage</b>                              | Hadoop is designed to handle batch processing efficiently.                                     | Spark is designed to handle real-time data efficiently.                                                                      |
| <b>Latency</b>                            | Hadoop is a high latency computing framework, which does not have an interactive mode.         | Spark is a low latency computing and can process data interactively.                                                         |
| <b>Data</b>                               | With Hadoop MapReduce, a developer can only process data in batch mode only.                   | Spark can process real-time data, from real-time events like Twitter, and Facebook.                                          |
| <b>Cost</b>                               | Hadoop is a cheaper option available while comparing it in terms of cost                       | Spark requires a lot of RAM to run in-memory, thus increasing the cluster and hence cost.                                    |
| <b>Algorithm Used</b>                     | The PageRank algorithm is used in Hadoop.                                                      | Graph computation library called GraphX is used by Spark.                                                                    |

# Hadoop V/S Spark:

|                            |                                                                                                                                                                      |                                                                                                                                                      |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Fault Tolerance</b>     | Hadoop is a highly fault-tolerant system where Fault-tolerance achieved by replicating blocks of data.<br>If a node goes down, the data can be found on another node | Fault-tolerance achieved by storing chain of transformations<br>If data is lost, the chain of transformations can be recomputed on the original data |
| <b>Security</b>            | Hadoop supports LDAP, ACLs, SLAs, etc and hence it is extremely secure.                                                                                              | Spark is not secure, it relies on the integration with Hadoop to achieve the necessary security level.                                               |
| <b>Machine Learning</b>    | Data fragments in Hadoop can be too large and can create bottlenecks. Thus, it is slower than Spark.                                                                 | Spark is much faster as it uses MLlib for computations and has in-memory processing.                                                                 |
| <b>Scalability</b>         | Hadoop is easily scalable by adding nodes and disk for storage.<br>It supports tens of thousands of nodes.                                                           | It is quite difficult to scale as it relies on RAM for computations. It supports thousands of nodes in a cluster.                                    |
| <b>Language support</b>    | It uses Java or Python for MapReduce apps.                                                                                                                           | It uses Java, R, Scala, Python, or Spark SQL for the APIs.                                                                                           |
| <b>User-friendliness</b>   | It is more difficult to use.                                                                                                                                         | It is more user-friendly.                                                                                                                            |
| <b>Resource Management</b> | YARN is the most common option for resource management.                                                                                                              | It has built-in tools for resource management.                                                                                                       |

# Hadoop V/S Apache Spark

| Features                  | Hadoop                                                                                          | Apache Spark                                                                                                                |
|---------------------------|-------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Data Processing           | Apache Hadoop provides batch processing                                                         | Apache Spark provides both batch processing and stream processing                                                           |
| Memory usage              | Hadoop is disk-bound                                                                            | Spark uses large amounts of RAM                                                                                             |
| Security                  | Better security features                                                                        | Its security is currently in its infancy                                                                                    |
| Fault Tolerance           | Replication is used for fault tolerance.                                                        | RDD and various data storage models are used for fault tolerance.                                                           |
| Graph Processing          | Algorithms like PageRank is used.                                                               | Spark comes with a graph computation library called GraphX.                                                                 |
| Ease of Use               | Difficult to use.                                                                               | Easier to use.                                                                                                              |
| Real-time data processing | It fails when it comes to real-time data processing.                                            | It can process real-time data.                                                                                              |
| Speed                     | Hadoop's MapReduce model reads and writes from a disk, thus it slows down the processing speed. | Spark reduces the number of read/write cycles to disk and store intermediate data in memory, hence faster-processing speed. |
| Latency                   | It is high latency computing framework.                                                         | It is a low latency computing and can process data interactively                                                            |

# Pre-Requisites For Spark

- ▶ Basic knowledge of Python, Scala, SQL, Linux, Hadoop, Statistics.

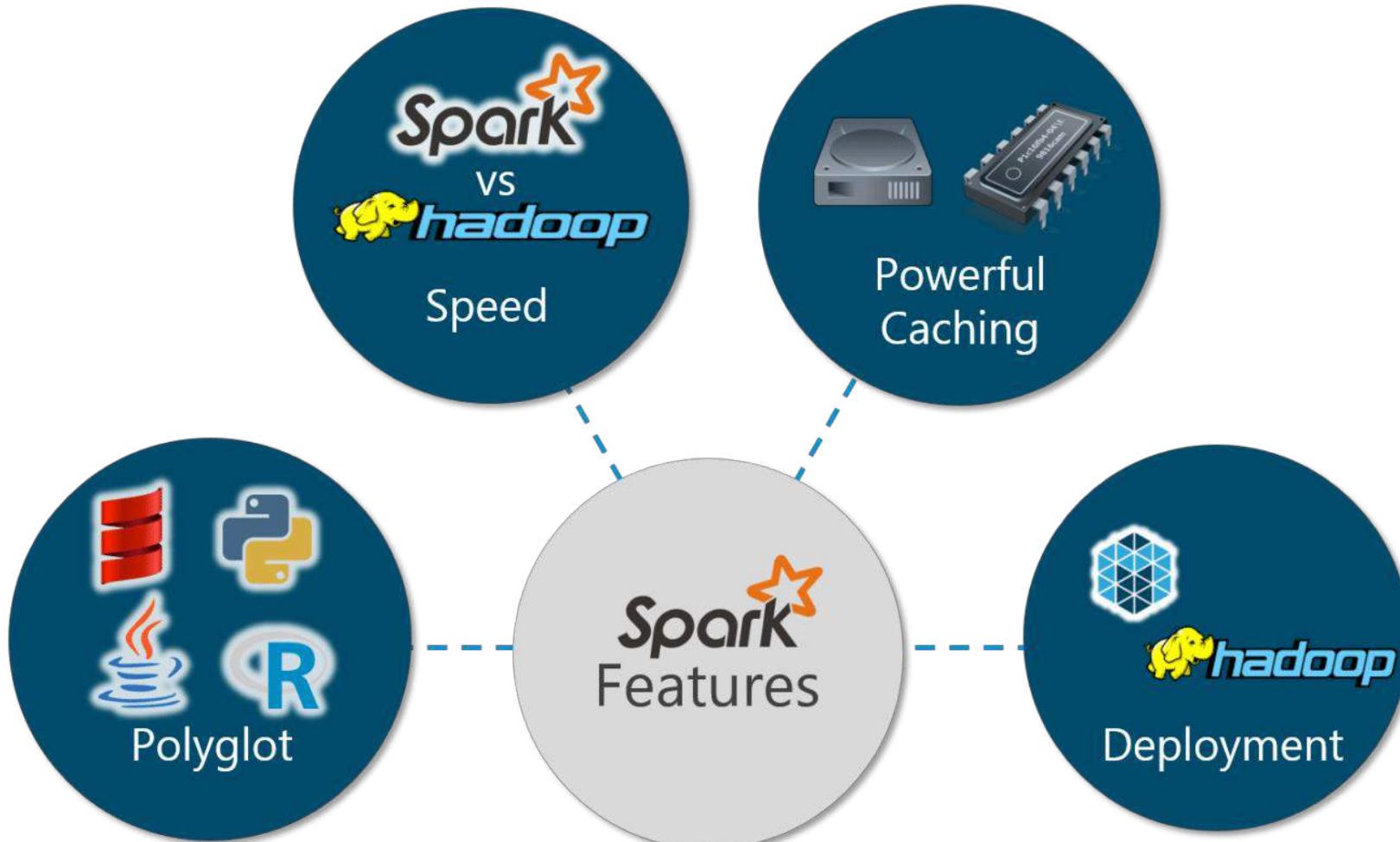
# Uses of Spark:

- ▶ Data processing applications
  - Batch Applications
  - SQL
  - Machine Learning
  - Streaming data processing
  - Graph data processing

# Feature of Spark / Why Spark?

- ▶ In memory processing.
- ▶ Tight integration of components.
- ▶ Easy and inexpensive.

# Features of Spark



## Features of Apache Spark

**Swift Processing:** swift processing speed of up to 100x faster in memory and 10x faster than Hadoop even when running on disk.

**Dynamic:** 80 high-level operators, Scala being defaulted language for Spark. We can also work with Java, Python, R.

**In – Memory Processing:** Disk seeks is becoming very costly, Spark keeps data in memory for faster access. Spark owns advanced DAG execution engine.

**Reusability:** Apache Spark provides the provision of code reusability for batch processing, join streams against historical data, or run adhoc queries on stream state.

**Fault Tolerance:** Spark RDD (Resilient Distributed Dataset), abstraction are designed to seamlessly handle failures of any worker nodes in the cluster.  
Real-Time Stream Processing

# Features of Spark

## Polyglot:

Spark provides high-level APIs in Java, Scala, Python and R. Spark code can be written in any of these four languages. It provides a shell in Scala and Python. The Scala shell can be accessed through **./bin/spark-shell** and Python shell through **./bin/pyspark** from the installed directory.



## Speed:



Spark runs up to 100 times faster than Hadoop MapReduce for large-scale data processing. Spark is able to achieve this speed through controlled partitioning. It manages data using partitions that help parallelize distributed data processing with minimal network traffic.

# Features of Spark

## Multiple Formats:

Spark supports multiple data sources such as Parquet, JSON, Hive and Cassandra apart from the usual formats such as text files, CSV and RDBMS tables. The Data Source API provides a pluggable mechanism for accessing structured data through Spark SQL. Data sources can be more than just simple pipes that convert data and pull it into Spark.



## Real Time Computation:

Spark's computation is real-time and has low latency because of its in-memory computation. Spark is designed for massive scalability and the Spark team has documented users of the system running production clusters with thousands of nodes and supports several computational models.



# Features of Spark



- 1  + 
- 2  + 

## Hadoop Integration:

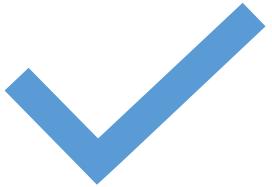
Apache Spark provides smooth compatibility with Hadoop. This is a boon for all the Big Data engineers who started their careers with Hadoop. Spark is a potential replacement for the MapReduce functions of Hadoop, while Spark has the ability to run on top of an existing Hadoop cluster using YARN for resource scheduling.

## Machine Learning:

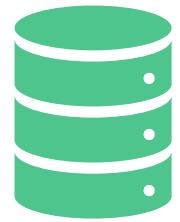
Spark's MLlib is the machine learning component which is handy when it comes to big data processing. It eradicates the need to use multiple tools, one for processing and one for machine learning. Spark provides data engineers and data scientists with a powerful, unified engine that is both fast and easy to use.



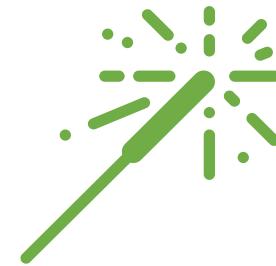
# Cluster Managers:



Hadoop YARN



Apache Mesos

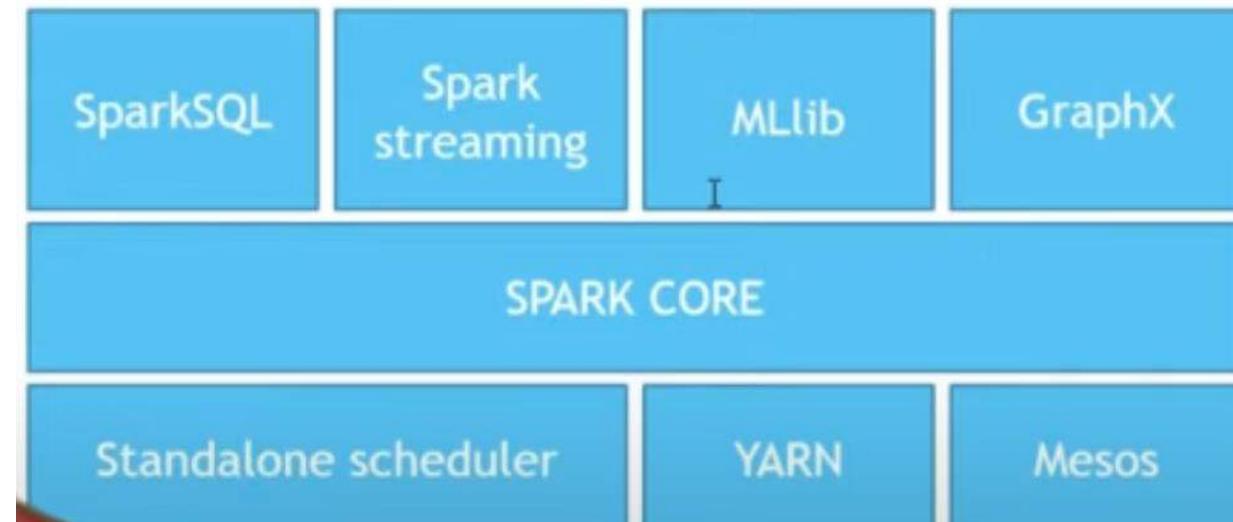


Standalone Scheduler : Install  
Spark on empty set of  
machines.

# Storage layer of Spark

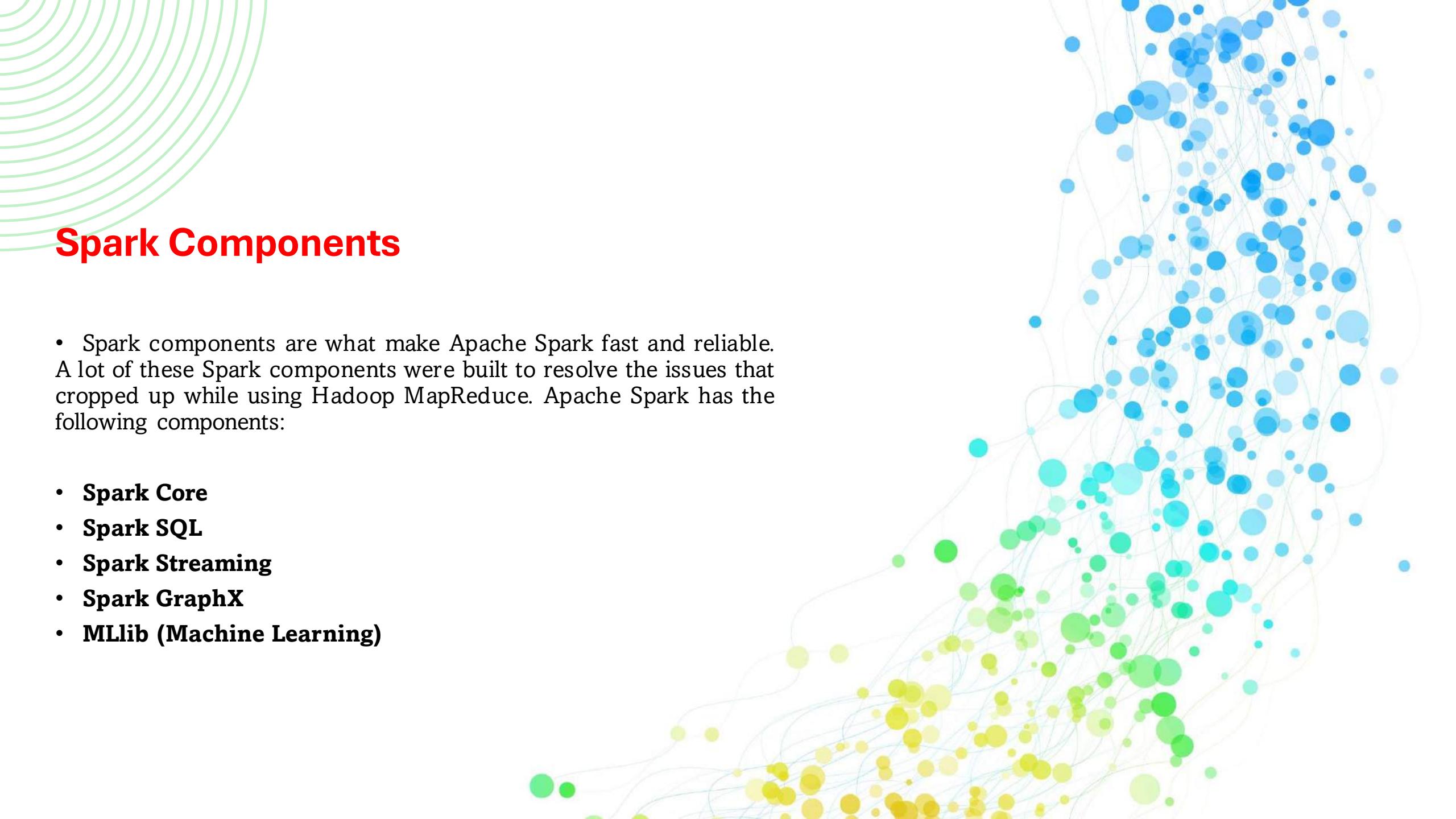
- ▶ HDFS and other storage systems supported by Hadoop APIs. ( Local filesystem, Amazon S3, Cassandra, etc )
- ▶ Supports text files, Avro, Parquet, and many other Hadoop InputFormat.

# Unified Stack:



- **Base layer** = Cluster Managers
- **Middle layer** = Spark Core (Engine of spark)
  - **Spark Core** = It will manage:
  - Task Scheduling, Distributing, Monitoring applications
- **Top Layer** = Own components of spark to perform various operations

| SL.NO | Hadoop                                                                                                    | Apache Spark                                                                                                                                           |
|-------|-----------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1     | It does not have its own components to handle : structured data (Hive), machine learning (Install Mahout) | It has its own components to perform various processing types.                                                                                         |
| 2     | Lot of time required to install, deploy & maintain                                                        | Installation Time, Deployment time, Maintenance time is reduced.                                                                                       |
| 3     | Not much suitable                                                                                         | If new application to be developed – it should perform various workloads like query streaming, ML, graph data processing etc.<br>So spark is suitable. |



## Spark Components

- Spark components are what make Apache Spark fast and reliable. A lot of these Spark components were built to resolve the issues that cropped up while using Hadoop MapReduce. Apache Spark has the following components:
  - **Spark Core**
  - **Spark SQL**
  - **Spark Streaming**
  - **Spark GraphX**
  - **Mlib (Machine Learning)**

# Spark Core:

- ▶ Contains basic functionality of Spark. (task scheduling, memory management, fault recovery, interacting with storage systems)
- ▶ Home to API that defines RDDs

# Spark Core:

Spark Core is the base engine for large-scale parallel and distributed data processing



## 1. Spark Core

- *Spark Core* is the base engine for large<sup>27</sup>-scale **parallel** and **distributed** data processing.
- The core is the distributed execution engine and the **Java, Scala, and Python APIs** offer a platform for distributed ETL application development.
- Further, additional libraries which are built on the core allow **diverse workloads** for streaming, SQL, and machine learning.
- It is responsible for:
  - ✓ Memory management and fault recovery
  - ✓ Scheduling, distributing and
  - ✓ monitoring jobs on a cluster
  - ✓ Interacting with storage systems

# Spark Core:

28

Spark Core is embedded with RDDs (Resilient Distributed Datasets), an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel

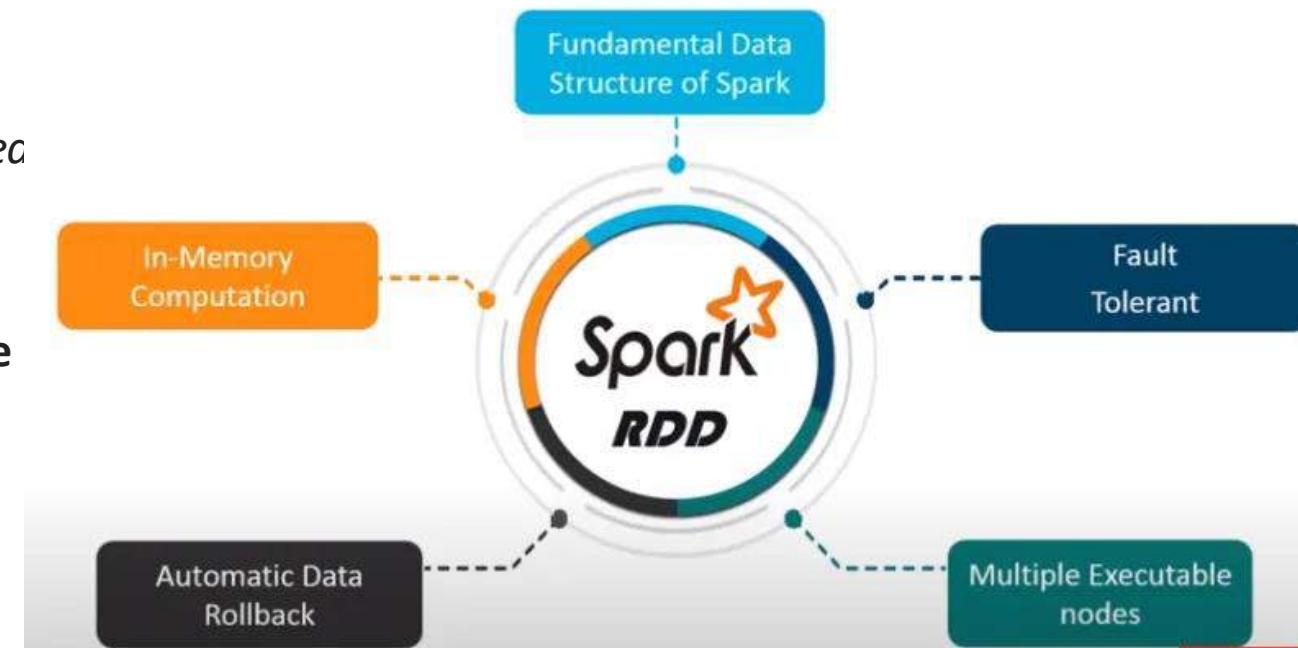


These are operations (such as map, filter, join, union) that are performed on an RDD that yields a new RDD containing the result

These are operations (such as reduce, first, count) that return a value after running a computation on an RDD

# What are RDDs?

- RDD or (**Resilient Distributed Data set**) is a fundamental **data structure** in Spark.
- The term **Resilient** defines the ability that generates the data automatically or data **rolling back** to the **original state** when an unexpected calamity occurs with a probability of data loss.
- RDD does **not need** something like **Hard Disks** or any other secondary storage. **It needs only RAM**.
- *RDD is not a Distributed File System instead it is Distributed Memory System.*
- Data can be loaded from any source to Apache Spark like Hadoop, HBase, Hive, SQL, S3
- They can process any type of data such as Structured, Unstructured, Semi-Structured data.



# Operations performed on RDDs

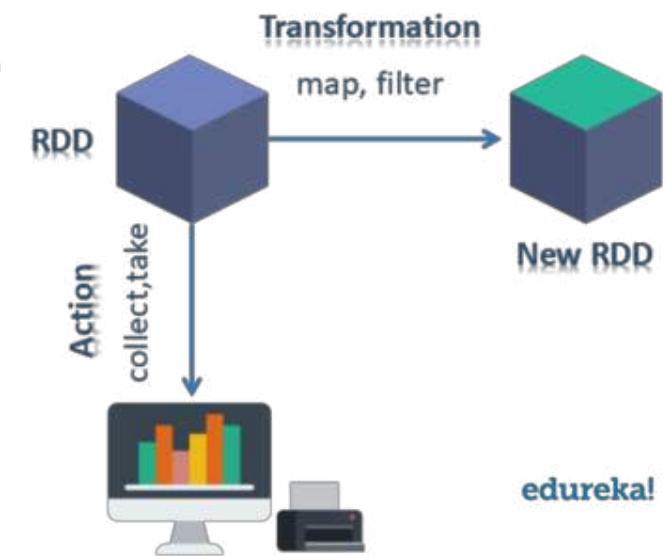
## Transformations:

- **filter, access** and **modify** the data in parent RDD to generate a **successive RDD**
- The new RDD returns a pointer to the previous RDD ensuring the dependency between them.
- Transformations are **Lazy Evaluations**
- **1. Narrow Transformations**

- `map()` , `filter()`
- `flatMap()`, `partition()`
- `mapPartitions()`

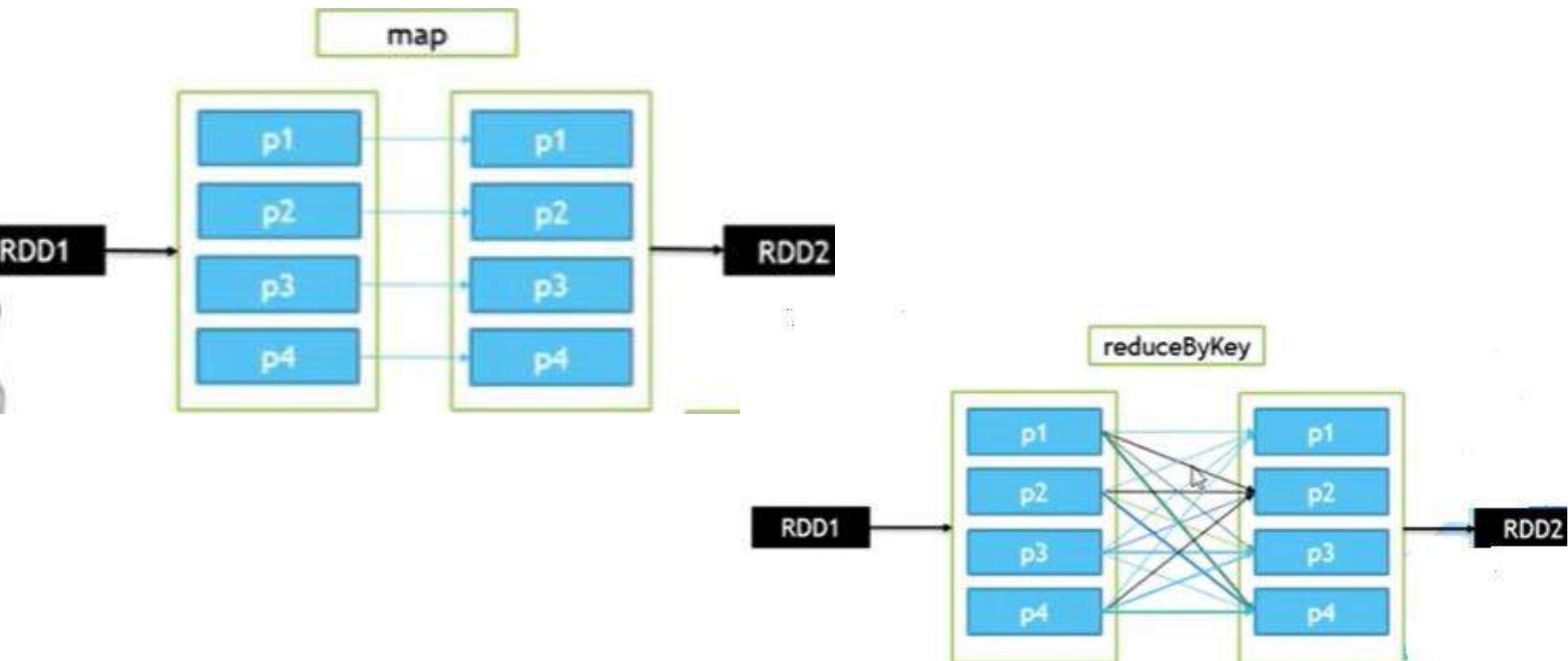
## 2. Wide Transformations

- `reduceBy()`
- `union()`
- `Intersection()`
- `Join()`



**Actions:** Actions instruct Apache Spark to apply **computation** and pass the result or an exception back to the driver RDD. Few of the actions include:**collect()**, **count()**, **take()**, **first()**

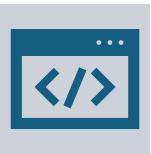
# Narrow & Wide Transformations



# RDD: Resilient Distributed Dataset



RDD Once done – It is **immutable** but can be overridden



If you want to **modify “data”** which is already there in specific RDD it can't be done , instead need to make **“new RDD”**



If data to be overridden –it can be successfully performed (means data from another file can be called in to current RDD, once data is overridden the content of current RDD will be updated with new called file data)

## Transformations :

```
var a = sc.textFile("hdfs://.....")
var b = a.flatMap(....)
var c = b.distinct(....)
```

- Here **a** is “**RDD**”
- **Transformation** = Process of making RDD's
  - In above example; b & c are new RDD.
  - Flatmap = Is Transformation
- **Actions** = Processing the data inside the cluster

# RDD:

- ▶ Fault tolerant distributed dataset.
- ▶ Lazy Evaluation
- ▶ Caching
- ▶ In memory computation
- ▶ Immutability
- ▶ Partitioning

# RDD

- Unless Actions are not called no Transformations get executed – **lazy Evaluation**.
- **RDD (2 operations)**
  - **Transformations** = process of **making chain of RDD's**
  - Ex: From a RDD making b RDD
  - During Transformations = **No executions take place** (in cluster)
  - **Actions** = Actual Executions take place
  - During this time of transformations – **spark makes (RDD's Map) using DAG** (which RDD made – sequence of RDD's).

Transformations :

```
var a = sc.textFile("hdfs://...../abc.txt") --> RDD0
var b = a.filter(....) --> RDD1
var c = b.distinct(....) --> RDD2
```

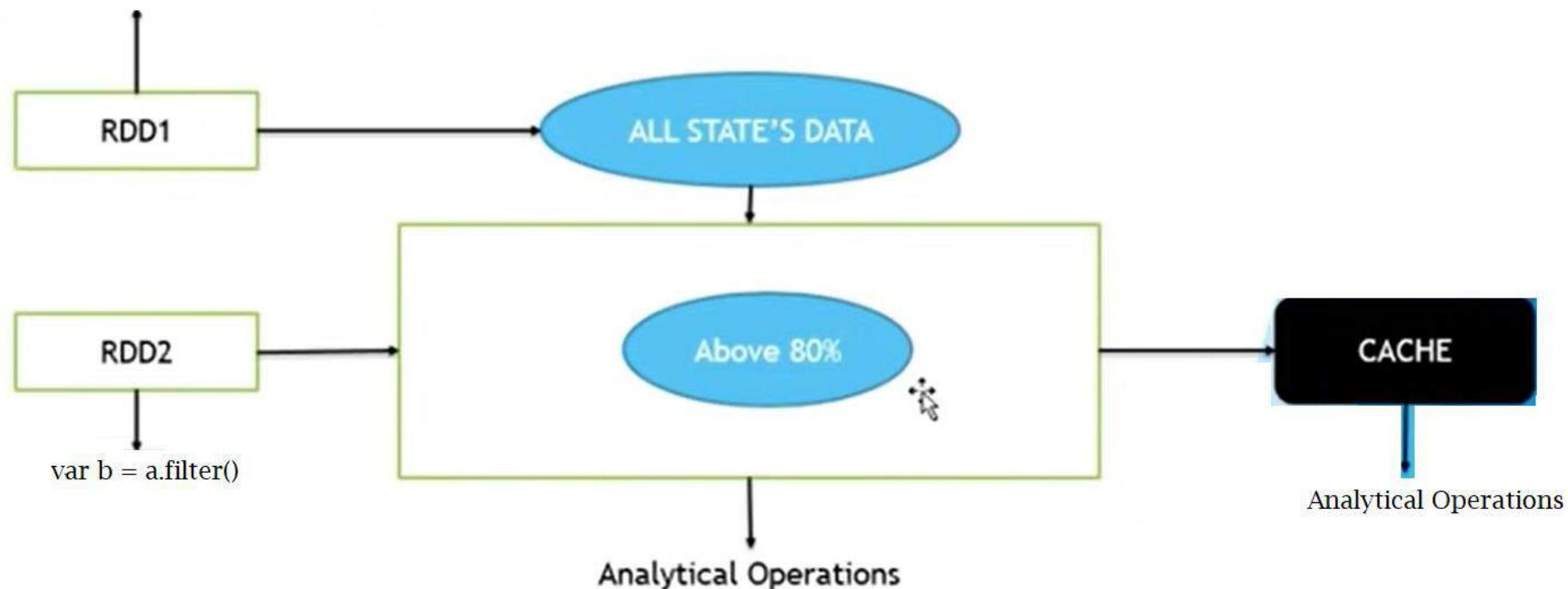
Actions

```
c.collect()
```

# Caching:

- If any modifications to be made in data we need to make new RDD always .
- Dataset used frequently is transferred to cache.

```
Var a = sc.textFile(hdfs://...)
```

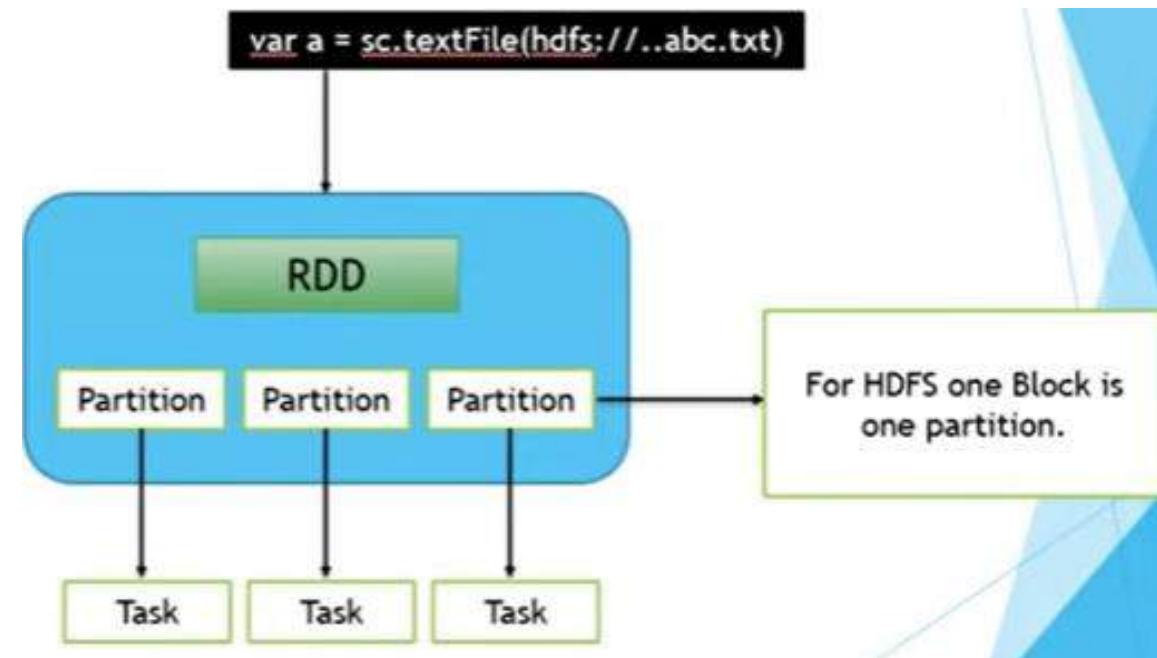


## Immutable:

- If any modifications to be made in data we need to make new RDD always (Immutable).
- Means RDD's are immutable(we cant change RDD).

# Partitioning

- ▶ Data is split up into partitions.
- ▶ Partition size depends on data source you are using.
- ▶ For HDFS one Block is one Partition.
- ▶ Single partition -- > Single Task

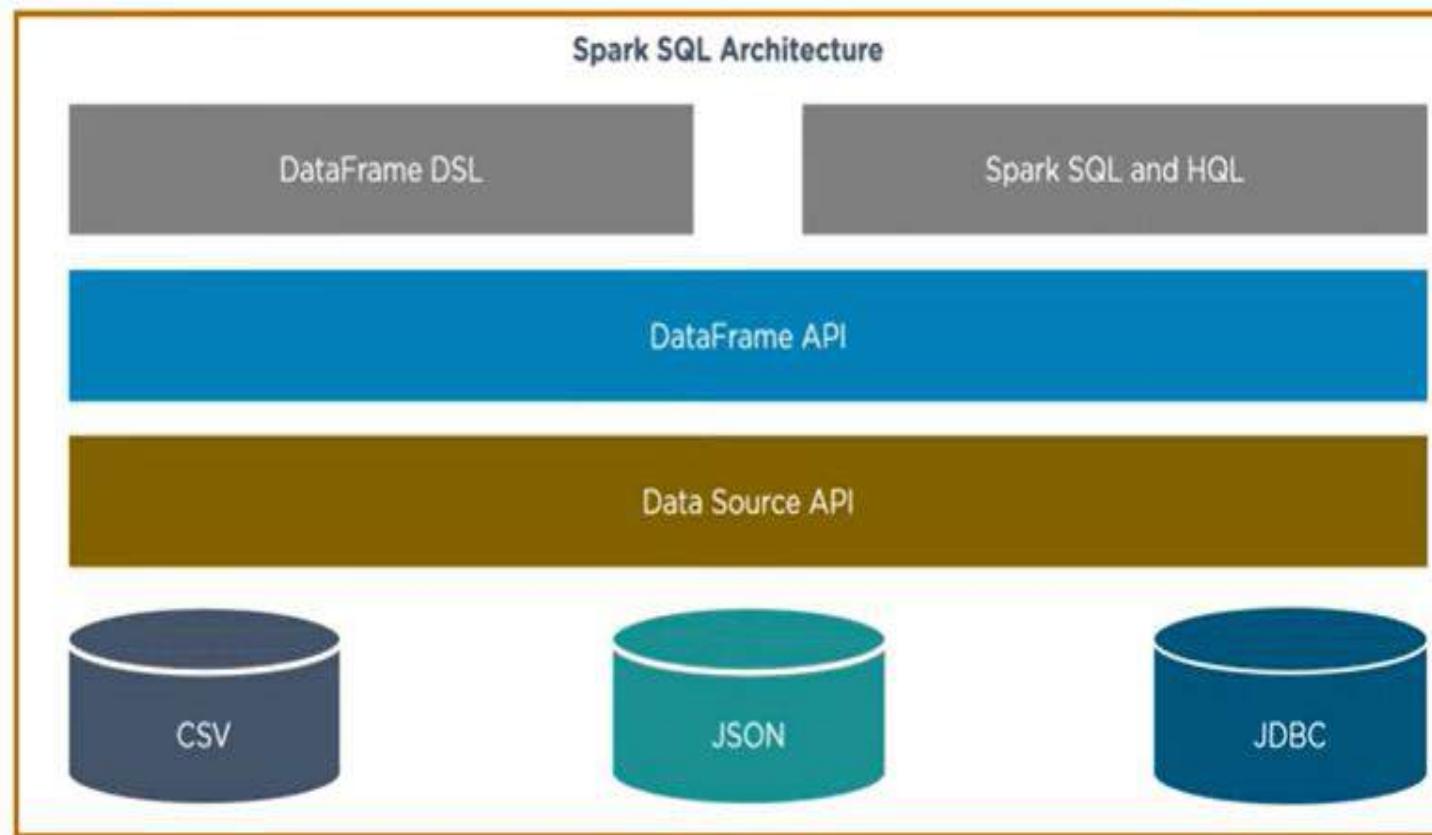


# Spark SQL – Structured Data

- ▶ Spark's package for working with structured data.
- ▶ Supports many sources of data including Hive tables, parquet, JSON.
- ▶ Allows developers to intermix SQL with programmatic data manipulations supported by RDDs in Python, Scala and Java.
- ▶ Shark was older SQL on Spark project.

# Spark SQL:

Spark SQL framework component is used for structured and semi-structured data processing



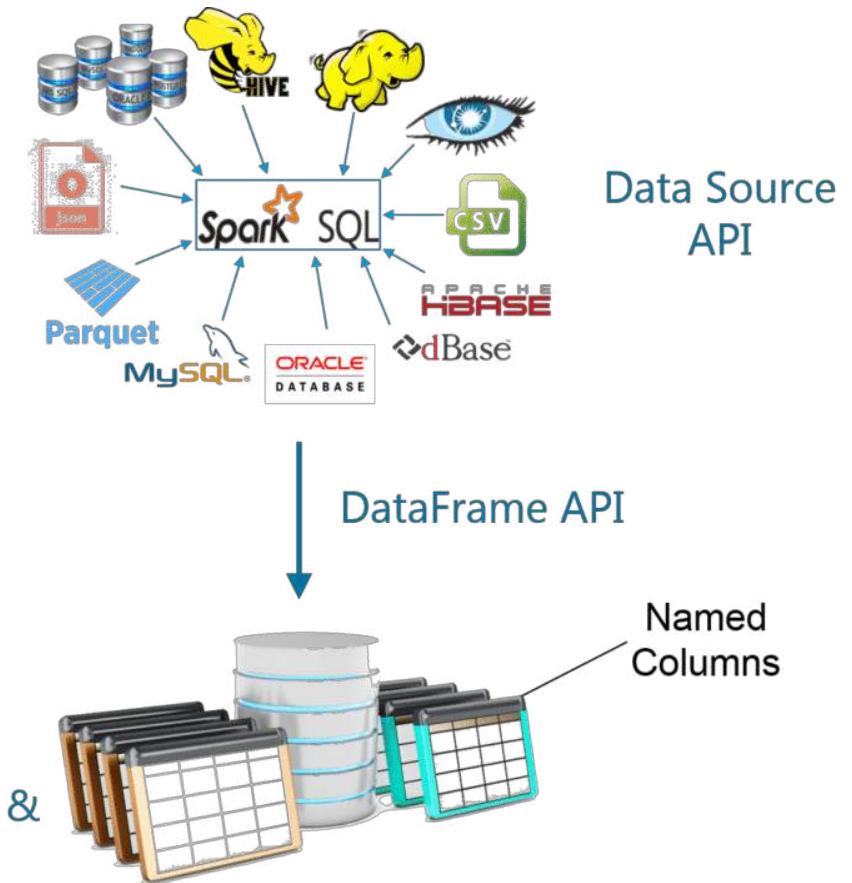
## 2. Spark SQL

Spark SQL is a new module in Spark which integrates relational processing with Spark's functional programming API. It supports querying data either via SQL or via the Hive Query Language.

Spark SQL integrates relational processing with Spark's functional programming. Further, it provides support for various data sources and makes it possible to weave SQL queries with code transformations thus resulting in a very powerful tool. The following are the four libraries of Spark SQL.

1. Data Source API
2. DataFrame API
3. Interpreter & Optimizer
4. SQL Service

# Spark SQL



**Figure:** The flow diagram represents a Spark SQL process using all the four libraries in sequence

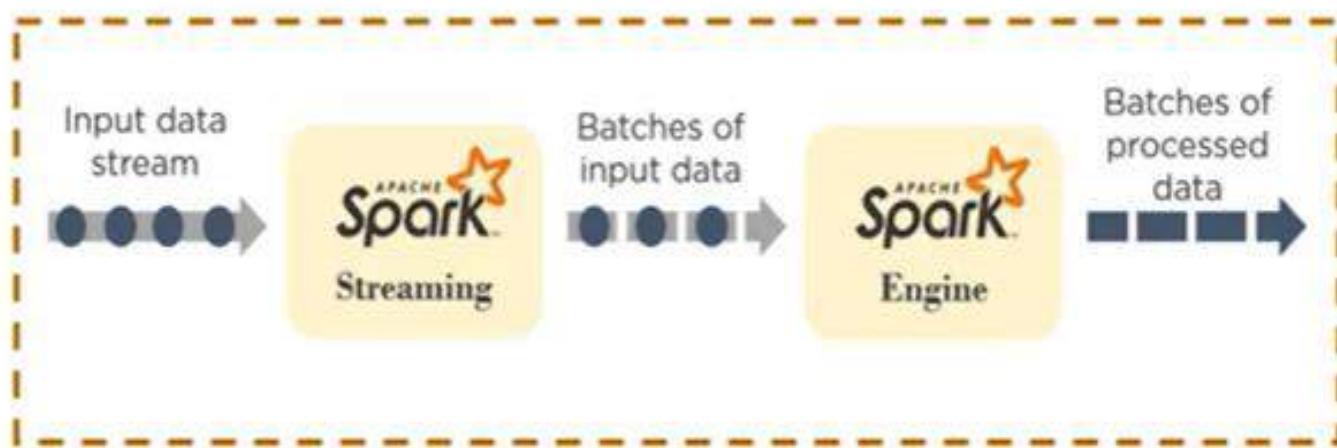
## Spark Streaming:

- ▶ Enables processing of live streams of data e.g. log files generated by production web servers.
- ▶ APIs are quite similar to spark core's RDD APIs.

# Spark Streaming:

Spark Streaming is a lightweight API that allows developers to perform batch processing and real-time streaming of data with ease

Provides secure, reliable, and fast processing of live data streams



### 3. Spark Streaming

*Spark Streaming* is the component of Spark which is used to process real-time streaming data. Thus, it is a useful addition to the core Spark API. It enables high-throughput and fault-tolerant stream processing of live data streams.

# MLLib

- ▶ Provides multiple types of machine learning algorithms.

# Spark MLlib:

MLlib is a low-level machine learning library that is simple to use, is scalable, and compatible with various programming languages

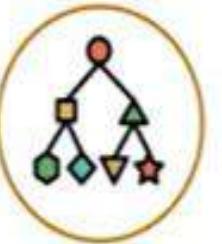
MLlib eases the deployment and development of scalable machine learning algorithms



It contains machine learning libraries that have an implementation of various machine learning algorithms



Clustering



Classification



Collaborative Filtering

## 4. MLlib (Machine Learning)

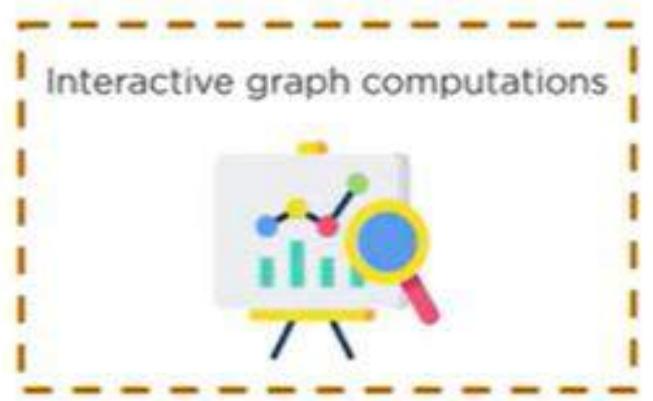
MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark.

# Graphx

- ▶ Library for manipulating Graphs.
- ▶ Extends Spark RDD API.
- ▶ Provides various operators for manipulating Graphs.

# Spark GraphX:

GraphX is Spark's own Graph Computation Engine and data store



## 5. GraphX

- *GraphX* is the Spark API for graphs and graph-parallel computation. Thus, it extends the Spark RDD with a Resilient Distributed Property Graph.
- The property graph is a directed multigraph which can have multiple edges in parallel. Every edge and vertex have user defined properties associated with it.
- Here, the parallel edges allow multiple relationships between the same vertices. At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge.



# Transformations

| Function       | Description                                                                                           |
|----------------|-------------------------------------------------------------------------------------------------------|
| map()          | Returns a new RDD by applying the function on each data element                                       |
| filter()       | Returns a new RDD formed by selecting those elements of the source on which the function returns true |
| reduceByKey()  | Aggregates the values of a key using a function                                                       |
| groupByKey()   | Converts a (key, value) pair into a (key, <iterable value>) pair                                      |
| union()        | Returns a new RDD that contains all elements and arguments from the source RDD                        |
| intersection() | Returns a new RDD that contains an intersection of the elements in the datasets                       |



# Actions

| Function           | Description                                                                    |
|--------------------|--------------------------------------------------------------------------------|
| count()            | Gets the number of data elements in an RDD                                     |
| collect()          | Gets all the data elements in an RDD as an array                               |
| reduce()           | Aggregates data elements into an RDD by taking two arguments and returning one |
| take(n)            | Fetches the first $n$ elements of an RDD                                       |
| foreach(operation) | Executes the operation for each data element in an RDD                         |
| first()            | Retrieves the first data element of an RDD                                     |

# Spark Context

- 1. Entry Point:** SparkContext is the entry point for Spark functionality in any Spark application.
- 2. Connection to Cluster:** It establishes a connection to the Spark cluster.
- 3. Resource Management:** SparkContext manages the resources allocated to the Spark application on the cluster.
- 4. RDD Creation:** It is used to create RDDs (Resilient Distributed Datasets), which are the fundamental data structures in Spark.
- 5. Driver Program Coordination:** The SparkContext is created by the driver program to coordinate the execution of Spark jobs on the cluster.
- 6. Access to Services:** It provides access to various Spark services like broadcast variables, accumulators, and shared variables.
- 7. Configuration:** SparkContext allows setting various configurations for Spark application execution.
- 8. Lifecycle Management:** It manages the lifecycle of the Spark application, including starting, running, and stopping the application.
- 9. Fault Tolerance:** SparkContext ensures fault tolerance by monitoring the execution of tasks and recovering from failures.
- 10. Parallel Operations:** SparkContext enables parallel operations on distributed data by leveraging the capabilities of the Spark cluster.



# How to Create RDD in Spark?

There are following three processes to create Spark RDD.

1. Using **parallelized collections**
2. From **external datasets** (viz . other external storage systems like a shared file system, HBase or HDFS)
3. From existing **Apache Spark RDDs**

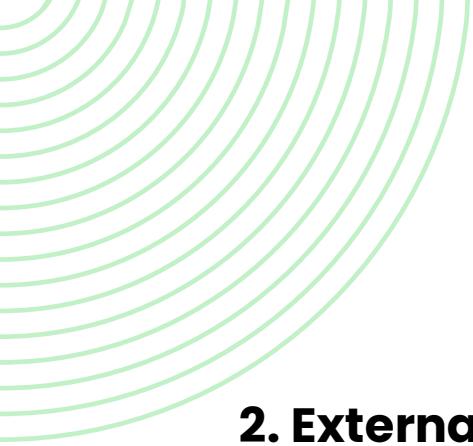
## 1. Parallelized Collections

create parallelized collections by calling **parallelize method of SparkContext interface** on the existing collection of driver program in Java, Scala or Python.

### Example:

To hold the numbers 2 to 6 as parallelized collection

```
val collection = Array(2, 3, 4, 5, 6)
val prData =
 spark.sparkContext.parallelize(collection)
```



# How to Create RDD in Spark?

## 2. External Datasets

Apache Spark can create distributed datasets from any **Hadoop supported file storage** which may include:

- Local file system
- HDFS
- Cassandra
- HBase
- Amazon S3

**Spark supports file formats like**

- Text files
- Sequence Files
- CSV
- JSON
- Any Hadoop Input Format

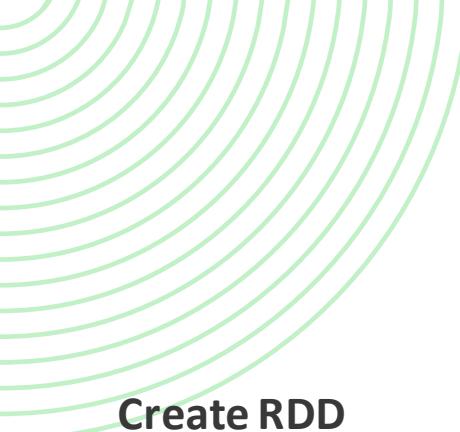


# How to Create RDD in Spark?

## 3. From Existing RDDs

RDD is immutable; hence you can't change it. However, using transformation, you can create a new RDD from an existing RDD. As no change takes place due to mutation, it maintains the consistency over the cluster. Few of the operations used for this purpose are:

- map
- filter
- count
- distinct
- flatmap



## Practical demo of RDD operations

### Create RDD

```
scala> val rdd1 = sc.parallelize(List(23, 45, 67, 86, 78, 27, 82, 45, 67, 86))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:27
```

Here, sc denotes SparkContext and each element is copied to form RDD.

### Read result

We can read the result generated by RDD by using the collect operation.

```
scala> rdd1.collect
res0: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86)

scala> ■
```

## Count

The count action is used to get the total number of elements present in the particular RDD.

```
scala> rdd1.count
res1: Long = 10
```

```
scala> █
```

## Distinct

Distinct is a type of transformation that is used to get the unique elements in the RDD.

```
scala> rdd1.distinct.collect
res3: Array[Int] = Array(82, 86, 78, 27, 23, 45, 67)
```

```
scala> █
```

## Filter

Filter transformation creates a new dataset by selecting the elements according to the given

```
scala> rdd1.filter(x => x < 50).collect
res5: Array[Int] = Array(23, 45, 27, 45)
```

```
scala> █
```

## **sortBy**

sortBy operation is used to arrange the elements in ascending order when the condition is true and in descending order when the condition is false.

```
scala> rdd1.sortBy(x => x, true).collect
res6: Array[Int] = Array(23, 27, 45, 45, 67, 67, 78, 82, 86, 86)
```

```
scala> rdd1.sortBy(x => x, false).collect
res7: Array[Int] = Array(86, 86, 82, 78, 67, 67, 45, 45, 27, 23)
```

```
scala> ■
```

## **Reduce**

Reduce action is used to summarize the RDD based on the given formula.

```
scala> rdd1.reduce((x, y) => x + y)
res8: Int = 606
```

```
scala> ■
```

## **Map**

Map transformation processes each element in the RDD according to the given condition and creates a new RDD.

```
scala> rdd1.map(x => x + 1).collect
res9: Array[Int] = Array(24, 46, 68, 87, 79, 28, 83, 46, 68, 87)
```

```
scala> ■
```

## Union, intersection, and cartesian

Let's create another RDD.

```
val rdd2 = sc.parallelize(List(25,73, 97, 78, 27, 82))
```

- Union operation combines all the elements of the given two RDDs.
- Intersection operation forms a new RDD by taking the common elements in the given RDDs.
- Cartesian operation is used to create a cartesian product of the required RDDs.

```
scala> rdd1.union(rdd2).collect
res12: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86, 25, 73, 97, 78
, 27, 82)
```

```
scala> rdd1.intersection(rdd2).collect
res13: Array[Int] = Array(82, 78, 27)
```

```
scala> rdd1.cartesian(rdd2).collect
res14: Array[(Int, Int)] = Array((23,25), (23,73), (23,97), (45,25), (45,73), (4
5,97), (67,25), (67,73), (67,97), (86,25), (86,73), (86,97), (78,25), (78,73),
(78,97), (23,78), (23,27), (23,82), (45,78), (45,27), (45,82), (67,78), (67,27),
(67,82), (86,78), (86,27), (86,82), (78,78), (78,27), (78,82), (27,25), (27,73),
(27,97), (82,25), (82,73), (82,97), (45,25), (45,73), (45,97), (67,25), (67,73)
, (67,97), (86,25), (86,73), (86,97), (27,78), (27,27), (27,82), (82,78), (82,27
), (82,82), (45,78), (45,27), (45,82), (67,78), (67,27), (67,82), (86,78), (86,2
7), (86,82))
```

```
scala> ■
```

## First

First is a type of action that always returns the first element of the RDD.

```
scala> rdd1.first()
res15: Int = 23
```

```
scala> █
```

## Take

Take action returns the first n elements in the RDD.

```
scala> rdd1.take(5)
res16: Array[Int] = Array(23, 45, 67, 86, 78)
scala> █
```



# Pair RDD Operations

- Pair RDDs are a unique class of data structure in Spark that take the form **of key-value pairs**
- most **real-world data** is in the form **of Key/Value pairs**, Pair RDDs are practically employed more frequently.
- The terms "key" and "value" are different by the Pair RDDs. The **value is data**, whereas the **key is an identifier**.

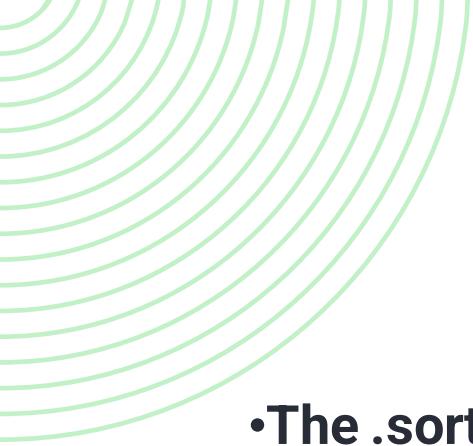
## Transformations in Pair RDDs

We must utilize operations that use keys and values since Pair RDDs are built from many tuples.

Following are the widely used Transformation on a Pair RDD:

### 1. The `.reduceByKey()` Transformation

For each key in the data, the `.reduceByKey()` transformation runs multiple parallel operations, **combining the results for the same keys**. **The task is carried out using a lambda or anonymous function**. Since it is a transformation, the outcome is an RDD.



# Pair RDD Operations

- **The .sortByKey() Transformation**

Using the keys from key-value pairs, the.sortByKey() transformation **sorts the input data in ascending or descending order**. It returns a unique RDD as a result.

- **The .groupByKey() Transformation**

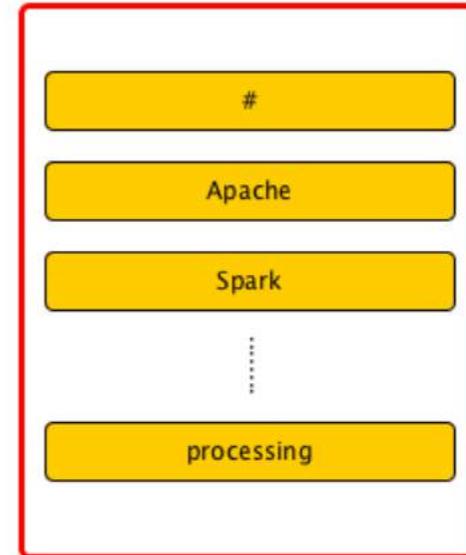
The.groupByKey() transformation **groups all values in the given data with the same key**. As a result, a **new RDD is returned**. For instance, the.groupByKey() function will be useful if we need to extract all the Cultural Members from a list of committee members.

# Actions in Pair RDDs

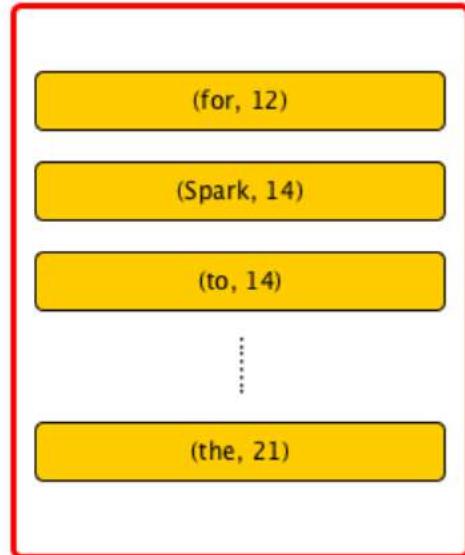
## The countByKey() Action

The number of values linked with each key in the provided data is counted using the.`countByKey()` action. This operation returns a dictionary, which can be iterated using loops to retrieve the keys and values. We can also utilize dictionary methods like`.keys()`,`.values()`, and`.items` because the result is a dictionary ()�.

RDD of Strings



RDD of Pairs



## **DataFrame:**

- A Data Frame is used for storing data in tables.
- It is equivalent to a table in a relational database but with richer optimization.
- It is a data abstraction and domain-specific language (DSL) applicable to a structure and semi-structured data.
- It is a distributed collection of data in the form of named column and row.
- It has a matrix-like structure whose column may be different types (numeric, logical, factor, or character ).
- We can say the data frame has a two-dimensional array-like structure where each column contains the value of one variable and row contains one set of values for each column.
- It combines features of lists and matrices.

## **RDD:**

- It is the representation of a set of records, an immutable collection of objects with distributed computing.
- RDD is a large collection of data or RDD is an array of references for partitioned objects.
- Each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster.
- RDDs are fault-tolerant i.e. self-recovered/recomputed in the case of failure.
- The dataset could be data loaded externally by the users which can be in the form of JSON file, CSV file, text file or database via JDBC with no specific data structure.

# Differentiation: RDD vs Datasets vs DataFrame

| Basis               | RDD                                                                                    | Datasets                                                                                           | DataFrame                                                                      |
|---------------------|----------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| Inception Year      | RDD came into existence in the year 2011.                                              | Datasets entered the market in the year 2013.                                                      | DataFrame came into existence in the year 2015.                                |
| Meaning             | RDD is a collection of data where the data elements are distributed without any schema | Datasets are distributed collections where the data elements are organized into the named columns. | Datasets are basically the extension of DataFrames with added features         |
| Optimization        | In case of RDDs, the developers need to manually write the optimization codes.         | Datasets use catalyst optimizers for optimization.                                                 | Even in the case of DataFrames, catalyst optimizers are used for optimization. |
| Defining the Schema | In RDDs, the schema needs to be defined manually.                                      | The schema is automatically defined in case of Datasets                                            | The schema is automatically defined in DataFrame                               |



## Advantages of RDD

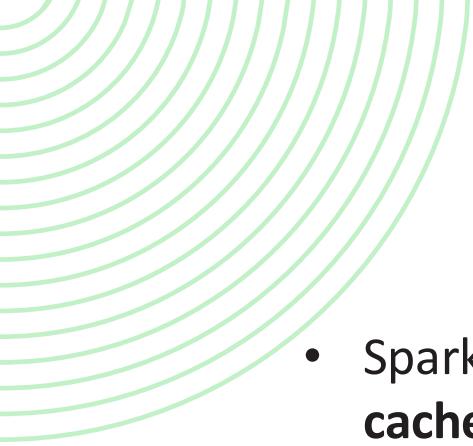
There are multiple advantages of RDD in Spark. We have covered few of the important ones here

- RDD aids in increasing the **execution speed** of Spark.
- RDDs are the **basic unit of parallelism** and hence help in achieving the **consistency of data**.
- RDDs help in **performing and saving the actions** separately
- They are **persistent** as they can be used repeatedly.



# Limitation of RDD

- There is **no input optimization** available in RDDs
- One of the biggest limitations of RDDs is that the **execution process does not start instantly.**
- **No changes can be made** in RDD once it is **created**.
- **RDD lacks enough storage memory.**
- The **run-time type safety** is absent in RDDs.



# RDD Persistence

- Spark RDD persistence is an **optimization technique** which **saves** the result of RDD evaluation in **cache memory**.
- Spark provides a convenient way to work on the dataset by **persisting it in memory across operations**.
- we can **also reuse** them in other tasks on that dataset.
- We can use either **persist()** or **cache()** method to mark an RDD **to be persisted**.
- In any case, if the partition of an RDD is lost, it will automatically be **recomputed** using the **transformations that originally created it**.
- There is an **availability of different storage levels** which are used to **store persisted RDDs**.
- Use these levels by passing a **StorageLevel** object (Scala, Java, Python) to persist().
- The **cache()** method is used for the **default storage level**, which is **StorageLevel.MEMORY\_ONLY**.



## Example

RDD1: Contains a list of numbers [1, 2, 3, 4, 5].

RDD2: Created by doubling each element of RDD1.

RDD3: Created by squaring each element of RDD2.

```
Create RDD1
```

```
rdd1 = sc.parallelize([1, 2, 3, 4, 5])
```

```
Create RDD2 by performing a transformation on RDD1
```

(doubling each element)

```
rdd2 = rdd1.map(lambda x: x * 2)
```

```
Create RDD3 by performing a transformation on RDD2
```

(squaring each element)

```
rdd3 = rdd2.map(lambda x: x ** 2)
```

Now, let's say we perform an action on RDD3, such as counting the number of elements:

```
Perform an action on RDD3 (e.g., count the number of elements)
```

```
count = rdd3.count()
```



## Example

Here's an example of how you can cache RDD3:

```
Assuming RDD3 is already created
rdd3.persist()
```

```
Perform actions on RDD3
count = rdd3.count()
Other operations on RDD3...
```

- Intermediate results are stored in memory or disk, leading to **faster and more efficient** computations when accessing RDD3 multiple times.

**Note: rdd.cache() is same as rdd.persist()**



# Need of Persistence in Apache Spark

- In Spark, we can use some RDD's **multiple times**.
- We repeat the same process of **RDD evaluation** each time it required or brought into action.
- This task can be **time and memory consuming**, especially for iterative algorithms that look at data multiple times.
- **To solve** the problem of **repeated computation** the technique of persistence came into the picture.
- There are some **advantages of RDD caching** and persistence mechanism in spark. It makes the whole system
  - 1. Time efficient
  - 2. Cost efficient
  - 3. Lessen the execution time.

# The set of storage levels:

| Storage Level                             | Description                                                                                                                                                                                         |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MEMORY_ONLY                               | It stores the RDD as deserialized Java objects in the JVM. This is the default level. If the RDD doesn't fit in memory, some partitions will not be cached and recomputed each time they're needed. |
| MEMORY_AND_DISK                           | It stores the RDD as deserialized Java objects in the JVM. If the RDD doesn't fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.             |
| MEMORY_ONLY_SER<br>(Java and Scala)       | It stores RDD as serialized Java objects ( i.e. one-byte array per partition). This is generally more space-efficient than deserialized objects.                                                    |
| MEMORY_AND_DISK_SER<br>(Java and Scala)   | It is similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them.                                                                                |
| DISK_ONLY                                 | It stores the RDD partitions only on disk.                                                                                                                                                          |
| MEMORY_ONLY_2,<br>MEMORY_AND_DISK_2, etc. | It is the same as the levels above, but replicate each partition on two cluster nodes.                                                                                                              |
| OFF_HEAP (experimental)                   | It is similar to MEMORY_ONLY_SER, but store the data in off-heap memory. The off-heap memory must be enabled.                                                                                       |



# Persistence Storage Levels

- MEMORY\_ONLY (default) – same as cache

**rdd.persist(StorageLevel.MEMORY\_ONLY) or rdd.persist()**

- MEMORY\_AND\_DISK – Stores partitions on disk which do not fit in memory (This is also called Spilling)

**rdd.persist(StorageLevel.MEMORY\_AND\_DISK)**

- DISK\_ONLY – Stores all partitions on the disk

**rdd.persist(StorageLevel.DISK\_ONLY)**

## -MEMORY\_ONLY\_SER and MEMORY\_AND\_DISK\_SER

- Persisting the **RDD in a serialized** (binary) form helps to **reduce the size of the RDD**, thus making space for more RDD to be persisted in the cache memory. So these two memory formats are **space-efficient**.
- They are **not time-efficient** because we need **to incur the cost of time** involved in **deserializing the data**.



# Partition Replication

Stores the partition on two nodes.

**DISK\_ONLY\_2**

**MEMORY\_AND\_DISK\_2**

**MEMORY\_ONLY\_2**

**MEMORY\_AND\_DISK\_SER\_2**

**MEMORY\_ONLY\_SER\_2**

- These options stores a replicated copy of the RDD into some other Worker Node's cache memory as well.
- it helps to recompute the RDD if the other worker node goes down.

## Unpersisting the RDD

- To stop persisting and remove from memory or disk
- To change an RDD's persistence level

**rdd.unpersist()**

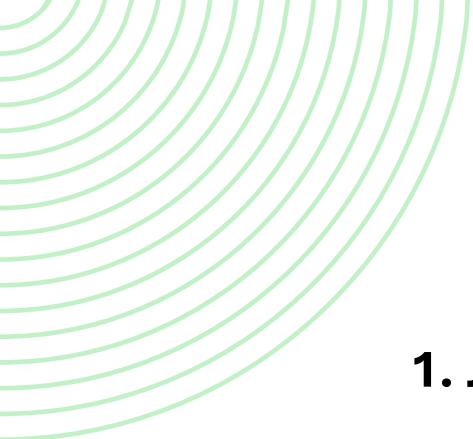
# Representation of the Storage level

| Storage Level       | Space used | CPU time | In memory | On-disk | Serialized | Recompute some partitions |
|---------------------|------------|----------|-----------|---------|------------|---------------------------|
| MEMORY_ONLY         | High       | Low      | Y         | N       | N          | Y                         |
| MEMORY_ONLY_SER     | Low        | High     | Y         | N       | Y          | Y                         |
| MEMORY_AND_DISK     | High       | Medium   | Some      | Some    | Some       | N                         |
| MEMORY_AND_DISK_SER | Low        | High     | Some      | Some    | Y          | N                         |
| DISK_ONLY           | Low        | High     | N         | Y       | Y          | N                         |



# Serialization

- In distributed systems, **data transfer over the network is the most common task.**
- If this is **not handled efficiently**, you may end up facing numerous problems, like **high memory usage, network bottlenecks, and performance issues.**
- **Serialization** refers to **converting objects into a stream of bytes** and vice-versa (de-serialization) in an optimal way to transfer it over nodes of network or store it in a file/memory buffer.
- Spark provides **two serialization libraries** and modes are supported and configured through **spark.serializer** property.



# Two serialization libraries

## 1. Java serialization (default)

The serialization of a class is enabled by the class implementing the [java.io.Serializable](#) interface.

Java serialization is slow and leads to large serialized formats for many classes. We can fine-tune the performance by extending [java.io.Externalizable](#).

## 2. Kryo serialization (recommended by Spark)

- Kryo is a Java serialization framework that focuses on **speed, efficiency, and a user-friendly API**.
- Kryo has **less memory footprint**, which becomes very important when you are shuffling and caching a large amount of data.
- It is **not natively supported to serialize to the disk**.
- A class is never serialized **only object of a class is serialized**.



```
from pyspark import SparkConf, SparkContext
from pyspark.serializers import KryoSerializer

conf = SparkConf().setAppName("my_app").setMaster("local")

conf.set("spark.serializer", KryoSerializer.getName())

sc = SparkContext(conf=conf)

Now we can create and process RDDs using Kryo serialization

Create RDD1
rdd1 = sc.parallelize([1, 2, 3, 4, 5])
```



# Shared Variable in Spark

- Variables those we want to share **throughout our cluster**
  - Transformation functions passed on variable, it executes on the specific remote cluster node.
  - Usually, it works on separate copies of all the variables those we use in functions.
  - These specific variables are precisely copied to each machine.
  - on the remote machine, no updates to the variables sent back to the driver program.
  - Therefore, it would be inefficient to support general, read-write shared variables across tasks.

**Two types of shared variables, such as:**

1. Broadcast Variables
2. Accumulators



## Broadcast Variables:

- Broadcast variables allow you to efficiently **distribute read-only data to all the worker nodes** in a Spark cluster.
- They are useful when you have large datasets or lookup tables that need to be shared across all nodes but don't change frequently.
- Broadcast variables are **broadcasted to all the worker nodes** once and then cached locally on each node for future reference, which **reduces network overhead**.
- They are **immutable** and can be safely used in parallel operations.

```
Creating a broadcast variable
broadcast_var = sc.broadcast([1, 2, 3, 4, 5])

Accessing the broadcast variable value on each worker node
def process_data(element):
 broadcast_value = broadcast_var.value

 processed_data = [x * 2 for x in element if x in
 broadcast_value]
 return processed_data

Applying a transformation operation using the broadcast
variable
rdd = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
processed_rdd = rdd.map(process_data)
```



# Accumulators:

- Accumulators are variables that are only "added" to through **an associative and commutative operation** and are only "read" **by the driver program**.
- They are primarily used for **aggregating information across all the worker nodes** during parallel computations, such as **counting occurrences or summing values**.
- Accumulators are useful for tasks like **collecting statistics** or **monitoring the progress of a job**.

```
Creating an accumulator for counting occurrences
accum = sc.accumulator(0)
```

```
Defining a function to increment the accumulator
def process_data(element):
 global accum
 if 3 in element:
 accum += 1
 return element
```

```
Applying a transformation operation using the accumulator
rdd = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
processed_rdd = rdd.map(process_data)
```

```
Performing an action to trigger the accumulation
processed_rdd.count()
```

```
Accessing the value of the accumulator in the driver
program
print("Occurrences of 3:", accum.value)
```

## DAG: Directed Acyclic Graph

Spark creates a graph when you enter code in spark console.

When an action is called on Spark RDD, Spark submits graph to DAG scheduler.

Operators are divided into stages of task in DAG Scheduler.

The Stages are passed on Task Scheduler, Which launches task through CM.

# Importance of DAG in Spark

The DAG plays a critical role in this process by providing **a logical execution plan for the job.**

- The DAG breaks the job down into a **sequence of stages**, where **each stage** represents a **group of tasks** that can be **executed independently** of each other. The **tasks within each stage can be executed in parallel across the machines**.
- The DAG allows Spark to perform various optimizations, such as **pipelining, task reordering, and pruning unnecessary operations**, to improve the efficiency of the job execution.
- By breaking down the job into smaller stages and tasks, Spark can execute them in parallel and distribute them across a cluster of machines for faster processing.

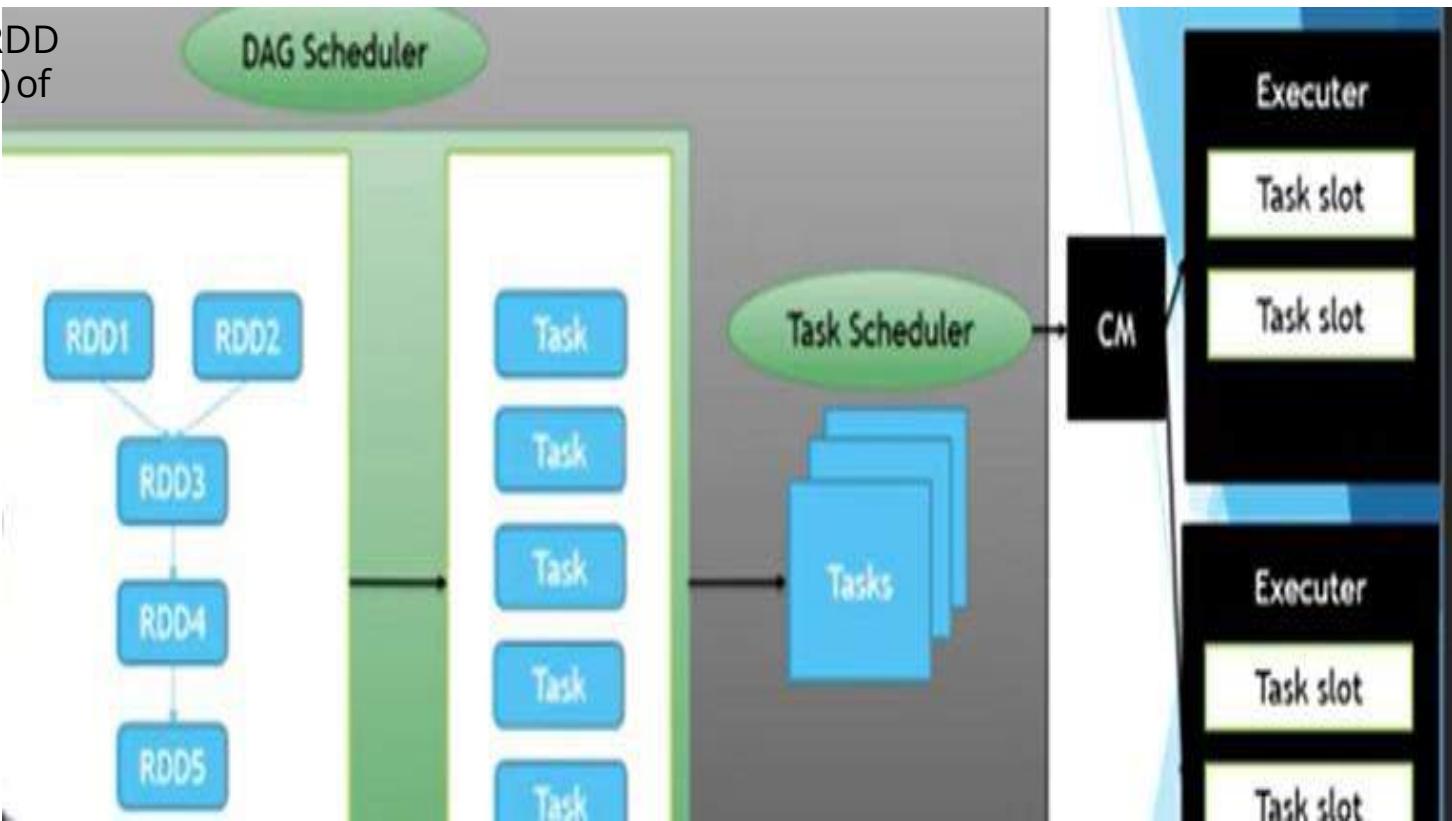
# How Apache Spark builds a DAG and Physical Execution Plan ?

1. User **submits a spark application** to the Apache Spark.
2. Driver is the module that takes in the **application from Spark side**.
3. Driver **identifies transformations and actions present** in the spark application. These identifications are the tasks.
4. Based on the flow of program, these **tasks are arranged in a graph like structure** with directed flow of execution from task to task forming **no loops** in the graph (also called DAG).  
DAG is pure logical.
5. This **logical DAG is converted to Physical Execution Plan**. Physical Execution Plan contains stages.
6. **DAG Scheduler** creates a **Physical Execution Plan from the logical DAG**. Physical Execution Plan contains tasks and are bundled to be sent to nodes of cluster.

# DAG Scheduler

DAG Scheduler is responsible for transforming a sequence of RDD transformations and actions into a directed acyclic graph (DAG) of stages and tasks

- Stages will be created.
- Which task need to put in which stage is decided by DAG Scheduler.
- Stage list will be created by DAG scheduler and submitted to Task scheduler.
- Each task will be executed via Cluster manager and cluster manager (CM) will launch it.
- To perform particular task, it needs data and resources.
- Cluster manager will help task scheduler to provide information from where the data need to be fetched in order to execute the tasks list and required resources check will be done by cluster manager.
- Later on task scheduler requests the CM to provide some slot to execute task .
- Executer run on slave machine. TO execute particular task executer has required resources



DAGScheduler transforms a **logical execution plan** (RDD lineage of dependencies built using RDD transformations) to a **physical execution plan** (using stages).

# Stages, Tasks, Dependencies

To work with the DAG Scheduler in Spark, you need to understand the following concepts:

## Stages:

- A stage represents a set of tasks that can be executed in parallel.
- There are two types of stages in Spark: **shuffle stages** and **non-shuffle stages**.
- **Shuffle** stages involve the **exchange of data between nodes**, while non-shuffle stages do not.

## Tasks:

- A task represents a single unit of work that can be executed on a single partition of an RDD.
- Tasks are the smallest units of parallelism in Spark.

## Dependencies:

- The dependencies between RDDs determine the order in which tasks are executed.
- There are two types of dependencies in Spark: **narrow dependencies** and **wide dependencies**.
- Narrow dependencies indicate that each partition of the parent RDD is used by at most one partition of the child RDD.
- while wide dependencies indicate that each partition of the parent RDD can be used by multiple partitions of the child RDD.

# Example



- By visualizing the DAG diagram, developers can better understand the logical execution plan of a Spark job and identify any potential bottlenecks or performance issues.

- The DAG diagram consists of five stages: Text RDD, Filter RDD, Map RDD, Reduce RDD, and Output RDD.
- The arrows indicate the dependencies between the stages, and each stage is made up of multiple tasks that can be executed in parallel.
- The Text RDD stage represents the initial loading of the data from a text file, and the subsequent stages involve applying transformations to the data to produce the final output.
- The Filter RDD stage applies a filter transformation to remove any unwanted data.
- The map RDD stage applies a map transformation to transform the remaining data.
- The reduce rdd stage applies a reduce transformation to aggregate the data and
- The output RDD stage writes the final output to a file.

# Fault tolerance with the help of DAG

- Spark achieves fault tolerance using the DAG by using a technique called lineage, which is the record of the transformations that were used to create an RDD.
- When a partition of an RDD is lost due to a node failure, Spark can use the lineage to rebuild the lost partition.
- The lineage is built up as the DAG is constructed, and Spark uses it to recover from any failures during the job execution.
- Spark uses the lineage to recompute the lost partitions.

To achieve fault tolerance, Spark uses two mechanisms:

1. RDD Persistence
2. Checkpointing

## Advantages of DAG in spark

### Efficient execution:

- The DAG allows Spark to break down a large-scale data processing job into smaller, independent tasks that can be executed in parallel.
- By executing the tasks in parallel, Spark can distribute the workload across multiple machines and perform the job much faster than if it was executed sequentially.

### Optimization:

- The DAG allows Spark to optimize the job execution by performing various optimizations, such as pipelining, task reordering, and pruning unnecessary operations.
- This helps to reduce the overall execution time of the job and improve performance.

### Fault tolerance:

- The DAG allows Spark to achieve fault tolerance by using the lineage to recover from node failures during the job execution.
- This ensures that the job can continue running even if a node fails, without losing any data.

### Reusability:

- The DAG allows Spark to reuse the intermediate results generated by a job.
- This means that if a portion of the data is processed once, it can be reused in subsequent jobs, thereby reducing the processing time and improving performance.

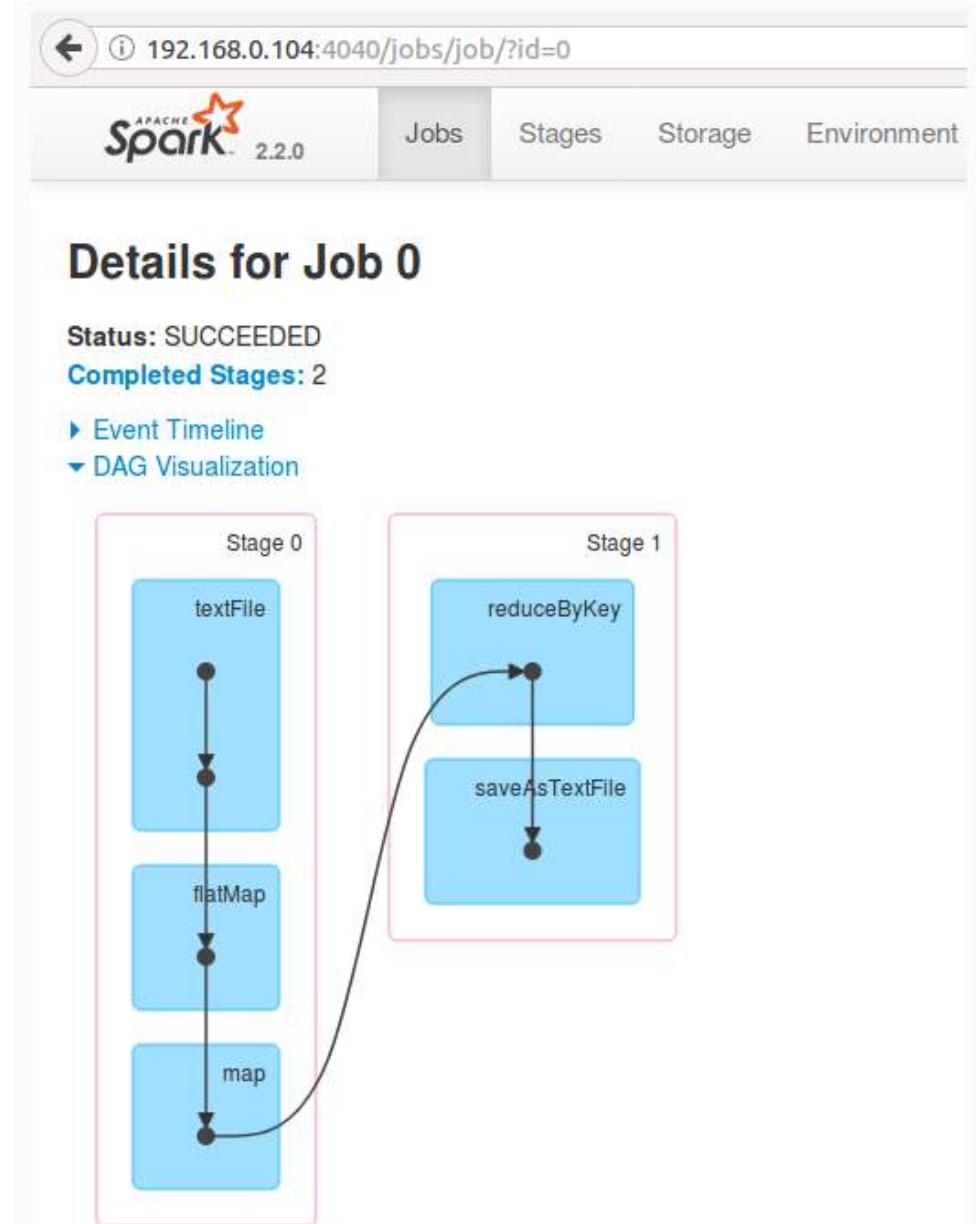
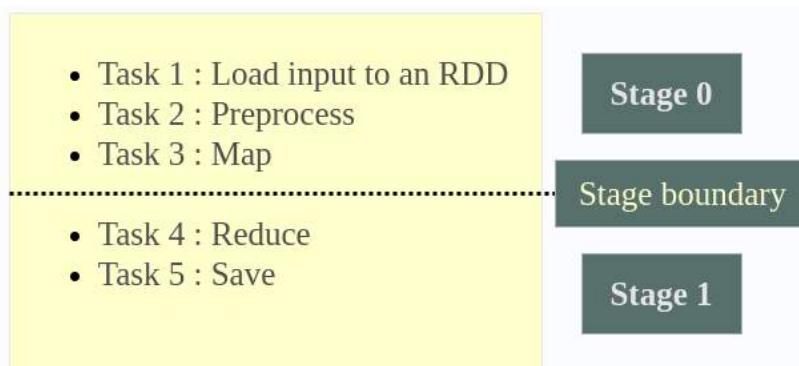
# Advantages of DAG in spark

## Visualization:

The DAG provides a visual representation of the logical execution plan of the job, which can help users to better understand the job and identify any potential bottlenecks or performance issues.

This could be visualized in Spark Web UI, once you run the WordCount example.

1. Hit the url `192.168.0.104:4040/jobs/`
2. Click on the link under Job Description.
3. Expand 'DAG Visualization'



To work with the DAG Scheduler, you can use the following approaches:

**Visualize the DAG:**

- You can use the Spark UI to visualize the DAG of a Spark job.
- This allows you to see the different stages and tasks that make up the job and identify any potential bottlenecks or performance issues.

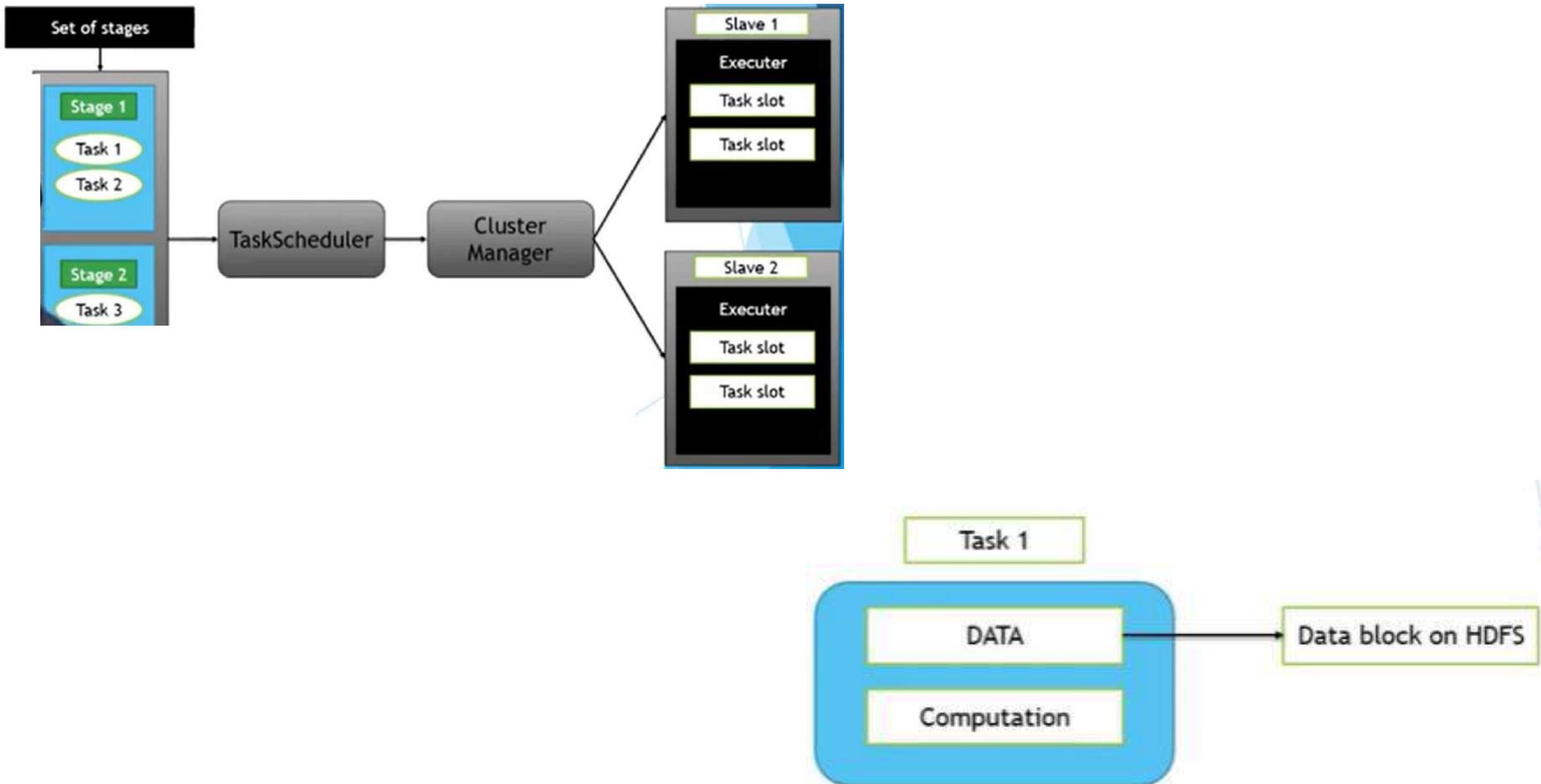
**Optimize the DAG:**

- You can optimize the DAG by using techniques such as pipelining, caching, and reordering of tasks to improve the performance of the job.

**Debug issues:**

- If you encounter issues with a Spark job, you can use the DAG Scheduler to identify the root cause of the problem.
- For example, you can use the Spark UI to identify any slow or failed stages and use this information to troubleshoot the issue.

# DAG Scheduler:



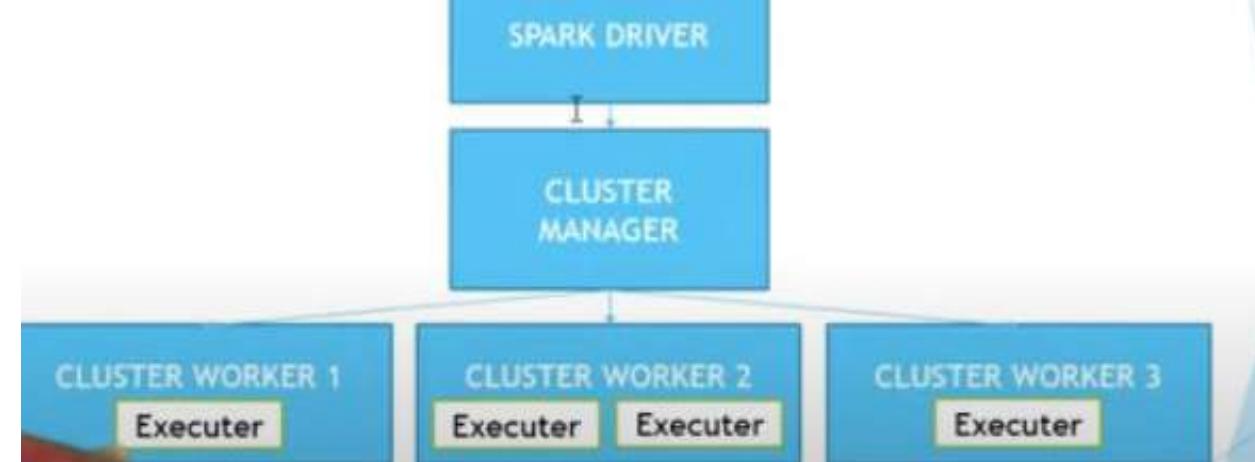
## Spark Context (Ticket To Park)

- ▶ Establish a connection to spark execution environment.
- ▶ It can be used to create RDDs, accumulators and broadcast variables.

# Spark Architecture

- **Spark Driver**

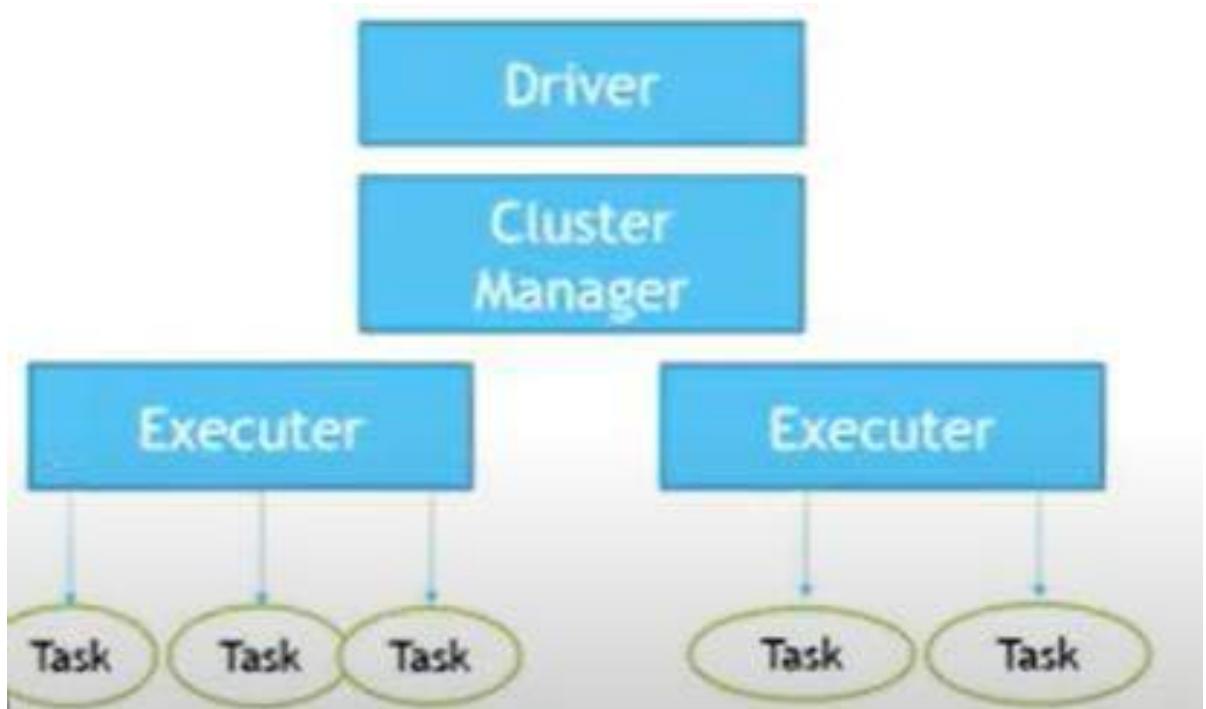
- It works on master slave concept.
- Master = Spark Driver
- Slave = Executer
- Spark Driver
- Runs on top of **master system**
- When we submit spark program , the **main method execution** is done by driver- (driver execution will be inside jvm)
- Creation of **Context** also takes place in driver
- Indirectly **Dag scheduler & task scheduler** are created at driver end itself.
- User **spark program (application)** will be converted into **actual spark job** by “Spark driver”
- **Cluster Manager- (CM)**
- It's a **pluggable component**
- Use – Yarn , Mesos, or Spark scheduler
- Jobs: **Resource Allocate or Deallocate**
- Driver – **send Tasks** – CM- To execute Task – **CM checks on which system data is present** – CM checks required resources are there or no (CM will check for availability of RAM on slave)- **If RAM available** – CM will assign the task to **Specific Executer Present** in worker Machine
- After the specific Task is executed –CM will **Deallocate Resources**



- **Executers:**

- Are **distributed**
- Run on **slave machines**
- Whenever particular **task need to be executed** it always executed inside executer.
- **Executer $\geq$ 1** can be present inside slave machine
- 1 Executer can have  $\geq$ 1 Task
- **Jobs:**
  - Data Read
  - Data Process
  - Results Writing (memory/disk/cache)

# Spark Program Execution:



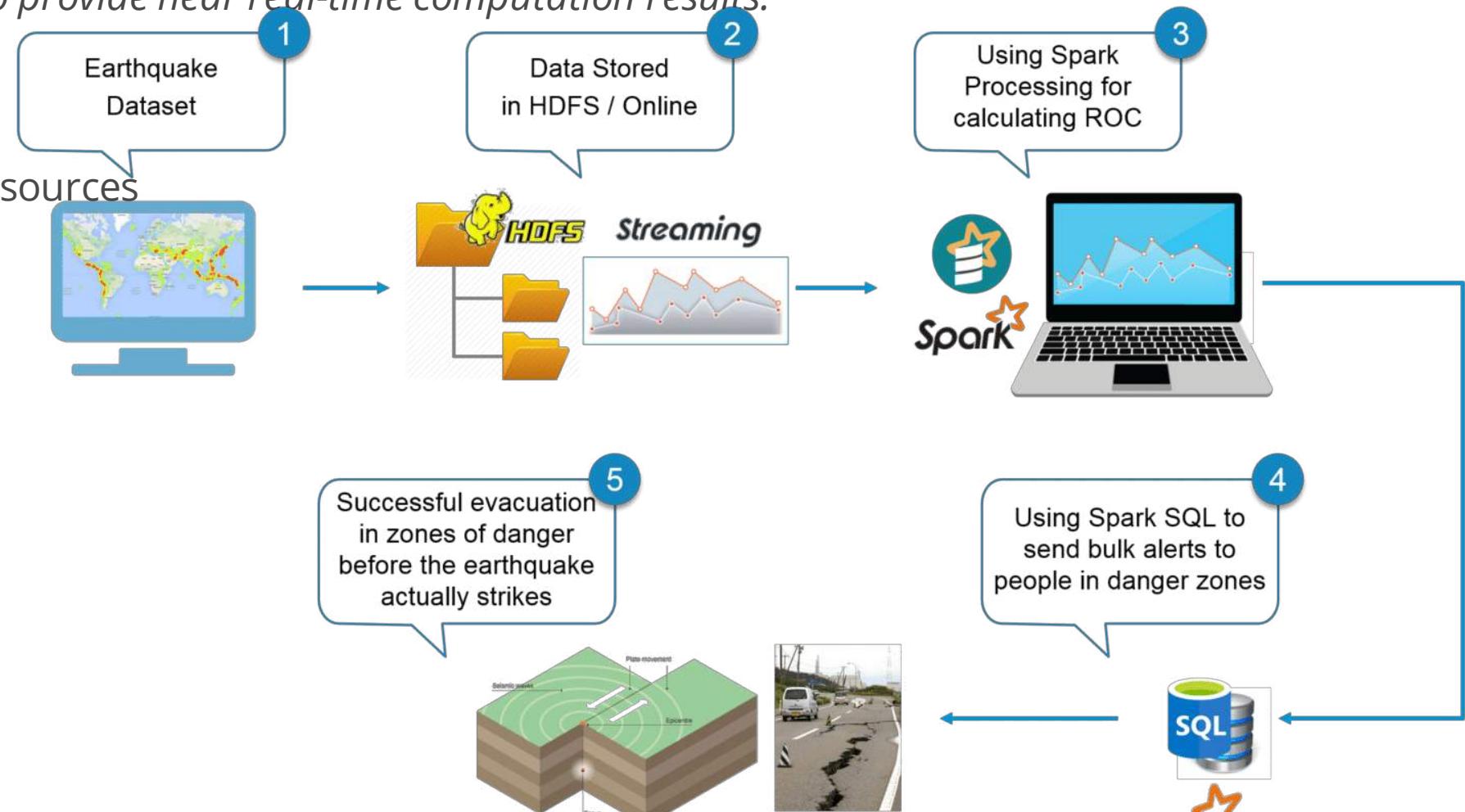
- User - Spark application code is submitted to spark
- Spark Driver internal components - convert that program (user code) in to DAG (Map – execution graph)
- DAG will be give to DAG Scheduler
- DAG Scheduler – output (stages) – Task scheduler – Task list execution – Negotiate with CM – CM checks availability of resources & RAM - CM assigns Task list to Executer -Inside executer Task will be executed- Data read- data process and write results – Results will be sent from executer to driver .
- After completion of spark application- CM deallocates resources
- Before execution of Task in Executer- Executer has to Register in Driver (Driver will have information of running executers, quantity of executers present)

# Use Case: Earthquake Detection using Spark

**Problem Statement:** To design a Real Time Earthquake Detection Model to send life saving alerts, which should improve its machine learning to provide near real-time computation results.

## Use Case - Requirements:

1. Process data in real-time
2. Handle input from multiple sources
3. Easy to use system
4. Bulk transmission of alerts



# Common Transformations & Actions

```
scala> val rdd1 = sc.parallelize(List(1,2,3,4))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at p

scala> val maprdd1 = rdd1.map(x => x+5)
maprdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at map

scala> maprdd1.collect
res0: Array[Int] = Array(6, 7, 8, 9)

scala> val filterrdd1 = rdd1.filter(x => x!=3)
filterrdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at f

scala> filterrdd1.collect
res1: Array[Int] = Array(1, 2, 4)

scala> val rdd2 = sc.parallelize(List(1,2))
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:24

scala> val maprdd2 = rdd2.map(x=>x.to(3))
maprdd2: org.apache.spark.rdd.RDD[scala.collection.immutable.Range.Inclusive] = MapPartitionsRDD[4] at map at <console>:25

scala> maprdd2.collect
res2: Array[scala.collection.immutable.Range.Inclusive] = Array(Range(1, 2, 3), Range(2, 3))

scala> val flatmaprdd2 = rdd2.flatMap(x=>x.to(3))
flatmaprdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[5]

scala> flatmaprdd2.collect
res3: Array[Int] = Array(1, 2, 3, 2, 3)

scala> val rdd3 = sc.parallelize(List(1,1,2,3,2,4))
rdd3: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[6] at p

scala> val distinctrdd3 = rdd3.distinct
distinctrdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[9] at d

scala> distinctrdd3.collect
res4: Array[Int] = Array(4, 1, 2, 3)
```

```
scala> val rdd4 = sc.parallelize(List(1,2,3,4))
rdd4: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at parallelized
scala>
scala> val rdd5 = sc.parallelize(List(4,5))
rdd5: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at parallelized
scala> val unionrdd = rdd4.union(rdd5)
unionrdd: org.apache.spark.rdd.RDD[Int] = UnionRDD[12] at union at <console>
scala> unionrdd.collect
res5: Array[Int] = Array(1, 2, 3, 4, 4, 5)
```

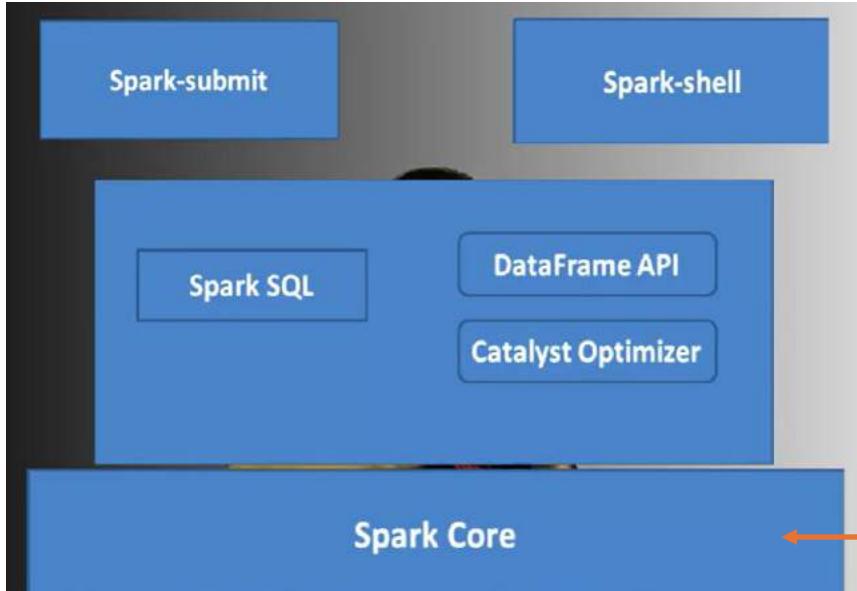
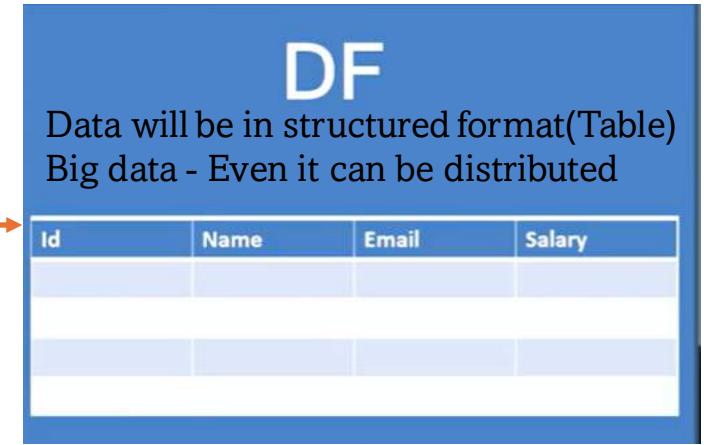
```
scala> subtractrdd.collect
res7: Array[Int] = Array(1, 2, 3)
scala> rdd4.reduce((x, y) => x + y)
res8: Int = 10
scala> rdd4.take(2)
res9: Array[Int] = Array(1, 2)
scala> rdd4.top(3)
res10: Array[Int] = Array(4, 3, 2)
```

```
scala> val intersectionrdd = rdd4.intersection(rdd5)
intersectionrdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[13] at intersection at <console>
scala> intersectionrdd.collect
res6: Array[Int] = Array(4)
scala> val subtractrdd = rdd4.subtract(rdd5)
```

# DATAFRAME (DF):

DF = Is RDD in which SQL Table is inserted

- RDD (Limitations)
  - Not Easy as SQL
  - No support of catalyst optimizer (CO – Helps to prioritize the operations = Increase performance, arranges the flow of work)



Execution of  
DF/RDD

# RDD To DF

```
hduser@ubuntu:~$ spark-shell --conf spark.sql.catalogImplementation=in-memory */
20/05/28 13:28:31 WARN Utils: Your hostname, ubuntu resolves to a loopback address:
```

```
scala> val pehlaRdd = sc.parallelize(1 to 10).map(x => (x,"df ka data"))
pehlaRdd: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[1] at map at <console>:24

scala> pehlaRdd.collect
20/05/28 13:31:11 WARN SizeEstimator: Failed to check whether UseCompressedOoops is set; assuming yes
res0: Array[(Int, String)] = Array((1,df ka data), (2,df ka data), (3,df ka data), (4,df ka data), (5,df
ka data), (6,df ka data), (7,df ka data), (8,df ka data), (9,df ka data), (10,df ka data))
```

```
scala> val pehlaDf = pehlaRdd.toDF("id","simple_string")
pehlaDf: org.apache.spark.sql.DataFrame = [id: int, simple_string: string]

scala> pehlaRdd.collect.foreach(println)
(1,df ka data)
(2,df ka data)
(3,df ka data)
(4,df ka data)
(5,df ka data)
(6,df ka data)
(7,df ka data)
(8,df ka data)
(9,df ka data)
(10,df ka data)
```

- We **didn't provide any schema**
- Spark **detected the data types by itself** – Because of RDD.
- Assume if you were suppose to read data file from Hadoop (HDFS) or from local system – define RDD – then we **need to define schema**.
- AS we cannot use **parallelize method** to create RDD always.

```
scala> pehlaDf.show
+---+-----+
| id|simple_string|
+---+-----+
1	df ka data
2	df ka data
3	df ka data
4	df ka data
5	df ka data
6	df ka data
7	df ka data
8	df ka data
9	df ka data
10	df ka data
+---+-----+
```

# Creating DF – Providing Schema

```
scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row
```

```
scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._
```

```
scala> val vidyarthiRdd = sc.parallelize(Array(Row(1,"sdp",25),(Row(2,"xyz",31))))
vidyarthiRdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = ParallelCollectionRD
e at <console>:28
```

```
scala> vidyarthiRdd
val vidyarthiRdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
```

```
scala> vidyarthiRdd.collect
20/05/28 15:39:27 WARN SizeEstimator: Failed to check whether UseCompressedOoops is set;
res0: Array[org.apache.spark.sql.Row] = Array([1,sdp,25], [2,xyz,31])
```

```
scala> val schema009 = StructType(
 | Array(
 | StructField("rollNo",IntegerType,true),
 | StructField("name",StringType,true),
 | StructField("marks",IntegerType,true)
 |)
schema009: org.apache.spark.sql.types.StructType = StructType(StructField(rollNo,IntegerType,true)
tField(name,StringType,true), StructField(marks,IntegerType,true))

scala> val vidyarthiDf = spark.createDataFrame(vidyarthiRdd,schema009);
vidyarthiDf: org.apache.spark.sql.DataFrame = [rollNo: int, name: string ... 1 more field]
```

```
scala> vidyarthiDf.printSchema
root
|-- rollNo: integer (nullable = true)
|-- name: string (nullable = true)
|-- marks: integer (nullable = true)
```

```
scala> vidyarthi
vidyarthiDf vidyarthiRdd

scala> vidyarthiDf.show
+---+---+---+
|rollNo|name|marks|
+---+---+---+
| 1| sdp| 25|
| 2| xyz| 31|
+---+---+---+
```

# Creating DF using CSV & Parquet files

\*spark\_df\_video

```
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2012
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,javed jafree,2007|
```

- Transfer the file to spark (read from HDFS / Local system)
- Create DF & Define Schema
  - Create Class
  - Create RDD
  - Create DF

```
scala> case class Movies(rank:Int,movie:String,actor:String,release_year:Int)
defined class Movies
```

Class

```
scala> val FavouriteMoviesRdd = sc.textFile("/home/hduser/Desktop/data/fav_movies")
FavouriteMoviesRdd: org.apache.spark.rdd.RDD[String] = /home/hduser/Desktop/data/fav_movies
RDD[7] at textFile at <console>:28
```

RDD (Read file from local)

```
scala> FavouriteMoviesRdd.collect
collect collectAsync
```

```
scala> FavouriteMoviesRdd.collect.foreach(println)
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2012
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,iafree,2007
```

```
scala> FavouriteMoviesRdd.collect.foreach(println)
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2012
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,javed jafree,2007
```

```
scala> val FavouriteMoviesDf = FavouriteMoviesRdd.map(x => x.split(',')).map(x => Movies(x(0).toInt,x(1),
x(2),x(3).toInt)).toDF
```

$x(0) = 1$   
 $x(1) = \text{logan}$   
 $x(2) = \text{hugh jackman}$   
 $x(3) = 2017$

→ Create DF

```
case class Movies(rank:Int,movie:String,
actor:String,release_year:Int)
```

```
scala> val FavouriteMoviesDf = FavouriteMoviesRdd.map(x => x.split(',')).
map(x => Movies(x(0).toInt,x(1),
x(2),x(3).toInt)).toDF
FavouriteMoviesDf: org.apache.spark.sql.DataFrame = [rank: int, movie: string ... 2 more fields]

scala> FavouriteMovies
FavouriteMoviesDf FavouriteMoviesRdd

scala> FavouriteMoviesDf.printSchema
root
 |-- rank: integer (nullable = false)
 |-- movie: string (nullable = true)
 |-- actor: string (nullable = true)
 |-- release_year: integer (nullable = false)
```

er@ubunuc:~

```
scala> FavouriteMovies
FavouriteMoviesDf FavouriteMoviesRdd
```

```
scala> FavouriteMoviesDf.printSchema
```

```
root
|-- rank: integer (nullable = false)
|-- movie: string (nullable = true)
|-- actor: string (nullable = true)
|-- release_year: integer (nullable = false)
```

```
scala> FavouriteMovies
FavouriteMoviesDf FavouriteMoviesRdd
```

```
scala> FavouriteMoviesDf.show
```

| rank | movie                | actor              | release_year |
|------|----------------------|--------------------|--------------|
| 1    | logan                | hugh jackman       | 2017         |
| 2    | The Dark Knight      | Heath ledger       | 2008         |
| 3    | Wolverine            | hugh jackman       | 2013         |
| 4    | Deadpool             | ryan reynolds      | 2016         |
| 5    | Rocky                | Sylvester Stallone | 1976         |
| 6    | go goa gone          | saif               | 2013         |
| 7    | Joker                | joaquin phoenix    | 2019         |
| 8    | Avengers             | rdj                | 2012         |
| 9    | Sherlock Holmes      | rdj                | 2009         |
| 10   | pirates of the ca... | johnny depp        | 2003         |
| 11   | The Expendables      | sylvester stallone | 2010         |
| 12   | dhamaal              | javed jafree       | 2007         |

# Direct Load CSV file – No need to define Schema- No class creation

```
*spark_df_video
rank, movie, actor, release_year
1, logan, hugh jackman, 2017
2, The Dark Knight, Heath ledger, 2008
3, Wolverine, hugh jackman, 2013
4, Deadpool, ryan reynolds, 2016
5, Rocky, Sylvester Stallone, 1976
6, go goa gone, saif, 2013
7, Joker, joaquin phoenix, 2019
8, Avengers, rdj, 2008
9, sherlock Holmes, rdj, 2009
10, pirates of the caribbean, johnny depp, 2003
11, The Expendables, sylvester stallone, 2010
12, dhamaal, javed jafree, 2007
```

Define Column name in File itself  
Detect type (schema by spark)

```
scala> val moviesDf1 = spark.read.option("header","true").option("inferSchema","true").csv("/home/hduser/Desktop/data/fav_movies_with_header")
moviesDf1: org.apache.spark.sql.DataFrame = [rank: int, movie: string ... 2 more fields]

scala> moviesDf1.printSchema
root
|-- rank: integer (nullable = true)
|-- movie: string (nullable = true)
|-- actor: string (nullable = true)
|-- release_year: integer (nullable = true)

scala> moviesDf1.show
+---+-----+-----+-----+
|rank| movie| actor|release_year|
+---+-----+-----+-----+
1	logan	hugh jackman	2017
2	The Dark Knight	Heath ledger	2008
3	Wolverine	hugh jackman	2013
4	Deadpool	ryan reynolds	2016
5	Rocky	Sylvester Stallone	1976
6	go goa gone	saif	2013
7	Joker	joaquin phoenix	2019
8	Avengers	rdj	2008
9	sherlock Holmes	rdj	2009
10	pirates of the ca...	johnny depp	2003
11	The Expendables	sylvester stallone	2010
12	dhamaal	javed jafree	2007
+---+-----+-----+-----+
```

# Parquet File- No column Names, Schema

- Parquet File – Inside it will have information of “Column names, data types, Schema”- No need to define them.
- But file need to be with parquet extension (read from local/hdfs system)

```
scala> val moviesDf2 = spark.read.load("/home/hduser/Desktop/data/movies/movies.parquet")
moviesDf2: org.apache.spark.sql.DataFrame = [rank: int, movie: string ... 2 more fields]
```

```
scala> moviesDf2.show
+---+-----+-----+-----+
|rank| movie| actor|release_year|
+---+-----+-----+-----+
1	logan	hugh jackman	2017
2	The Dark Knight	Heath ledger	2008
3	Wolverine	hugh jackman	2013
4	Deadpool	ryan reynolds	2016
5	Rocky	Sylvester Stallone	1976
6	go goa gone	saif	2013
7	Joker	joaquin phoenix	2019
8	Avengers	rdj	2008
9	sherlock Holmes	rdj	2009
10	pirates of the ca...	johnny depp	2003
11	The Expendables	sylvester stallone	2010
12	dhamaal	javed jafree	2007
+---+-----+-----+-----+
```

A dark green background featuring a repeating pattern of tropical leaves, including palm fronds and smaller greenery.

*Thank you*

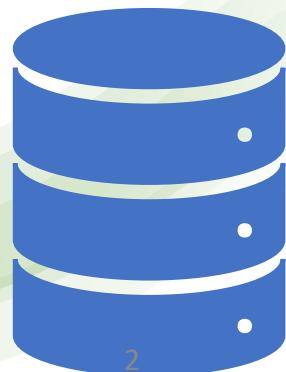
BY: Dr. RASHMI L MALGHAN



Apache Hive

# HIVE

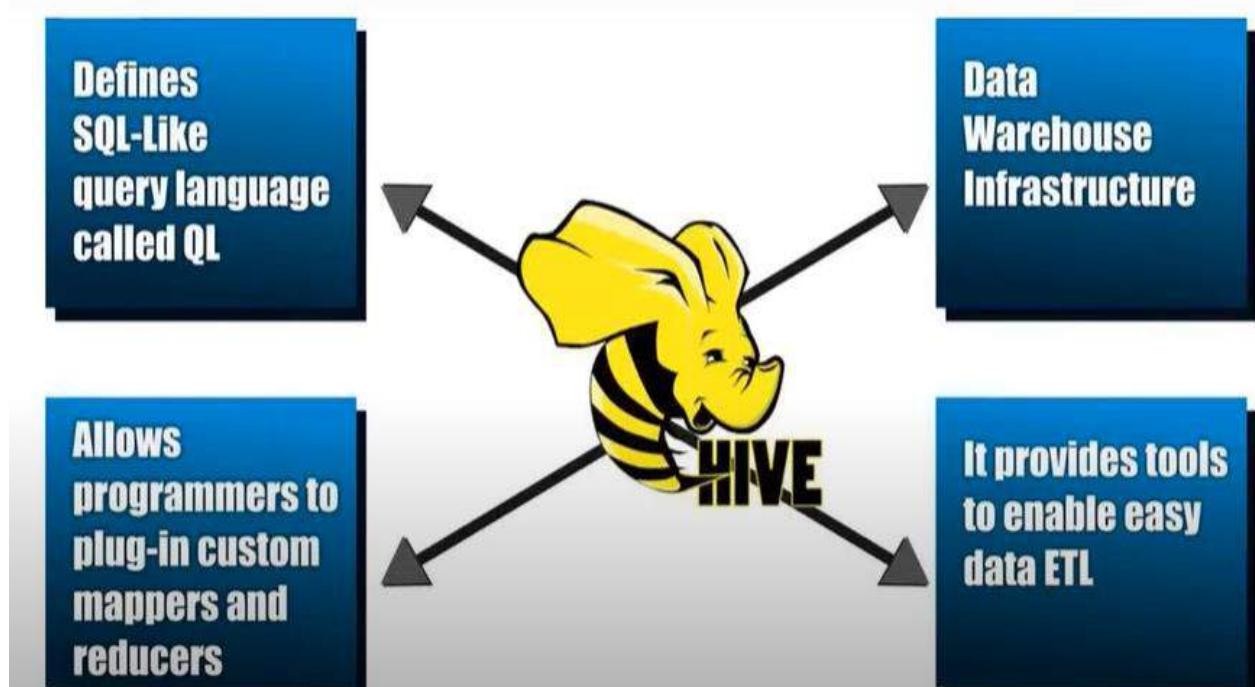
- Apache Hive is an open-source software utility **built on top of Hadoop** to provide easy **data storage and analysis in style of SQL**.
- Hive - Basically, a tool which we call a **data warehousing tool** is Hive. However, Hive gives **SQL queries** to perform an analysis and also an abstraction. Although, Hive it is not a database it gives you **logical abstraction over the databases and the tables**
- **It is a distributed, fault-tolerant data warehouse system** that enables analytics at a massive scale and facilitates reading, writing, and managing petabytes of data residing in distributed storage using SQL.
- Hive is built on top of Apache Hadoop and supports storage on S3, adls, gs etc though HDFS.
- It is not built for Online Transactional Processing (OLTP) workloads. **Limited concurrency**
- It is designed to **enhance scalability, extensibility, performance, fault-tolerance** and loose-coupling with its input formats.
- Hive uses a language called **HiveQL**, which is similar to SQL, to allow users to express data queries, transformations, and analyses in a familiar syntax.
- HiveQL statements are compiled into MapReduce jobs, which are then executed on the Hadoop cluster to process the data.



# What is HIVE?

- Data Warehousing package built on top of Hadoop
- Used for data analysis
- Targeted towards users comfortable with SQL
- It is similar to SQL and called HiveQL
- For managing and querying structured data
- Abstracts complexity of Hadoop
- No need learn java and Hadoop APIs
- Developed by Facebook and contributed to community
- Facebook analyzed several Terabytes of data everyday using Hive

# Features of HIVE

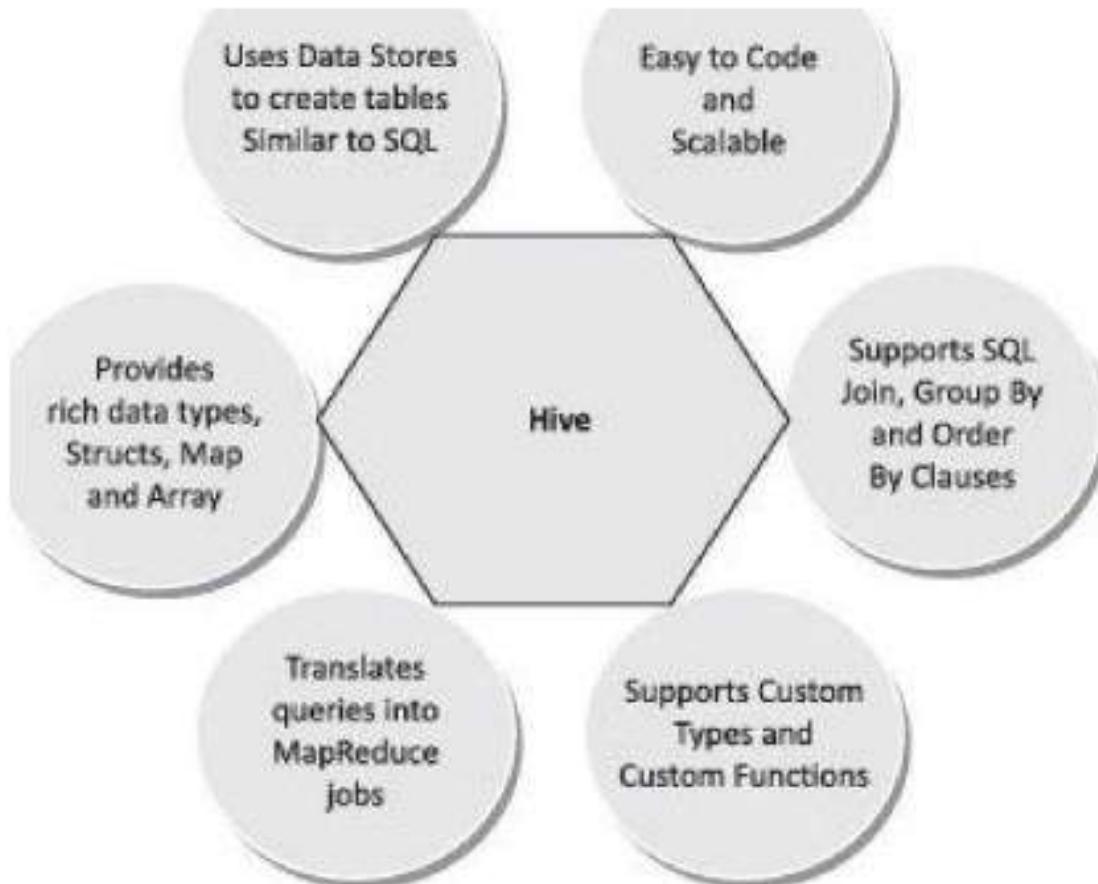


- Hive supports any client application written in Java, PHP, Python, C++ or Ruby by exposing its **Thrift server**. (You can use these client - side languages embedded with SQL for accessing a database such as DB2, etc.).
- As the **metadata information of Hive is stored in an RDBMS**, it significantly **reduces the time** to perform **semantic checks** during query execution.
- Useful for people who aren't from a programming background as it **eliminates the need to write complex MapReduce program**.
- **Extensible** and **scalable** to cope up with the growing volume and variety of data, without affecting performance of the system.
- It is as an **efficient ETL** (Extract, Transform, Load) tool.

## Features of HIVE

- It provides indexes, including bitmap indexes to accelerate the queries. Index type containing compaction and bitmap index as of 0.10.
- **Metadata storage in a RDBMS, reduces the time to function semantic checks** during query execution.
- **Built in user-defined functions (UDFs) to manipulation of strings, dates, and other data-mining tools.** Hive is reinforced to extend the UDF set to deal with the use-cases not reinforced by predefined functions.
- **DEFLATE, BWT, snappy**, etc are the **algorithms to operation on compressed data** which is stored in Hadoop Ecosystem.
- It is built for Online Analytical Processing (OLAP).
- It delivers various types of querying language which are frequently known as Hive Query Language (HQL or HiveQL).

# Hive Features...



- **Traditional relational databases** are designed for interactive queries on small to medium datasets and **do not process huge datasets** well.
- Hive instead **uses batch processing** so that it works quickly across a very large distributed database.
- It **queries data stored** in a distributed storage solution, like the **Hadoop Distributed File System (HDFS) or Amazon S3**
- Hive stores its database and table metadata in a **metastore**
- Hive includes **HCatalog**, which is a **table and storage management layer** that reads data from the Hive metastore to facilitate seamless integration between Hive, Apache Pig, and MapReduce.
- **By using the metastore, HCatalog allows Pig and MapReduce to use the same data structures as Hive**, so that the metadata doesn't have to be redefined for each engine.

# Hive Characteristics:

- ❖ Capability to “[Translate Queries in to MapReduce Jobs](#)”
- ❖ This makes Hive “[scalable](#)”, able to handle [Datawarehouse applications](#) and therefore suitable for the [analysis](#) of [static data](#) of an extremely large size, where the “[fast – response time](#)” is not a criterion.
- ❖ Supports [web interfaces](#) , [Application API](#) as well as [web-browser clients](#), can access the [Hive DB Server](#).
- ❖ Provides an SQL “[Dialect](#)” ([Hive Query Language](#), abbreviated [HIVEQL](#) or [HQL](#)).
- ❖ [Results of HiveQL query](#) and the [data load in the tables](#) which store at [Hadoop cluster](#) at [HDFS](#).
- ❖ Hive as data warehouse is built to manage and query only structured data which is residing under tables.
- ❖ **Tables and databases get** created first; then data gets loaded into the proper tables
- ❖ Hive supports four file formats: **ORC**, **SEQUENCEFILE**, **RCFILE** (Record Columnar File), and **TEXTFILE**
- ❖ difference between the Hive Query Language (HQL) and SQL is that **Hive executes queries on Hadoop's infrastructure instead of on a traditional database**.
- ❖ Hive supports Data Definition Language (DDL), Data Manipulation Language (DML), and User Defined Functions (UDF).

## Limitations of Hive

- Hive supports Online Analytical Processing (OLAP), but not Online Transaction Processing (OLTP).
- It doesn't support subqueries.
- It has a high latency.
- Hive tables don't support delete or update operations.
- Hive is not capable of handling real-time data.

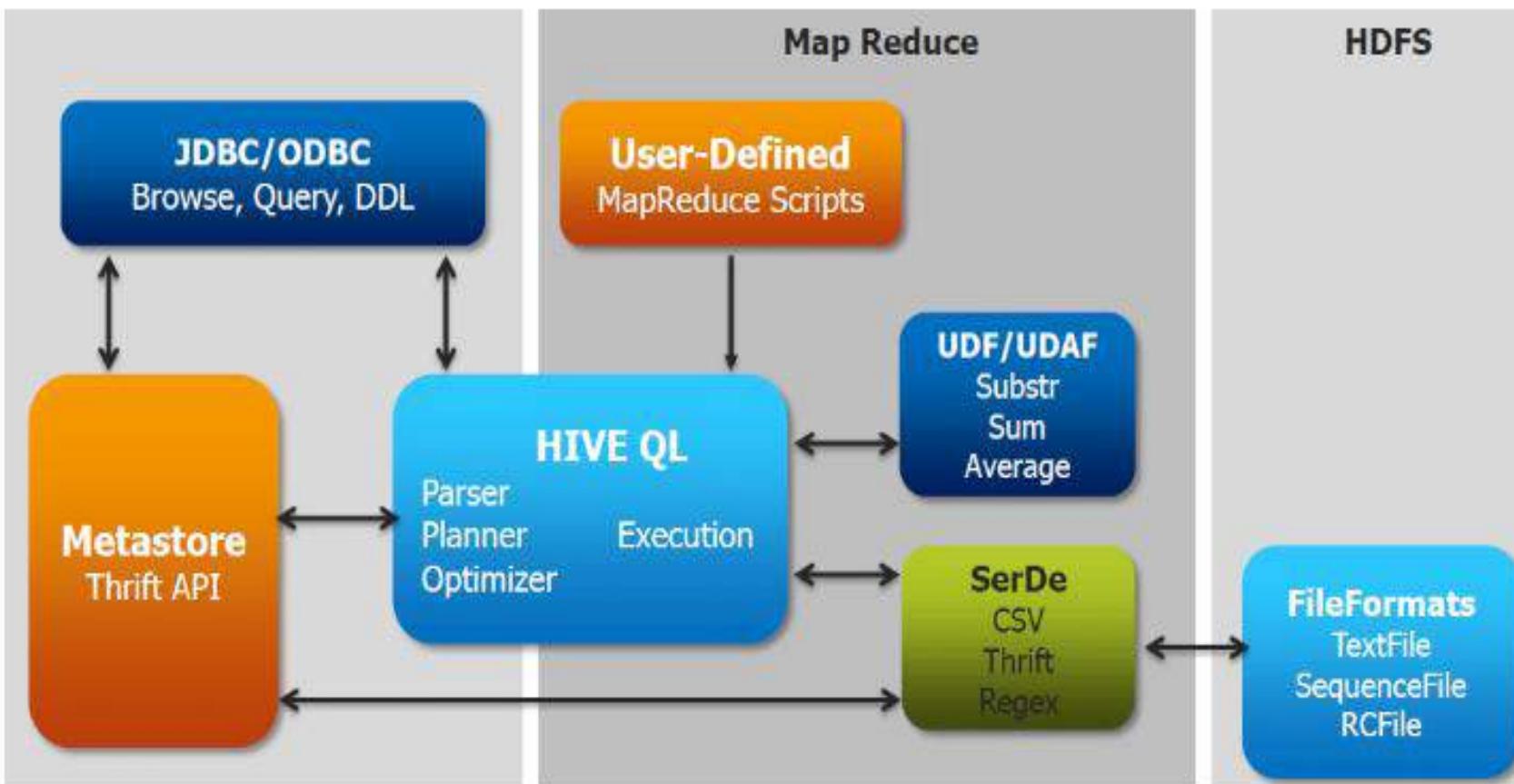
## Differences between Hive and Pig

| Hive                                     | Pig                                      |
|------------------------------------------|------------------------------------------|
| Hive is commonly used by Data Analysts.  | Pig is commonly used by programmers.     |
| It follows SQL-like queries.             | It follows the data-flow language.       |
| It can handle structured data.           | It can handle semi-structured data.      |
| It works on server-side of HDFS cluster. | It works on client-side of HDFS cluster. |
| Hive is slower than Pig.                 | Pig is comparatively faster than Hive.   |

## Hive Vs. SQL

| Feature            | Apache Hive                            | SQL                                   |
|--------------------|----------------------------------------|---------------------------------------|
| Purpose            | Batch and Interactive Query Processing | Relational Database Management System |
| Data Analysis      | Complex Data Processing                | Detailed Data Querying                |
| Architecture       | Data Warehousing Project               | RDBMS-Based Programming Language      |
| Data Types         | 9 Types Supported                      | 5 Types Supported                     |
| Multitable Inserts | Supported                              | Not Supported                         |
| MapReduce          | Supports MapReduce                     | No MapReduce Concept                  |
| OLTP               | Not Supported                          | Supports OLTP                         |
| Schema Support     | Supported                              | Used for Data Storage                 |
| Views              | Read-Only Format                       | Updateable Views                      |
| Data Size          | Can handle petabytes of data           | Can only handle terabytes of data     |

# Apache Hive Components:



Hive consists of the following major components:

**Metastore** – To store the metadata.

**JDBC/ODBC** – Query Compiler and Execution Engine to convert SQL queries to a sequence of MapReduce.

**SerDe and ObjectInspectors** – For data formats and types.

**UDF/UDAF** – For User Defined Functions.

**Clients** – Similar to MySQL command line and a web UI.

## Major Components

| Unit Name                | Operation                                                                                                                                                                                                                                                |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| User Interface           | Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).                                  |
| Meta Store               | Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.                                                                                                   |
| HiveQL<br>Process Engine | HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it. |
| Execution<br>Engine      | The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce.                                          |
| HDFS or<br>HBASE         | Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.                                                                                                                                                  |

# Hive Architecture:

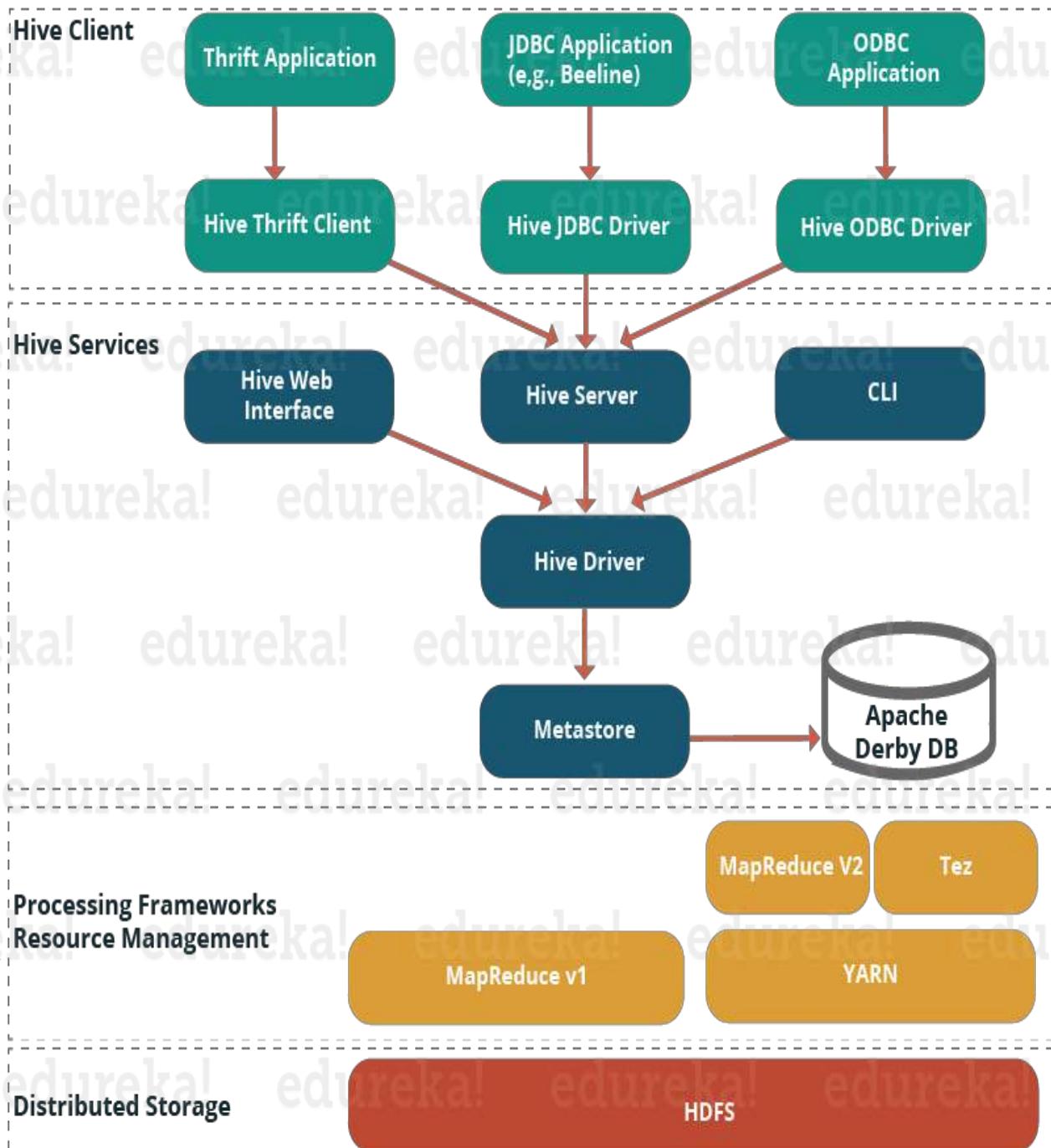
Hive Architecture can be categorized into the following components:

- **Hive Clients:** Hive supports application written in many languages like Java, C++, Python etc. using JDBC, Thrift and ODBC drivers. Hence one can always write hive client application written in a language of their choice.

- **Hive Services:** Apache Hive provides various services like CLI, Web Interface etc. to perform queries.

- **Processing framework and Resource Management:** Internally, Hive uses Hadoop MapReduce framework as execution engine to execute the queries.

- **Distributed Storage:** As Hive is installed on top of Hadoop, it uses the underlying HDFS for the distributed storage.



## Hive Client

- **Thrift Server** - It is a cross-language service provider platform that serves the request from all those programming languages that supports Thrift.
- **JDBC Driver** - It is used to establish a connection between hive and Java applications. The JDBC Driver is present in the class org.apache.hadoop.hive.jdbc.HiveDriver.
- **ODBC Driver** - It allows the applications that support the ODBC protocol to connect to Hive.

## Hive Services

- **Hive CLI** - The Hive CLI (Command Line Interface) is a shell where we can execute Hive queries and commands.
- **Hive Web User Interface** - The Hive Web UI is just an alternative of Hive CLI. It provides a web-based GUI for executing Hive queries and commands.
- **Hive MetaStore** - It is a central repository that stores all the structure information of various tables and partitions in the warehouse. It also includes metadata of column and its type information, the serializers and deserializers which is used to read and write data and the corresponding HDFS files where the data is stored.
- **Hive Server** - It is referred to as Apache Thrift Server. It **accepts the request from different clients** and provides it to **Hive Driver**.
- **Hive Driver** - It receives queries from different sources like **web UI, CLI, Thrift, and JDBC/ODBC driver**. It transfers the queries to the **compiler**.

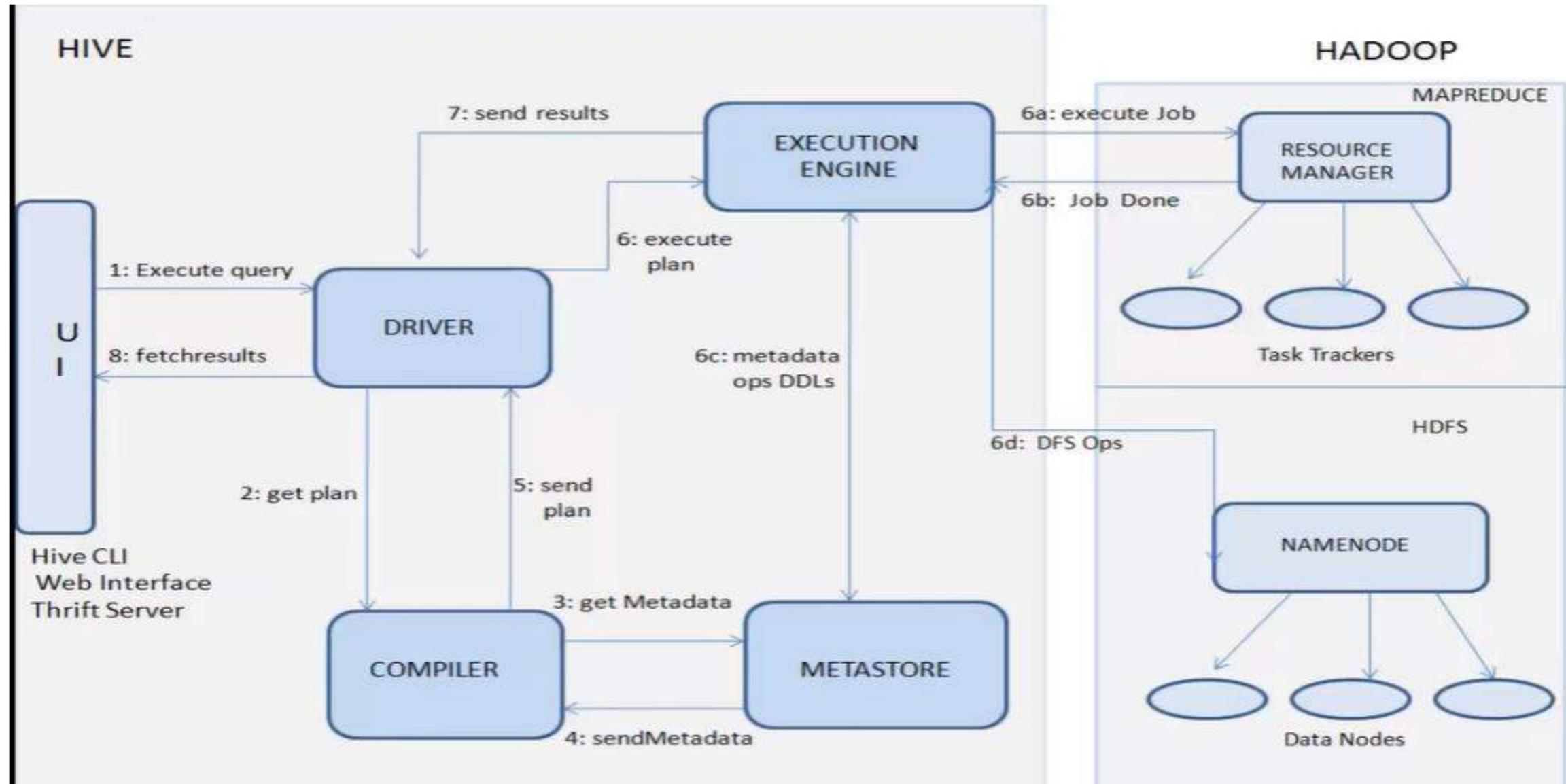
## Hive Services

- **Hive Compiler** - The purpose of the compiler is **to parse the query and perform semantic analysis** on the different query blocks and expressions. It **converts HiveQL statements into MapReduce jobs**.
- **Hive Execution Engine** - Optimizer generates the **logical plan in the form of DAG** of map-reduce tasks and HDFS tasks. In the end, the execution engine executes the incoming tasks in the order of their dependencies.

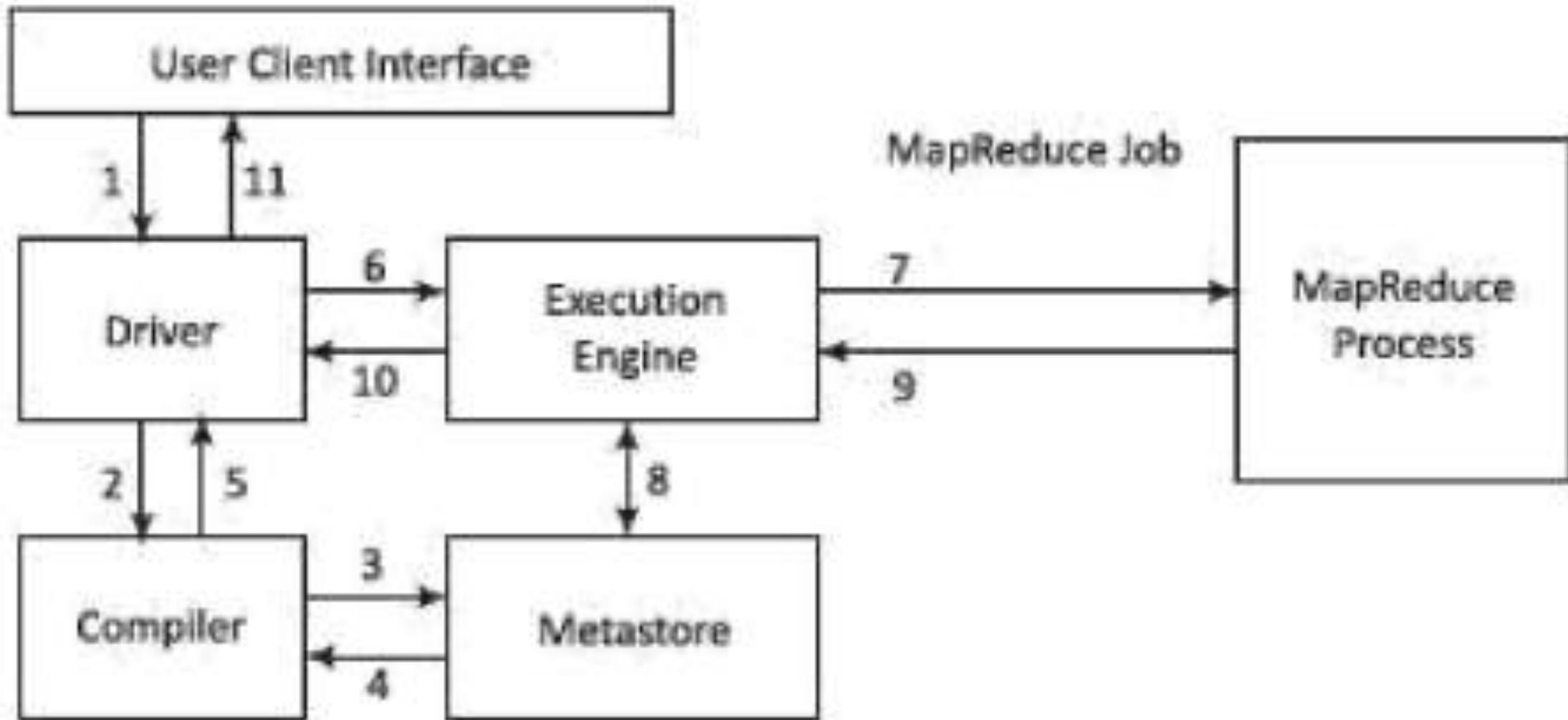
### Hive makes use of the following:

- HDFS for storage
- MapReduce for execution
- Stores metadata in RDBMS

# Program Execution



# Dataflow Sequence & Workflow Steps:



| No. | OPERATION                                                                                                                                                                                                                                                 |
|-----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1   | <b>Execute Query:</b> Hive interface (CLI or Web Interface) sends a query to Database Driver to execute the query.                                                                                                                                        |
| 2   | <b>Get Plan:</b> Driver sends the query to query compiler that parses the query to check the syntax and query plan or the requirement of the query.                                                                                                       |
| 3   | <b>Get Metadata:</b> Compiler sends metadata request to Metastore (of any database, such as MySQL).                                                                                                                                                       |
| 4   | <b>Send Metadata:</b> Metastore sends metadata as a response to compiler.                                                                                                                                                                                 |
| 5   | <b>Send Plan:</b> Compiler checks the requirement and resends the plan to driver. The parsing and compiling of the query is complete at this place.                                                                                                       |
| 6   | <b>Execute Plan:</b> Driver sends the execute plan to execution engine.                                                                                                                                                                                   |
| 7   | <b>Execute Job:</b> Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Then, the query executes the job. |
| 8   | <b>Metadata Operations:</b> Meanwhile the execution engine can execute the metadata operations with Metastore.                                                                                                                                            |
| 9   | <b>Fetch Result:</b> Execution engine receives the results from Data nodes.                                                                                                                                                                               |
| 10  | <b>Send Results:</b> Execution engine sends the result to Driver.                                                                                                                                                                                         |
| 11  | <b>Send Results:</b> Driver sends the results to Hive Interfaces.                                                                                                                                                                                         |

# Dataflow Sequence & Workflow Steps:

# Hive Components:



## Shell:-

Provides User interface provide an interface between user and hive. It enables user to submit queries and other operations to the system.

## Metastore –

All the structured data or information of the different tables and partition in the warehouse containing attributes and attributes level information are stored in the metastore.

## Execution Engine –

Execution of the execution plan made by the compiler is performed in the execution engine.

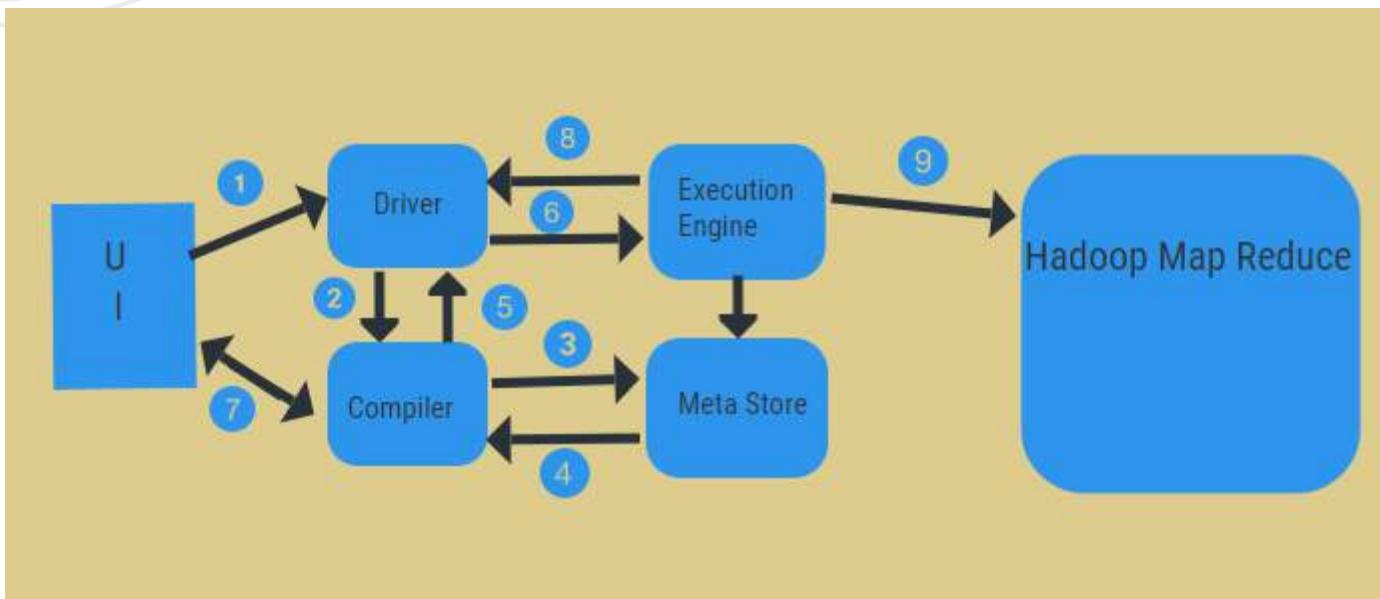
## Compiler –

Queries are parses, semantic analysis on the different query blocks and query expression is done by the compiler.

## Drivers:-

Queries of the user after the interface are received by the driver within the Hive. Concept of session handles is implemented by driver

# Hive Components:



**The steps include:**

- 1.execute the Query from UI
- 2.get a plan from the driver tasks DAG stages
- 3.get metadata request from the meta store
- 4.send metadata from the compiler
- 5.sending the plan back to the driver
- 6.Execute plan in the execution engine
- 7.fetching results for the appropriate user query
- 8.sending results bi-directionally
- 9.execution engine processing in HDFS with the map-reduce and fetch results from the data nodes created by the job tracker. it acts as a connector between Hive and Hadoop.

# Hive Metastore:

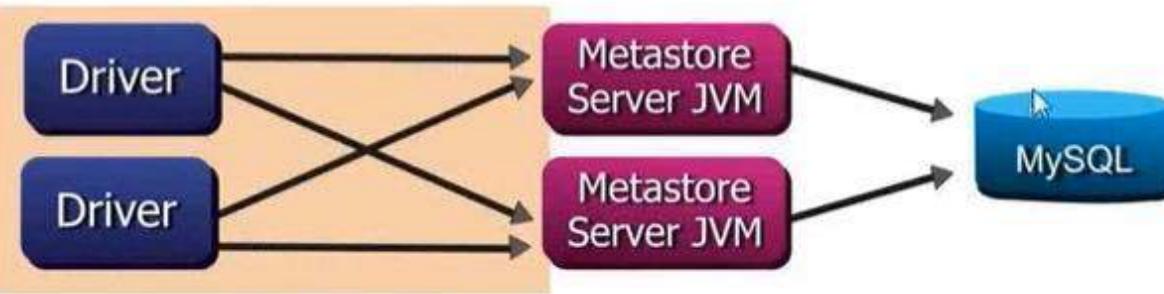
**Embedded Metastore**



**Local Metastore**



**Remote Metastore**



## Embedded Metastore:

It offers an embedded Derby database instance backed by the local disk for the Metastore, by **default**. It is what we call embedded Metastore configuration.

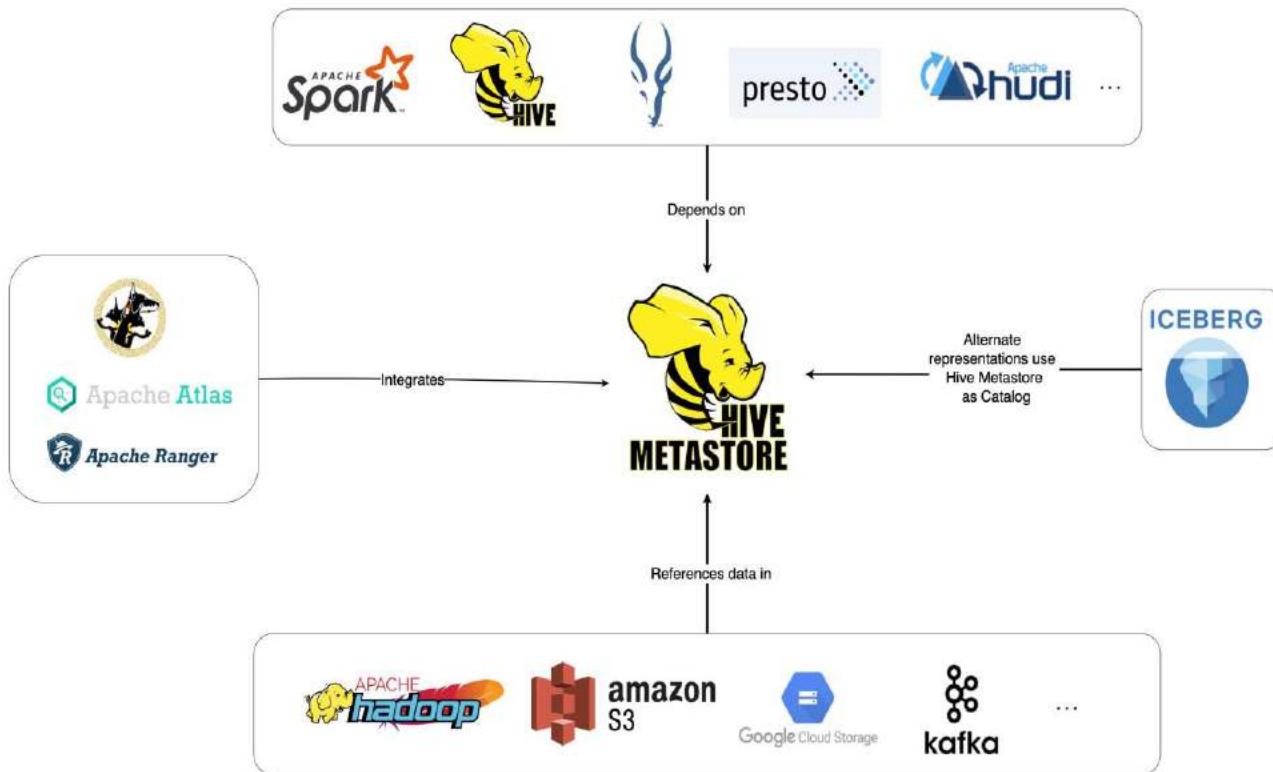
## Local Metastore:

**It is the Metastore service runs in the same JVM** in which the Hive service is running and connects to a database running in a separate JVM. Either on the same machine or on a remote machine.

## Remote Metastore:

In this configuration, **the Metastore service runs on its own separate JVM** and not in the Hive service JVM.

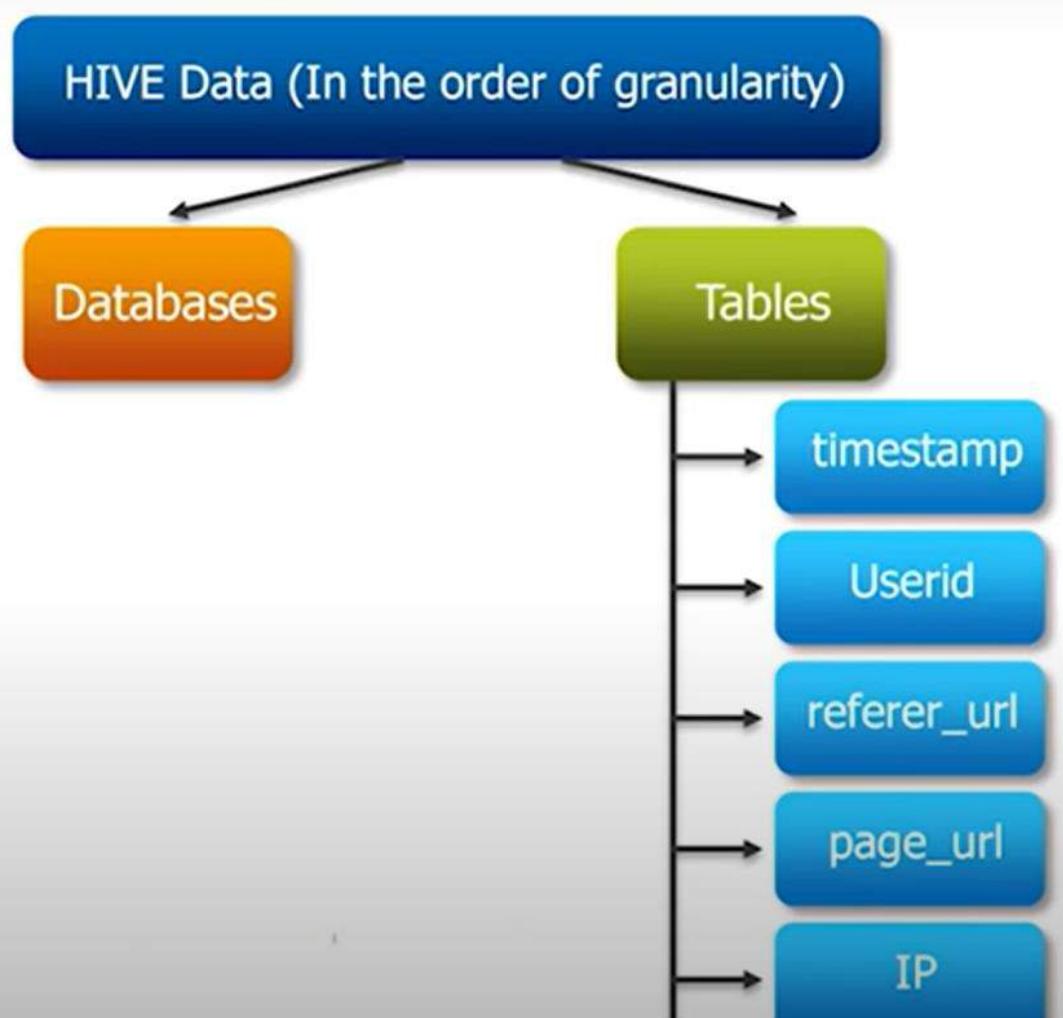
# Hive Metastore(HMS)



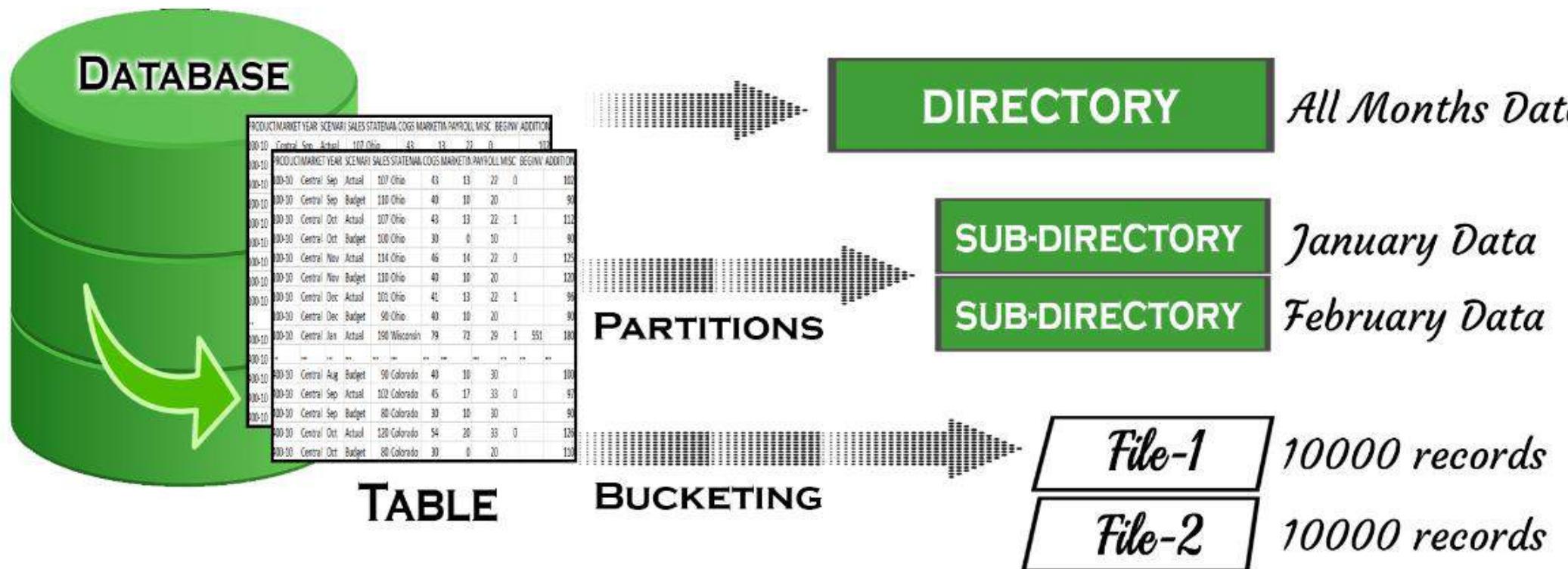
- The Hive Metastore (HMS) is a **central repository** of metadata for Hive tables and partitions in a relational database, and provides clients (including Hive, Impala and Spark) access to this information using the metastore service API.
- It has become a **building block for data lakes** that utilize the diverse world of open-source software, such as Apache Spark and Presto. In fact, a whole ecosystem of tools, open-source and otherwise, are built around the Hive Metastore.

# Hive Data Models

- ✓ Databases
  - ✓ Namespaces
- ✓ Tables
  - ✓ Schemas in namespaces
- ✓ Partitions
  - ✓ How data is stored in HDFS
  - ✓ Grouping data bases on some column
  - ✓ Can have one or more columns
- ✓ Buckets or Clusters
  - ✓ Partitions divided further into buckets bases on some other column
  - ✓ Use for data sampling



# Hive Data Models



# Create Database :

- ✓ **Create Database**
  - ✓ `Create database retail;`
- ✓ **Use Database**
  - ✓ `Use retail;`
- ✓ **Create table for storing transactional records**
  - ✓ `Create table txnrecords(txnno INT, txndate STRING, custno INT, amount DOUBLE, category STRING, product STRING, city STRING, state String, Spendby String )`
  - ✓ `Row format delimited`
  - ✓ `Fields terminated by ',' stored as textfile`

CREATE DATABASE |SCHEMA [IF NOT EXISTS] <database name>

hive> CREATE DATABASE [IF NOT EXISTS] userdb;

hive> CREATE SCHEMA userdb;

## Tables:

- Apache **Hive tables** are the **same as the tables** present in a **Relational Database**.
- We can perform **filter, project, join and union operations** on tables.
- Hive stores the **metadata in a relational database** and not in HDFS.
- Hive has **two types of tables** which are as follows:

### 1. Managed Table:

*Command:*

```
CREATE TABLE <table_name> (column1 data_type, column2 data_type);
LOAD DATA INPATH <HDFS_file_location> INTO table managed_table;
```

### 2. External Table:

*Command:*

```
CREATE EXTERNAL TABLE <table_name> (column1 data_type, column2 data_type) LOCATION '<table_hive_location>';
LOAD DATA INPATH '<HDFS_file_location>' INTO TABLE <table_name>;
```

For *external table*, Hive is not responsible for managing the data.

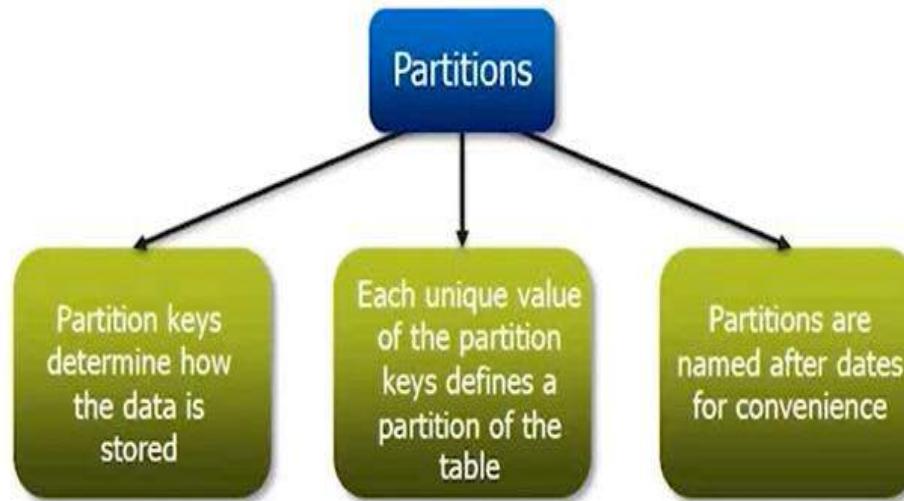
# External Table

- ✓ **Create the table in another hdfs location** and not in warehouse directory
- ✓ **Not managed by hive**
  - ✓ `CREATE EXTERNAL TABLE external_Table (dummy STRING)`
  - ✓ `LOCATION '/user/notroot/external_table';`
- ✓ **Hive does not delete the table (or hdfs files) even when the tables are dropped.**  
It leaves the table untouched and only metadata about the tables are deleted.

Need to specify the hdfs location

# Partition:

**Partition** means dividing a table into coarse grained parts based on the value of a partition column such as a date. This make it faster to do queries on slices of the data.



- student\_details containing the student information of students like student\_id, name, department, year, etc. Now, if I perform partitioning based on department column, the information of all the students belonging to a particular department will be stored together in that very partition.

- Partition: is used to group similar data types **together based on column or partition key**.
- Hive organizes tables into partitions.
- In other words, we can say that **partition is used to create a sub-directory in the table directory**.
- Each table can **have one or more partition keys** to identify a particular partition.
- Partitioning is used in **Hive to reduce the query latency**. Instead of scanning the entire tables, it scans only the relevant partitions and corresponding datasets.

# Dynamic Partitioning (Columns - values known – Execution Time)

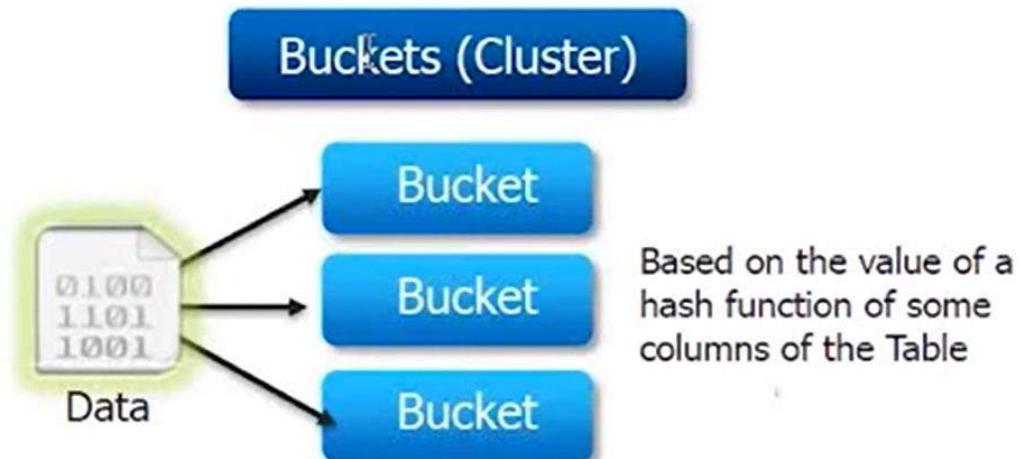
- A partitioning is called dynamic partitioning **while loading the data into the Hive table**. In other words, we can say that dynamic partitioning values for **partition columns** in the runtime.
- **Dynamic partitioning is used in the following cases:**
- While we Load data from an existing **non-partitioned table**, it is used to improve the **sampling**. Thus it decreases the **query latency**.
- While we do not know all **the values of the partitions beforehand**, so, finding these partition **values manually** from a huge dataset is a tedious task.

# Hive Data Model :

| Name       | Description                                                                                                                                 |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| Database   | Namespace for tables                                                                                                                        |
| Tables     | Similar to tables in RDBMS<br>Support filter, projection, join and union operations<br>The table data stores in a directory in HDFS         |
| Partitions | Table can have one or more partition keys that tell how the data stores                                                                     |
| Buckets    | Data in each partition further divides into buckets based on hash of a column in the table.<br>Stored as a file in the partition directory. |

# Buckets:

- ✓ Buckets give extra structure to the data that may be used for more efficient queries.
  - ✓ A join of two tables that are bucketed on the same columns – including the join column can be implemented as a Map Side Join.
  - ✓ Bucketing by user ID means we can quickly evaluate a user based query by running it on a randomized sample of the total set of users.



# Bucketing & Reasoning:

- Bucketing is the concept of **breaking data down into ranges**, which are known as buckets.
- Bucketing is mainly a **data organizing technique**.
- It is similar to partitioning in Hive with an **added functionality** that it divides large datasets into more **manageable parts known as buckets**.
- The partitioning into buckets can give **extra structure to the data** to use for more **efficient queries**.
- The range for a bucket is determined by **the hash value of one or more columns** in the dataset.
- **Reasons:**

**There are two main reasons for performing bucketing to a partition:**

- We perform bucketing to a partition because a **map side join** requires the data belonging to a unique join key to be **present in the same partition**.
- Bucketing facilitates us to decrease the **query time**, and it also makes the **sampling process** more efficient.

# Optimization: Indexing

- Indexing in Hive is a Hive query optimization technique.
- It is mainly used to speed up the access of a column or set of columns in a Hive database.
- With the use of the index, the Hive database system does not need to read all rows in the table, especially that one has selected.

# Hive Data Model:

## Partitions:

*Command:*

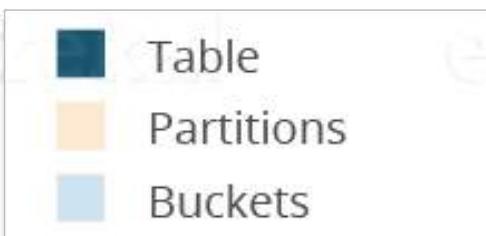
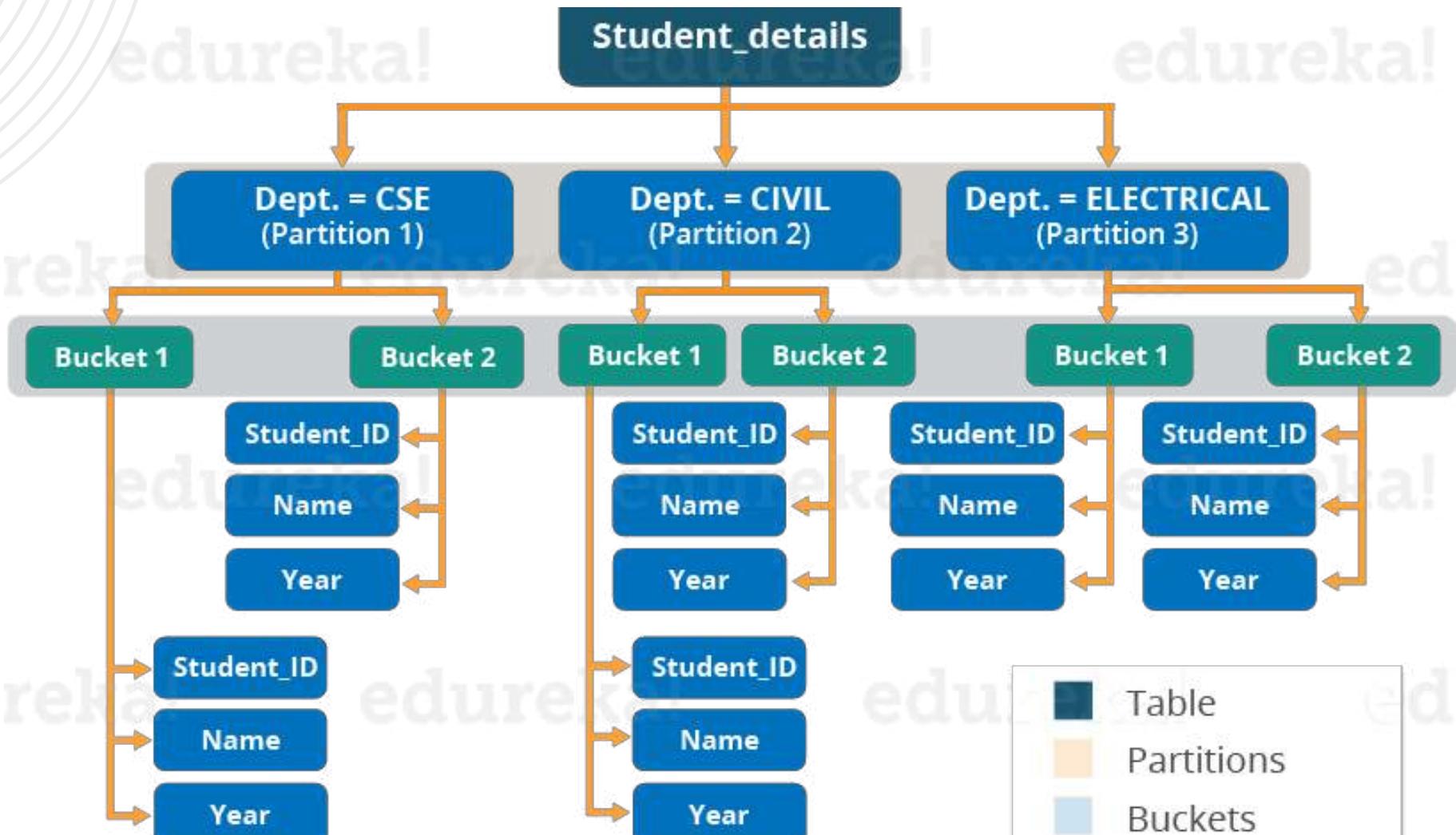
```
CREATE TABLE table_name (column1 data_type, column2 data_type) PARTITIONED BY (partition1 data_type, partition2 data_type,...);
```

Hive organizes tables into partitions for grouping similar type of data together based on a column or partition key. Each Table can have one or more partition keys to identify a particular partition. This allows us to have a faster query on slices of the data.

## Buckets:

*Commands:*

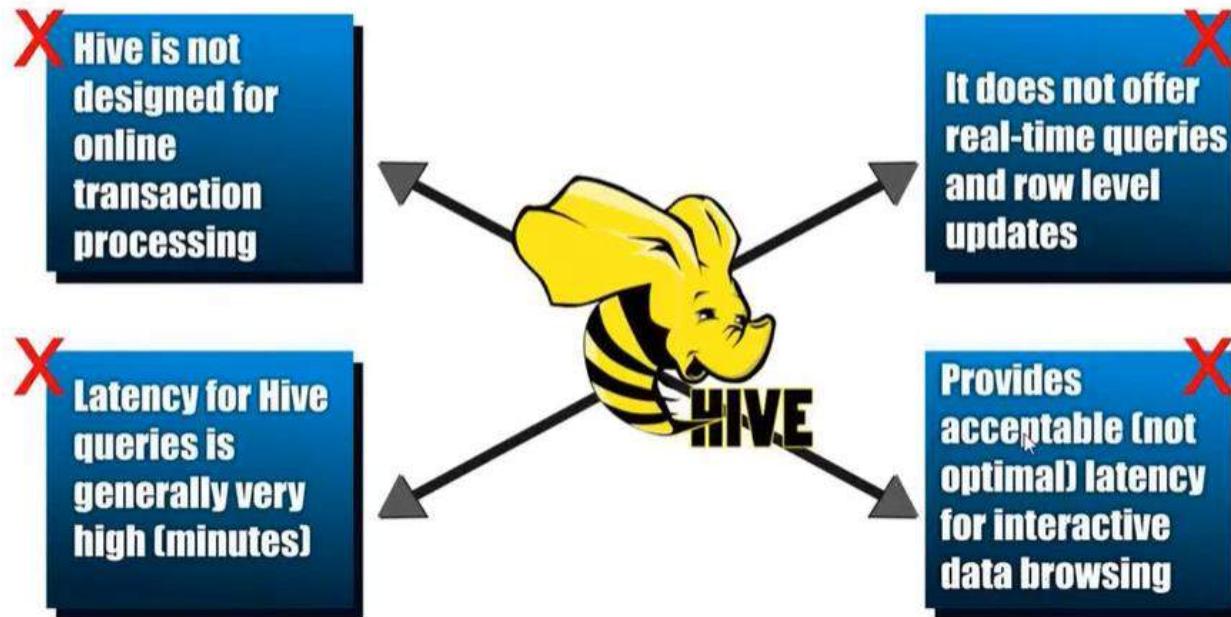
```
CREATE TABLE table_name PARTITIONED BY (partition1 data_type, partition2 data_type,...) CLUSTERED BY (column_name1, column_name2, ...) SORTED BY (column_name [ASC | DESC], ...) INTO num_buckets BUCKETS;
```



# MODES OF HIVE

- **Local Mode**
- **MapReduce Mode**
  - In Hive, the Map reduce mode is used in the following conditions:
  - To perform on a large amount of data sets and query going to execute in a parallel way .
  - When Hadoop has multiple data nodes and is distributed across different nodes, we should use this mode.
  - To achieve better performance.

# Limitation of Hive:



- Hive Does not provide **update, alter and deletion of records** in the database.
- Not developed for “**Unstructured Data**”
- Not designed for “**Real Time Queries**”.
- Performs the **partition** always from the **last column**.
- OLTP:** No. Because Hive does not provide **insert and update at the row level**, it is not suitable for the OLTP system.

Table 1: Primitive, String, Date/time datatypes

# Hive Data Types :

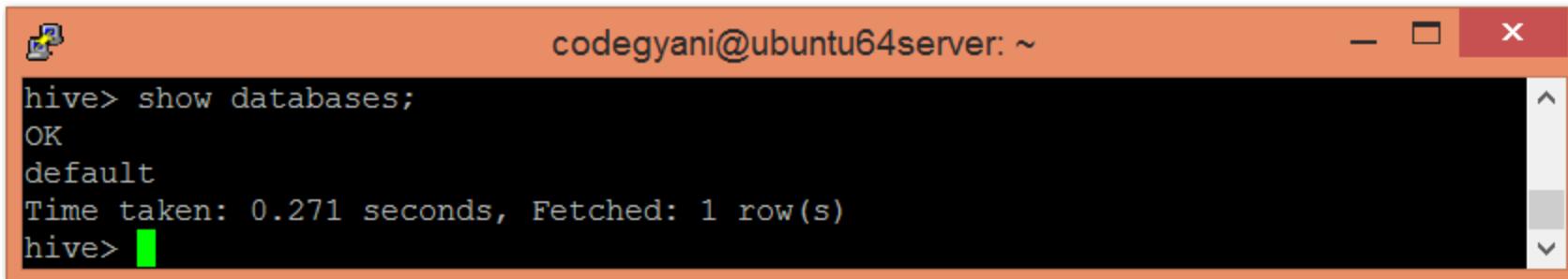
- Hive Defines various primitive, complex, string, date/time, collection data types and file formats.
- For handling & Storing different data formats.
- Table 1: Gives primitive, string , date/time and complex Hive data types & its description.

| Data Type Name | Description                                                                                                              |
|----------------|--------------------------------------------------------------------------------------------------------------------------|
| TINYINT        | 1 byte signed integer. Postfix letter is Y.                                                                              |
| SMALLINT       | 2 byte signed integer. Postfix letter is S.                                                                              |
| INT            | 4 byte signed integer                                                                                                    |
| BIGINT         | 8 byte signed integer. Postfix letter is L.                                                                              |
| FLOAT          | 4 byte single-precision floating-point number                                                                            |
| DOUBLE         | 8 byte double-precision floating-point number                                                                            |
| BOOLEAN        | True or False                                                                                                            |
| TIMESTAMP      | UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format “YYYY-MM-DD HH:MM:SS.fffffffff” |
| DATE           | YYYY-MM-DD format                                                                                                        |
| VARCHAR        | 1 to 65355 bytes. Use single quotes (‘ ’) or double quotes (“ ”)                                                         |
| CHAR           | 255 bytes                                                                                                                |
| DECIMAL        | Used for representing immutable arbitrary precision. DECIMAL (precision, scale) format                                   |

## Hive - Create Database

- In Hive, the database is considered as a **catalog or namespace of tables**. So, we can maintain **multiple tables within a database** where a **unique name is assigned to each table**. Hive also provides a **default database** with a name **default**.
- check the list of existing databases, follow the below command:-

```
hive> show databases;
```

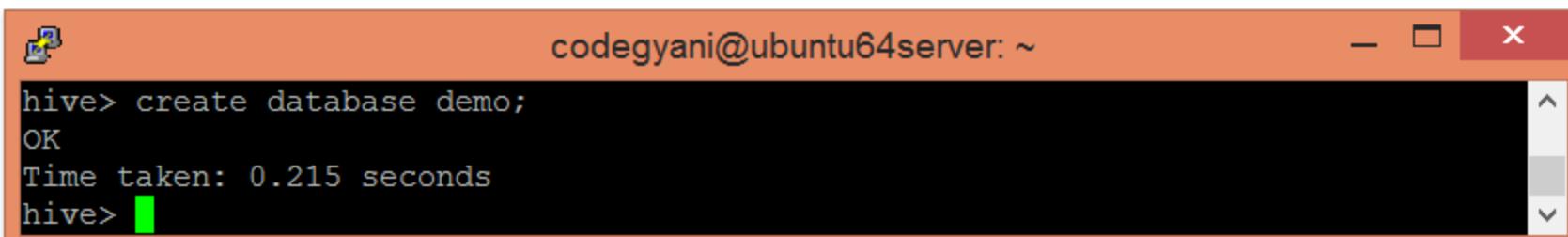


A screenshot of a terminal window titled "codegyani@ubuntu64server: ~". The window contains the following text:

```
hive> show databases;
OK
default
Time taken: 0.271 seconds, Fetched: 1 row(s)
hive> █
```

- Let's create a new database by using the following command:-

```
hive> create database demo;
```



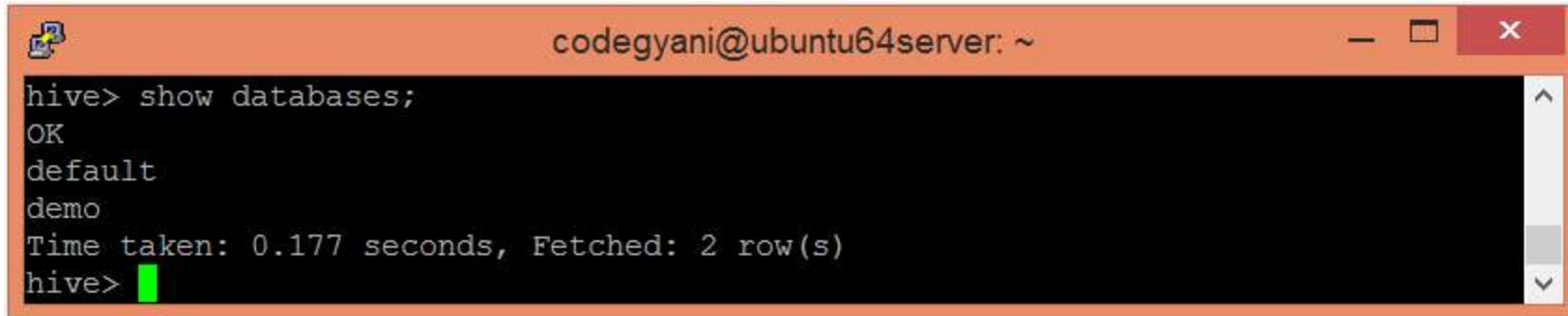
A screenshot of a terminal window titled "codegyani@ubuntu64server: ~". The window contains the following text:

```
hive> create database demo;
OK
Time taken: 0.215 seconds
hive> █
```

## Hive - Create Database

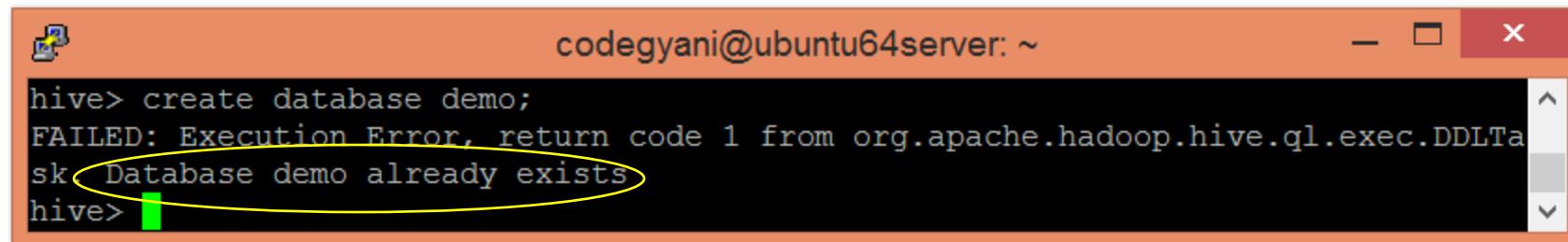
- Let's check the existence of a newly created database.

```
hive> show databases;
```



```
hive> show databases;
OK
default
demo
Time taken: 0.177 seconds, Fetched: 2 row(s)
hive>
```

- Each database must contain a **unique name**. If we create **two databases with the same name**, the following **error** generates: -



```
hive> create database demo;
FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask
sk Database demo already exists
hive>
```

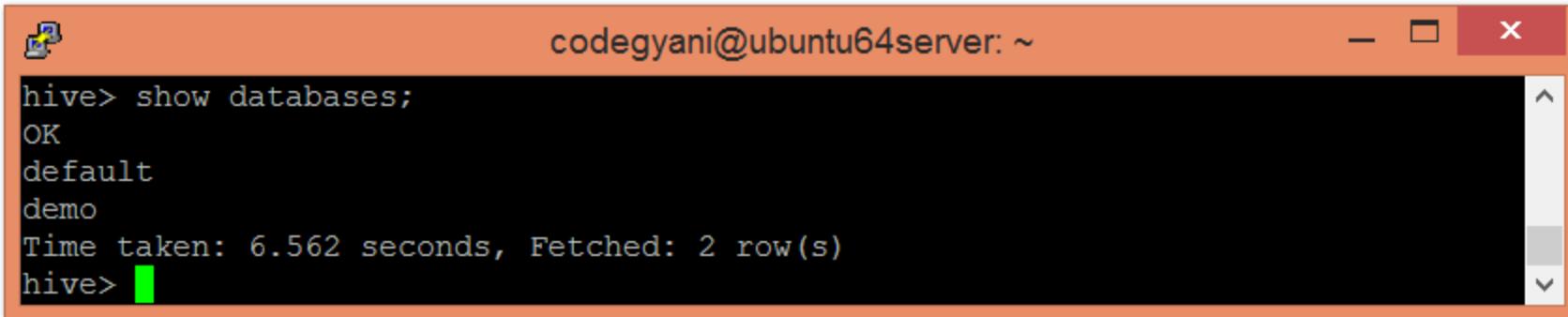
- If we want to **suppress the warning** generated by Hive on creating the database with the same name, follow the below command: -

```
hive> create database if not exists demo;
```

## Hive - Drop Database

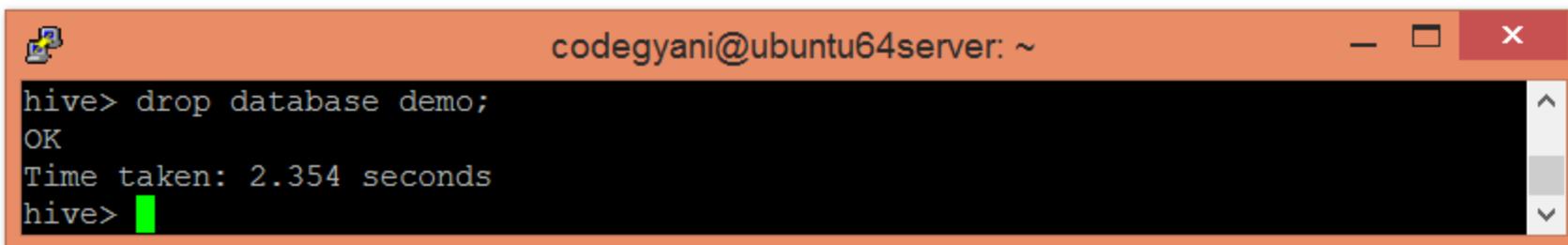
- check the **list of existing databases** by using the following command:-

```
hive> show databases;
```



```
codegyani@ubuntu64server: ~
hive> show databases;
OK
default
demo
Time taken: 6.562 seconds, Fetched: 2 row(s)
hive>
```

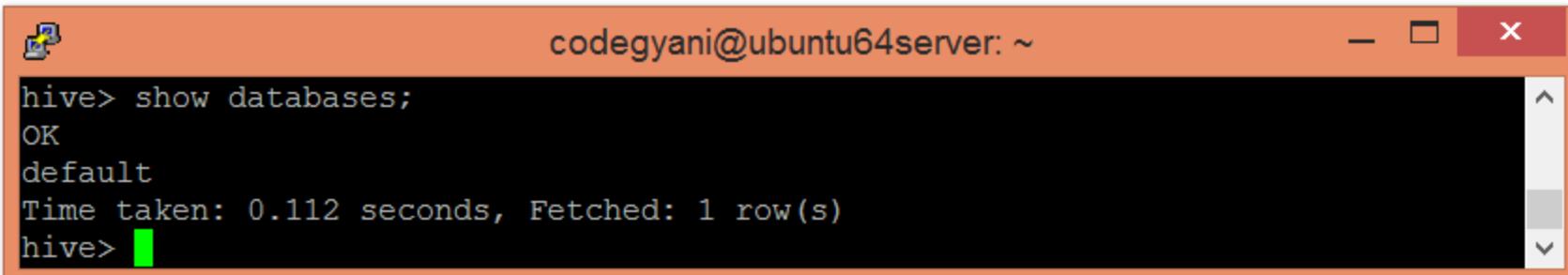
```
hive> drop database demo;
```



```
codegyani@ubuntu64server: ~
hive> drop database demo;
OK
Time taken: 2.354 seconds
hive>
```

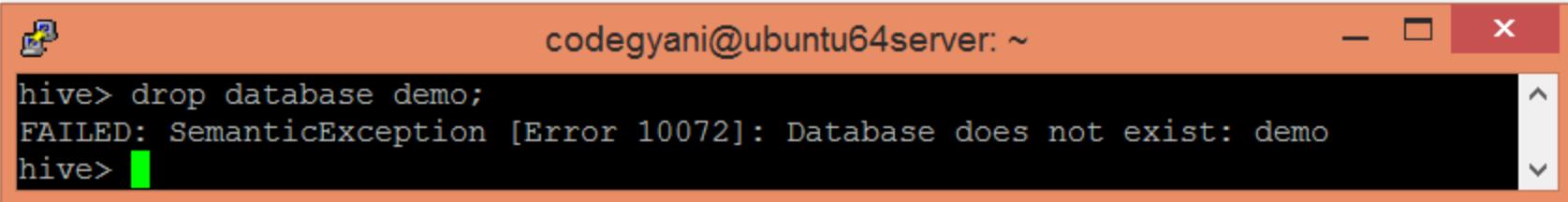
Let's check whether the database is dropped or not.

```
hive> show databases;
```



```
codegyani@ubuntu64server: ~
hive> show databases;
OK
default
Time taken: 0.112 seconds, Fetched: 1 row(s)
hive>
```

- If we try to drop the database that doesn't exist, the following error generates:



A screenshot of a terminal window titled "codegyani@ubuntu64server: ~". The window contains the following text:  
hive> drop database demo;  
FAILED: SemanticException [Error 10072]: Database does not exist: demo  
hive>

- if we want to suppress the warning generated by Hive on creating the database with the same name, follow the below command:-

hive> drop database if exists demo;

- In Hive, it is not allowed to drop the database that contains the tables directly. In such a case, we can drop the database either by dropping tables first or use Cascade keyword with the command.

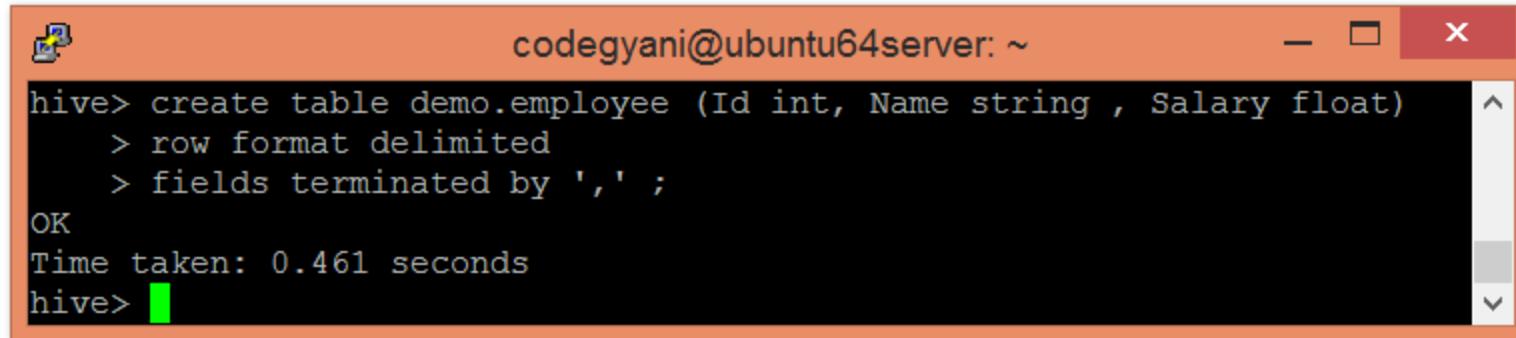
hive> drop database if exists demo cascade;

## Hive - Create Table

- In Hive, we can create a table by using the **conventions similar to the SQL**. It supports a wide range of flexibility where the data files for tables are stored. It provides two types of table:-

- create an **internal/managed table** by using the following command:-

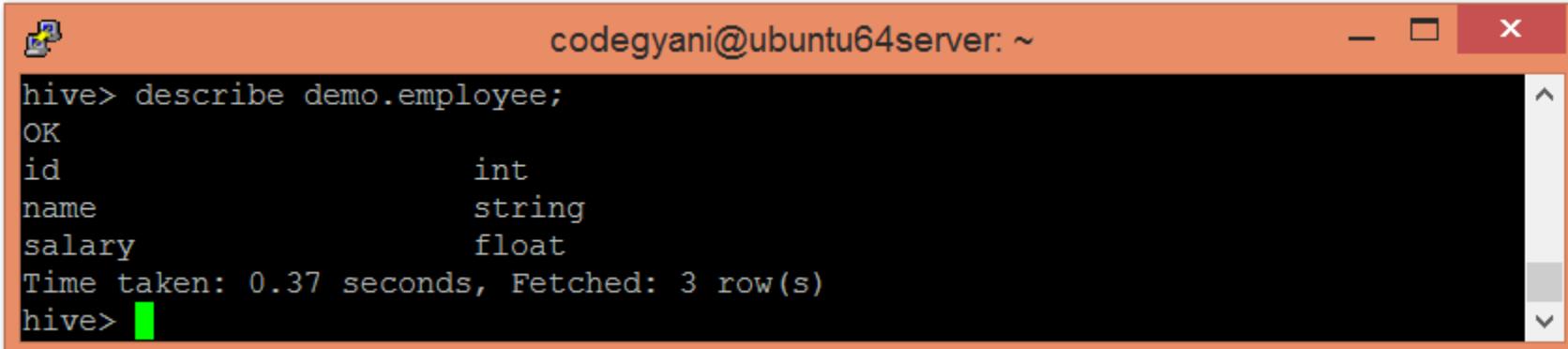
```
hive> create table demo.employee (Id int, Name string , Salary float)
row format delimited
fields terminated by ',';
```



A screenshot of a terminal window titled "codegyani@ubuntu64server: ~". The window contains the following text:

```
hive> create table demo.employee (Id int, Name string , Salary float)
 > row format delimited
 > fields terminated by ',' ;
OK
Time taken: 0.461 seconds
hive>
```

- The **metadata** of the created table by using the following command:-



A screenshot of a terminal window titled "codegyani@ubuntu64server: ~". The window contains the following text:

```
hive> describe demo.employee;
OK
id int
name string
salary float
Time taken: 0.37 seconds, Fetched: 3 row(s)
hive>
```

## Hive - Create Table

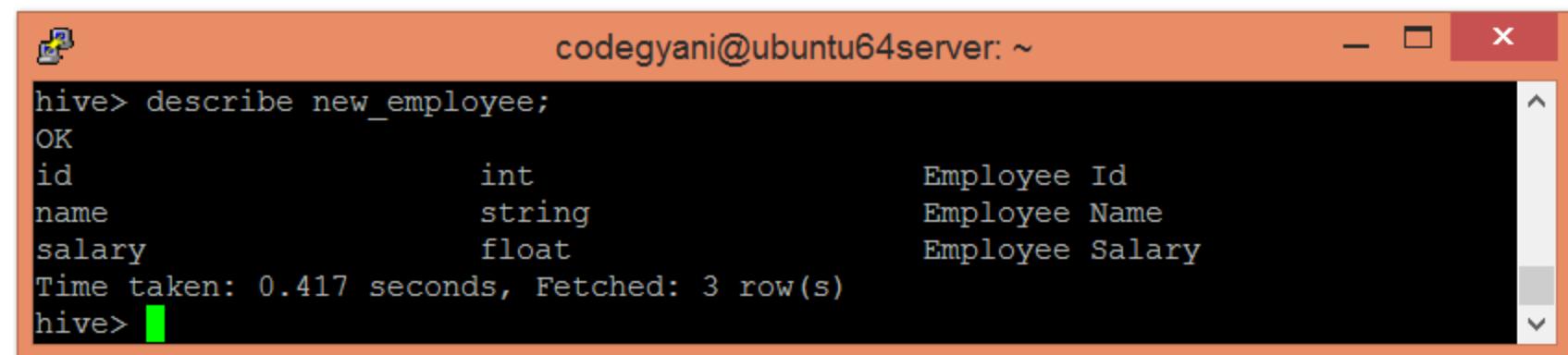
- Let's see the result when we try to **create the existing table again**.
- In such a case, the exception occurs. If we want to ignore this type of exception, we can use **if not exists** command while creating the table.
- While creating a table, we **can add the comments to the columns and can also define the table properties**.

```
codegyani@ubuntu64server: ~
hive> create table demo.employee (Id int, Name string , Salary float)
 > row format delimited
 > fields terminated by ',' ;
FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask. AlreadyExistsException(message:Table employee already exists)
hive>
```

```
codegyani@ubuntu64server: ~
hive> create table if not exists demo.employee (Id int, Name string , Salary float)
 > row format delimited
 > fields terminated by ',' ;
OK
Time taken: 0.166 seconds
hive>
```

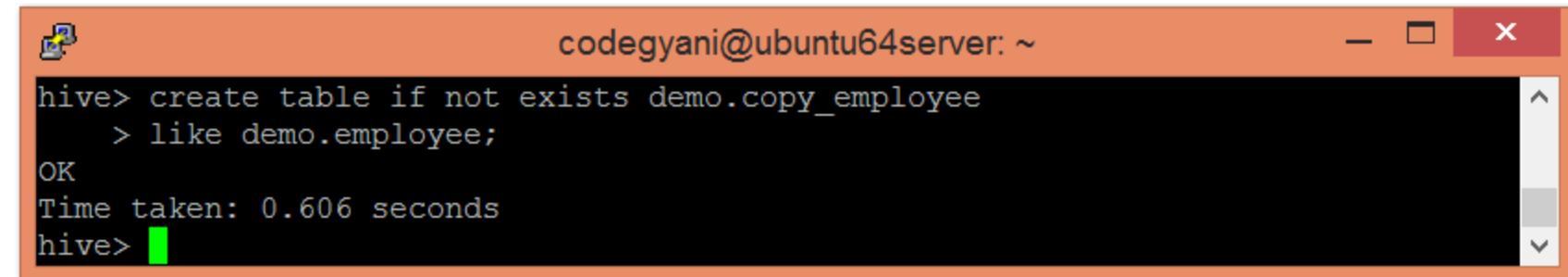
```
codegyani@ubuntu64server: ~
hive> create table demo.new_employee (Id int comment 'Employee Id', Name string
 comment 'Employee Name', Salary float comment 'Employee Salary')
 > comment 'Table Description'
 > TBLProperties ('creator'='Gaurav Chawla', 'created_at' = '2019-06-06 11:00')
 > ;
OK
Time taken: 3.236 seconds
```

```
hive> describe new_employee;
```



```
hive> describe new_employee;
OK
id int Employee Id
name string Employee Name
salary float Employee Salary
Time taken: 0.417 seconds, Fetched: 3 row(s)
hive>
```

- Hive allows creating a **new table** by using the schema of an existing table.



```
hive> create table if not exists demo.copy_employee
 > like demo.employee;
OK
Time taken: 0.606 seconds
hive>
```

## Hive - Load Data

Once the **internal table has been created**, the next step is to **load the data into it**. So, in Hive, we can easily load data from **any file to the database**.

- Let's load the data of the file into the database by using the following command:-
- Here, **emp\_details** is the file name that contains the data.
- If we want to **add more data** into the current database, execute the same query again by **just updating the new file name**.

```
load data local inpath
'/home/codegyani/hive/emp
_details1' into table
demo.employee;
```

```
hive> load data local inpath '/home/codegyani/hive/emp_details' into table demo.
employee;
Loading data to table demo.employee
Table demo.employee stats: [numFiles=1, numRows=0, totalSize=50, rawDataSize=0]
OK
Time taken: 7.505 seconds
hive>
```

```
hive> select * from demo.employee;
OK
1 "Gaurav" 30000.0
2 "Aryan" 20000.0
3 "Vishal" 40000.0
Time taken: 3.32 seconds, Fetched: 3 row(s)
hive>
```

Hadoop Overview Datanodes Snapshot Startup Progress Utilities

## Browse Directory

/user/hive/warehouse/demo.db/employee

Go!

| Permission | Owner     | Group      | Size | Last Modified         | Replication | Block Size | Name        |
|------------|-----------|------------|------|-----------------------|-------------|------------|-------------|
| -rwxr-xr-x | codegyani | supergroup | 50 B | 4/15/2019, 7:41:53 PM | 1           | 128 MB     | emp_details |

## External Table

To create an external table, follow the below steps:-

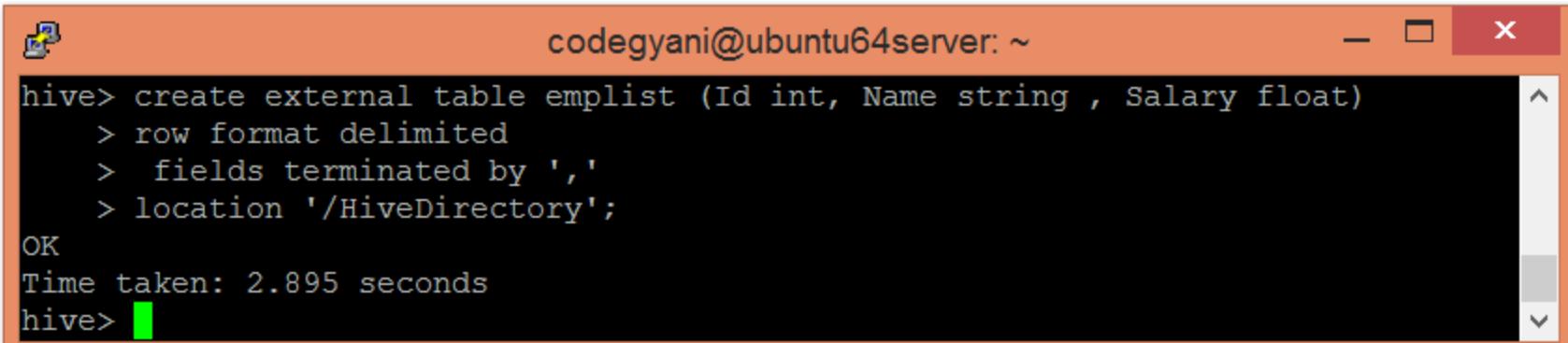
- Let's create a directory on HDFS by using the following command:-

```
hdfs dfs -mkdir /HiveDirectory
```

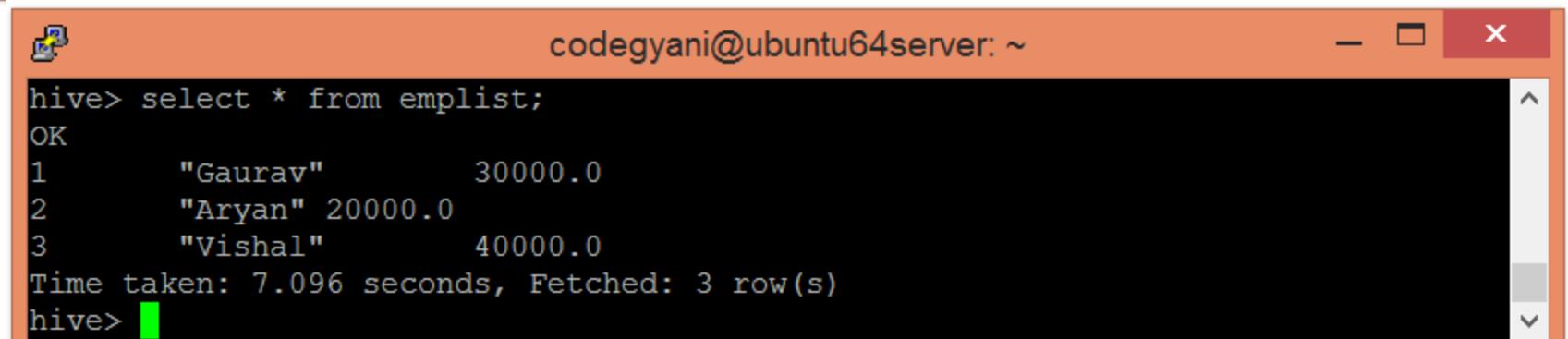
- Now, store the file on the created directory.

```
hdfs dfs -put hive/emp_details /HiveDirectory
```

- Let's create an external table using the following command:-



```
hive> create external table emplist (Id int, Name string , Salary float)
 > row format delimited
 > fields terminated by ','
 > location '/HiveDirectory';
OK
Time taken: 2.895 seconds
hive>
```



```
hive> select * from emplist;
OK
1 "Gaurav" 30000.0
2 "Aryan" 20000.0
3 "Vishal" 40000.0
Time taken: 7.096 seconds, Fetched: 3 row(s)
hive>
```

- Now, we can use the following command to retrieve the data:-

- In Hive, if we try to **load unmatched data** (i.e., **one or more column data doesn't match the data type of specified table columns**), it will **not throw any exception**. However, **it stores the Null value at the position of unmatched tuple**.

- Let's add one more file to the current table. This file contains the unmatched data.

```
codegyani@ubuntu64server: ~/hive
GNU nano 2.2.6 File: emp_details2
1, "John", "Woakes"
2, "Henry", "William"
3, "Shaun", "Morris"

[Read 3 lines]
^G Get Help ^O WriteOut ^R Read File^Y Prev Page^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page^U Uncut Tex^T To Spell
```

- load the data into the table.

```
codegyani@ubuntu64server: ~/hive
hive> load data local inpath '/home/codegyani/hive/emp_details2' into table demo
 .employee;
Loading data to table demo.employee
Table demo.employee stats: [numFiles=3, numRows=0, totalSize=138, rawDataSize=0]
OK
Time taken: 15.812 seconds
hive>
```

- Null values at the position of unmatched data.

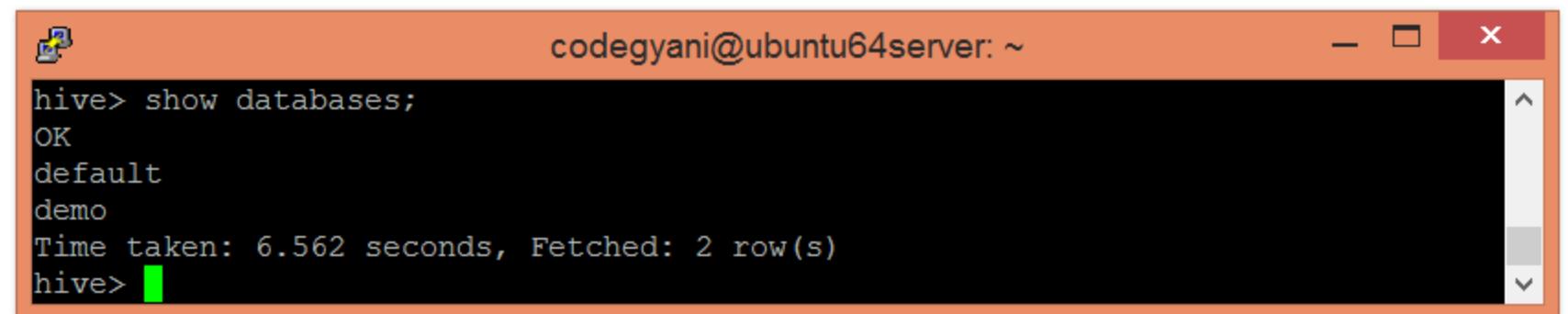


```
codegyani@ubuntu64server: ~/hive
hive> select * from demo.employee;
OK
1 "Gaurav" 30000.0
2 "Aryan" 20000.0
3 "Vishal" 40000.0
4 "John" 10000.0
5 "Henry" 25000.0
1 "John" NULL
2 "Henry" NULL
3 "Shaun" NULL
Time taken: 0.506 seconds, Fetched: 8 row(s)
hive>
```

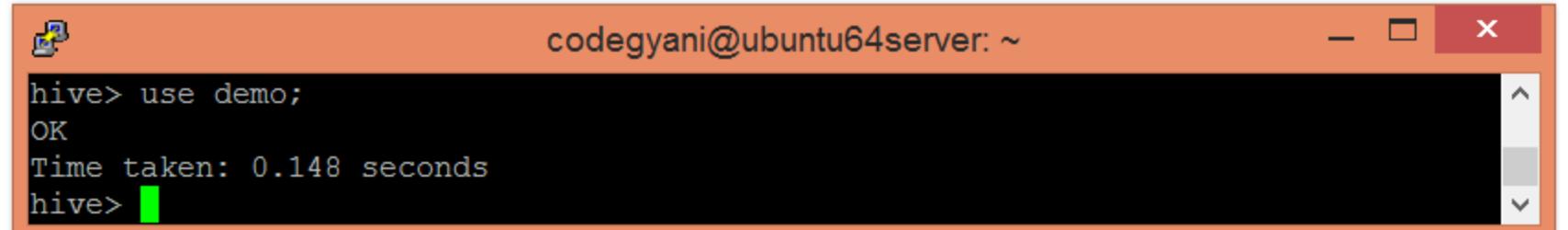
## Hive - Drop Table

Hive facilitates us to drop a table by using the **SQL drop table command**. Let's follow the below steps to drop the table from the database.

- check the list of existing databases by using the following command:-
- select the **database from which we want to delete the table**

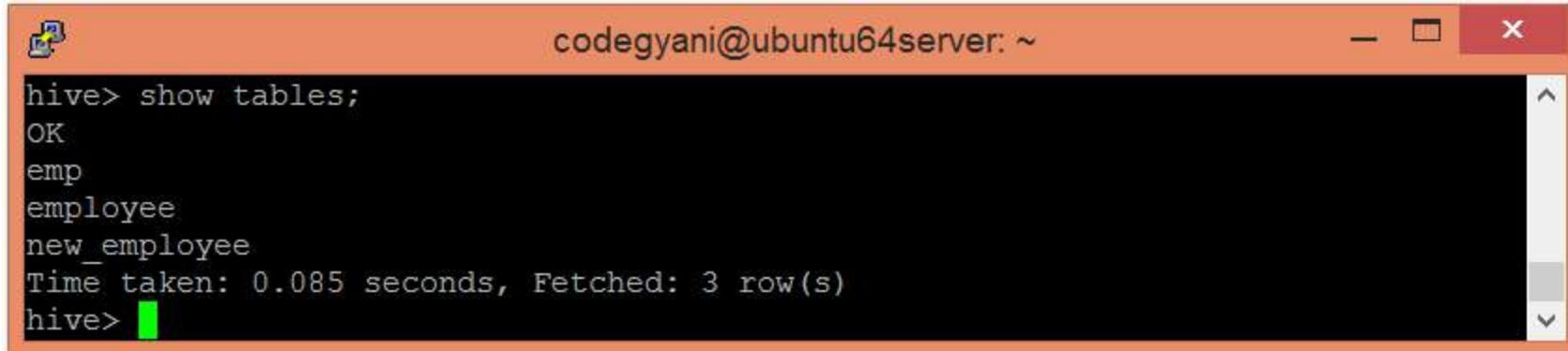


```
codegyani@ubuntu64server: ~
hive> show databases;
OK
default
demo
Time taken: 6.562 seconds, Fetched: 2 row(s)
hive>
```



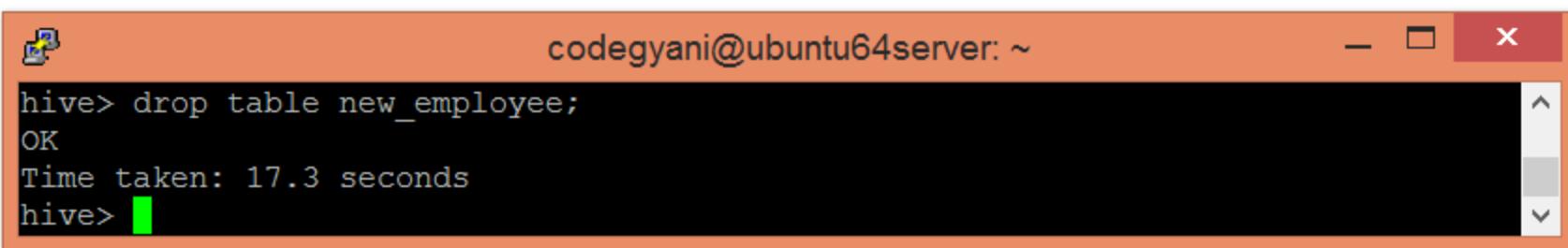
```
codegyani@ubuntu64server: ~
hive> use demo;
OK
Time taken: 0.148 seconds
hive>
```

- check the list of **existing tables** in the corresponding database.



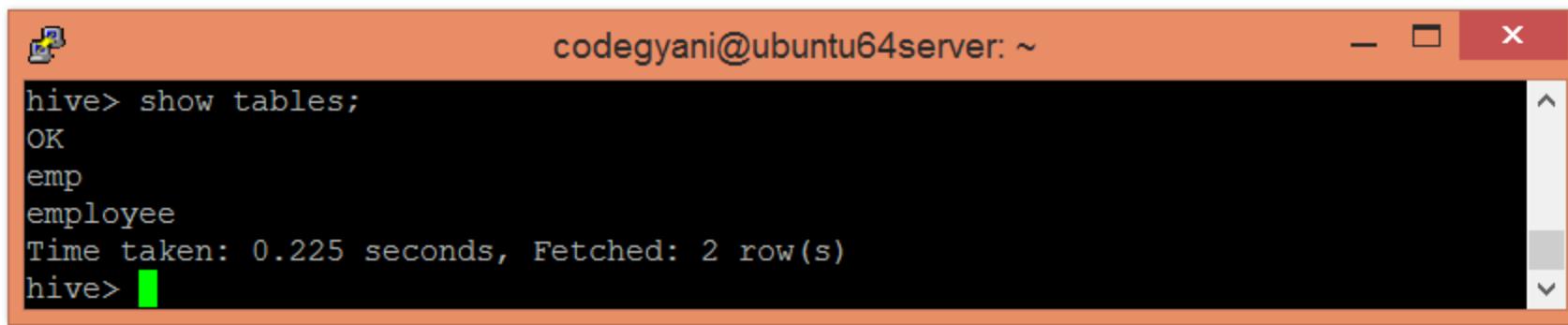
```
hive> show tables;
OK
emp
employee
new_employee
Time taken: 0.085 seconds, Fetched: 3 row(s)
hive>
```

- drop the table by using the following command:



```
hive> drop table new_employee;
OK
Time taken: 17.3 seconds
hive>
```

- check whether the table is dropped or not.
- As we can see, the table **new\_employee** is not present in the list. Hence, the table is dropped successfully.

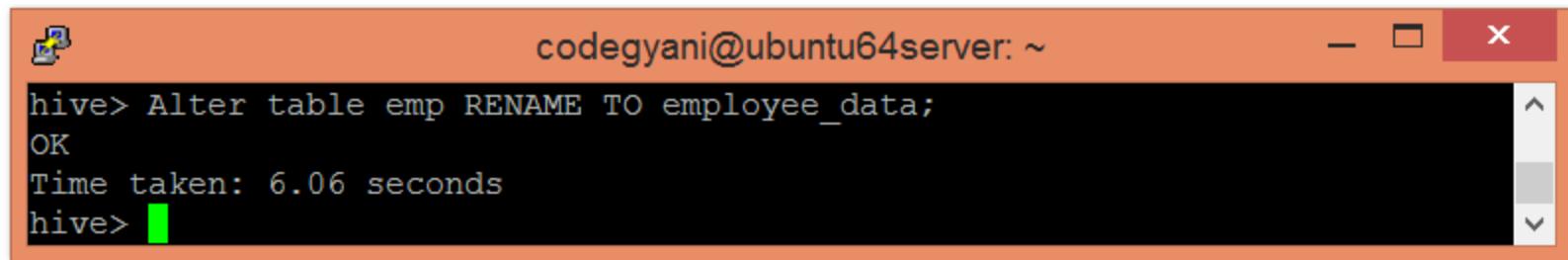


```
hive> show tables;
OK
emp
employee
Time taken: 0.225 seconds, Fetched: 2 row(s)
hive>
```

# Hive - Alter Table

In Hive, we can perform modifications in the existing table like changing the table name, column name, comments, and table properties. It provides SQL like commands to alter the table.

- change the name of the table by using the following command



A screenshot of a terminal window titled "codegyani@ubuntu64server: ~". The window contains the following text:

```
hive> Alter table emp RENAME TO employee_data;
OK
Time taken: 6.06 seconds
hive>
```

- In Hive, we can add one or more columns in an existing table by using the following signature:

**Alter table employee\_data add columns (age int);**

- change the name of the column by using the following command: -

**Alter table employee\_data change name first\_name string;**

- alter table employee\_data replace columns( id string, first\_name string, age int);**

# Partitioning in Hive

The partitioning in Hive can be executed **in two ways** -

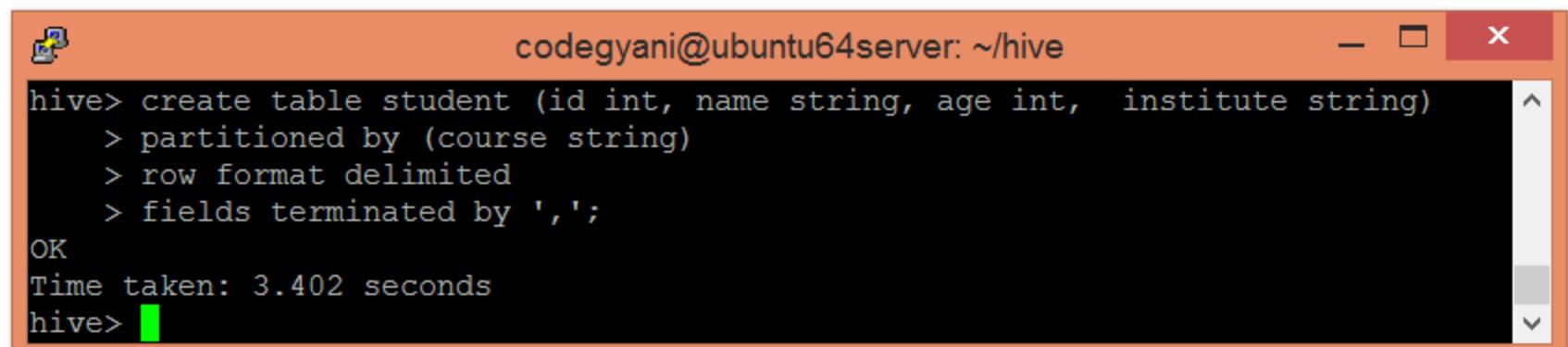
- Static partitioning
  - Dynamic partitioning
- 
- In static or manual partitioning, it is required to **pass the values of partitioned columns manually while loading the data into the table**. Hence, the data file doesn't contain the partitioned columns.

## Example of Static Partitioning

First, select the database in which we want to create a table.

```
hive> use test;
```

- Create the table and provide the partitioned columns by using the following command



A screenshot of a terminal window titled "codegyani@ubuntu64server: ~/hive". The window shows the following command being run:

```
hive> create table student (id int, name string, age int, institute string)
 > partitioned by (course string)
 > row format delimited
 > fields terminated by ',';
```

The command is completed with "OK" and a timestamp "Time taken: 3.402 seconds". The prompt "hive>" is visible at the bottom.

- Let's retrieve the information associated with the table.

```
hive> describe student;
OK
id int
name string
age int
institute string
course string

Partition Information
col_name data_type comment
course string

Time taken: 1.833 seconds, Fetched: 10 row(s)
hive>
```

- Load the data into the table and pass the values(key) of partition columns with it by using the following command:-

- Here, we are **partitioning** the students of an institute based on courses.

- Load the data of another file into the same table and pass the values of partition columns with it by using the following command:-

```
hive> load data local inpath '/home/codegyani/hive/student_details1' into table student
 > partition(course= "java");
Loading data to table test.student partition (course=jav
Partition test.student{course=jav} stats: [numFiles=1, numRows=0, totalSize=122
, rawDataSize=0]
OK
Time taken: 8.057 seconds
hive>
```

```
hive> load data local inpath '/home/codegyani/hive/student_details2' into table student
 > partition(course= "hadoop");
Loading data to table test.student partition (course=hadoop)
Partition test.student{course=hadoop} stats: [numFiles=1, numRows=0, totalSize=7
5, rawDataSize=0]
OK
Time taken: 2.402 seconds
hive>
```

- In the following screenshot, we can see that the **table student** is divided into two categories.

Hadoop Overview Datanodes Snapshot Startup Progress Utilities

## Browse Directory

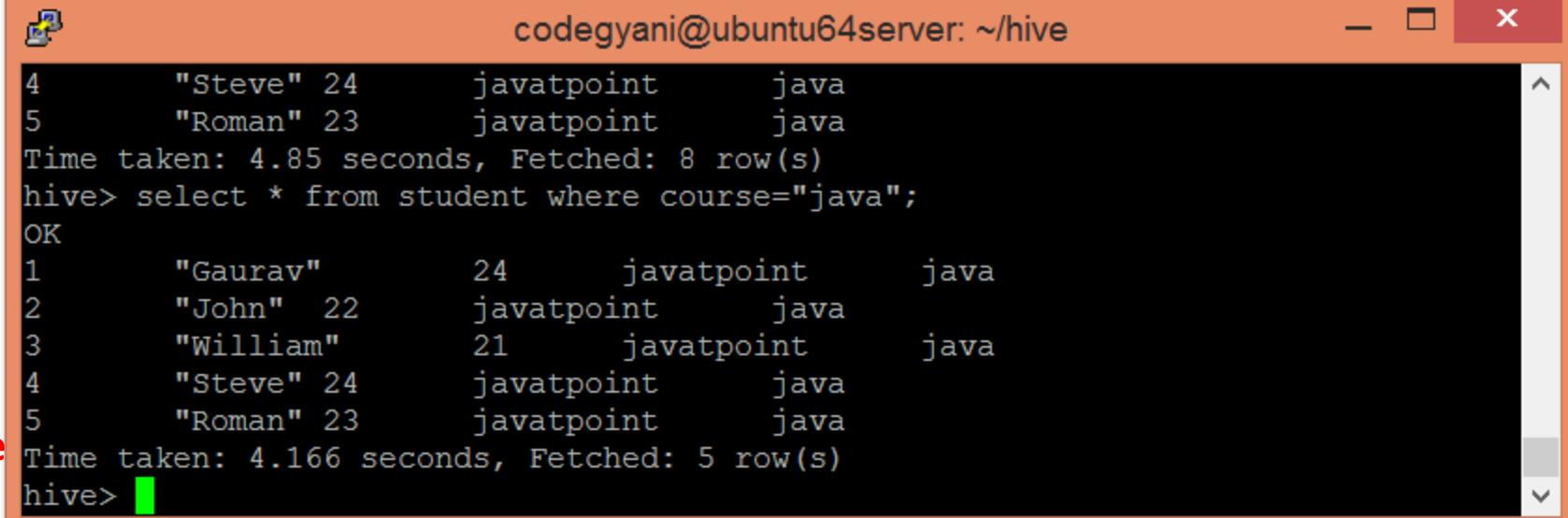
/user/hive/warehouse/test.db/student

| Permission | Owner     | Group      | Size | Last Modified        | Replication | Block Size | Name          |
|------------|-----------|------------|------|----------------------|-------------|------------|---------------|
| drwxr-xr-x | codegyani | supergroup | 0 B  | 8/1/2019, 5:39:27 PM | 0           | 0 B        | course=hadoop |
| drwxr-xr-x | codegyani | supergroup | 0 B  | 8/1/2019, 5:37:13 PM | 0           | 0 B        | course=java   |

- Let's retrieve the entire data of the table by using the following command

```
hive> select * from student;
OK
6 "Chris" 22 javatpoint hadoop
7 "Hariss" 21 javatpoint hadoop
8 "Angelina"24 NULL NULL hadoop
1 "Gaurav" 24 javatpoint java
2 "John" 22 javatpoint java
3 "William" 21 javatpoint java
4 "Steve" 24 javatpoint java
5 "Roman" 23 javatpoint java
Time taken: 4.85 seconds, Fetched: 8 row(s)
hive>
```

- try to retrieve the data based on **partitioned columns** by using the following command: -
- In this case, we are **not examining the entire data**. Hence, this approach **improves query response time**.



The screenshot shows a terminal window with the title "codegyani@ubuntu64server: ~/hive". The window displays the output of a Hive query. The query is:

```
hive> select * from student where course="java";
OK
1 "Gaurav" 24 javatpoint java
2 "John" 22 javatpoint java
3 "William" 21 javatpoint java
4 "Steve" 24 javatpoint java
5 "Roman" 23 javatpoint java
Time taken: 4.166 seconds, Fetched: 5 row(s)
hive>
```

The output shows five rows of data for students taking the "java" course. The columns are name, age, and two additional columns which are likely part of a partitioned key.

## Dynamic Partitioning

In dynamic partitioning, the **values of partitioned columns exist within the table**. So, it is not required to pass the values of partitioned columns manually.

- First, select the **database** in which we **want to create a table**.
- Enable the dynamic partition** by using the following →  
`hive> set hive.exec.dynamic.partition=true;`  
`hive> set hive.exec.dynamic.partition.mode=nonstrict;`

- Create a **dummy table** to store the data.

```
codegyani@ubuntu64server: ~/hive
hive> create table stud_demo(id int, name string, age int, institute string, course string)
 > row format delimited
 > fields terminated by ',';
OK
```

- Now, load the data into **dummy table**.

```
codegyani@ubuntu64server: ~/hive
hive> load data local inpath '/home/codegyani/hive/student_details' into table stud_demo;
Loading data to table show.stud_demo
Table show.stud_demo stats: [numFiles=1, totalSize=152]
OK
```

- Create a **partition table** by using the following command

```
codegyani@ubuntu64server: ~/hive
hive> create table student_part (id int, name string, age int, institute string)
 > partitioned by (course string)
 > row format delimited
 > fields terminated by ',';
OK
```

- Now, **insert the data of dummy table into the partition table.**

→ **hive> insert into student\_part  
partition(course)  
select id, name, age, institute, course  
from stud\_demo;**

- In the following screenshot, we can see that the table **student\_part** is divided into two categories.

The screenshot shows the Hadoop Web UI's 'Browse Directory' interface. The URL in the address bar is /user/hive/warehouse/show.db/student\_part. The table has the following data:

| Permission | Owner     | Group      | Size | Last Modified        | Replication | Block Size | Name                             |
|------------|-----------|------------|------|----------------------|-------------|------------|----------------------------------|
| drwxr-xr-x | codegyani | supergroup | 0 B  | 8/1/2019, 3:53:17 PM | 0           | 0 B        | course=__HIVE_DEFAULT_PARTITION_ |
| drwxr-xr-x | codegyani | supergroup | 0 B  | 8/1/2019, 3:53:15 PM | 0           | 0 B        | course=hadoop                    |
| drwxr-xr-x | codegyani | supergroup | 0 B  | 8/1/2019, 3:53:16    | 0           | 0 B        | course=java                      |

- Let's retrieve the entire data of the table by using the following command

```
hive> select * from student_part;
OK
NULL NULL NULL NULL __HIVE_DEFAULT_PARTITION__
2 "John" 22 javatpoint hadoop
3 "William" 21 javatpoint hadoop
1 "Gaurav" 24 javatpoint java
4 "Steve" 24 javatpoint java
5 "Roman" 23 javatpoint java
Time taken: 1.231 seconds, Fetched: 6 row(s)
hive>
```

- Now, try to retrieve the data based on partitioned columns by using the following command:

```
codegyani@ubuntu64server: ~/hive
hive> select * from student_part where course= "java";
OK
1 "Gaurav" 24 javatpoint java
4 "Steve" 24 javatpoint java
5 "Roman" 23 javatpoint java
Time taken: 0.569 seconds, Fetched: 3 row(s)
hive> [REDACTED]
```

- Let's also retrieve the data of another partitioned dataset by using the following command

```
codegyani@ubuntu64server: ~/hive
hive> select * from student_part where course= "hadoop";
OK
2 "John" 22 javatpoint hadoop
3 "William" 21 javatpoint hadoop
Time taken: 0.914 seconds, Fetched: 2 row(s)
hive> [REDACTED]
```

## Bucketing in Hive

- The bucketing in Hive is a data organizing technique. It is similar to partitioning in Hive with an added functionality that it **divides large datasets into more manageable parts** known as buckets.

- First, select the **database in which we want to create a table**.

```
hive> use showbucket;
OK
Time taken: 0.111 seconds
hive>
```

- Create a dummy table to store the data.

```
hive> create table emp_demo (Id int, Name string , Salary float)
 > row format delimited
 > fields terminated by ',' ;
OK
Time taken: 0.373 seconds
hive>
```

- Now, **load the data** into the table.

```
hive> load data local inpath '/home/codegyani/hive/emp_details' into table emp_demo;
Loading data to table showbucket.emp_demo
Table showbucket.emp_demo stats: [numFiles=1, totalSize=131]
OK
Time taken: 1.333 seconds
hive>
```

# Bucketing in Hive

- Enable the bucketing by using the following command

```
hive> set hive.enforce.bucketing = true;
codegyani@ubuntu64server: ~
hive> create table emp_bucket(Id int, Name string , Salary float)
 > clustered by (Id) into 3 buckets
 > row format delimited
 > fields terminated by ',' ;
OK
Time taken: 0.308 seconds
hive>
```

- Create a **bucketing table** by using the following command:

- Now, **insert the data of dummy table** into the bucketed table.

↓  
**hive> insert overwrite table emp\_bucket select \* from emp\_demo;**

- Here, we can see that the data is divided into three buckets.

Hadoop Overview Datanodes Snapshot Startup Progress Utilities ▾

## Browse Directory

/user/hive/warehouse/showbucket.db/emp\_bucket Go!

| Permission | Owner     | Group      | Size | Last Modified        | Replication | Block Size | Name     |
|------------|-----------|------------|------|----------------------|-------------|------------|----------|
| -rwxr-xr-x | codegyani | supergroup | 56 B | 8/2/2019, 4:11:20 AM | 1           | 128 MB     | 000000_0 |
| -rwxr-xr-x | codegyani | supergroup | 53 B | 8/2/2019, 4:11:20 AM | 1           | 128 MB     | 000001_0 |
| -rwxr-xr-x | codegyani | supergroup | 54 B | 8/2/2019, 4:11:20 AM | 1           | 128 MB     | 000002_0 |

- Let's retrieve the data of bucket 0.

According to hash function :

$$6 \% 3 = 0$$

$$3 \% 3 = 0$$

So, these columns stored in bucket 0.

```
codegyani@ubuntu64server: ~
codegyani@ubuntu64server:~$ hdfs dfs -cat /user/hive/warehouse/showbucket.db/emp
 _bucket/000000_0;
\N,\N,\N
\N,\N,\N
6,"William",9000.0
3,"Vishal",40000.0
```

- Let's retrieve the data of bucket 1

According to hash function :

$$7 \% 3 = 1$$

$$4 \% 3 = 1$$

$$1 \% 3 = 1$$

So, these columns stored in bucket 1.

```
codegyani@ubuntu64server: ~
codegyani@ubuntu64server:~$ hdfs dfs -cat /user/hive/warehouse/showbucket.db/emp
 _bucket/000001_0;
7,"Lisa",25000.0
4,"John",10000.0
1,"Gaurav",30000.0
```

- Let's retrieve the data of bucket 2.

According to hash function :

$$8 \% 3 = 2$$

$$5 \% 3 = 2$$

$$2 \% 3 = 2$$

So, these columns stored in bucket 2.

```
codegyani@ubuntu64server: ~
codegyani@ubuntu64server:~$ hdfs dfs -cat /user/hive/warehouse/showbucket.db/emp
 _bucket/000002_0;
8,"Ronit",20000.0
5,"Henry",25000.0
2,"Aryan",20000.0
```

# File Formats & Descriptions:

| File Format     | Description                                                                                                                                          |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| Text file       | The default file format, and a line represents a record. The delimiting characters separate the lines. Text file examples are CSV, TSV, JSON and XML |
| Sequential file | Flat file which stores binary key-value pairs, and supports compression.                                                                             |
| RCFile          | Record Columnar file                                                                                                                                 |
| ORCFILE         | ORC stands for Optimized Row Columnar which means it can store data in an optimized way than in the other file formats                               |

## Hive Text File Format

- Hive **Text file format is a default storage format.**
- You can use the text format to **interchange** the data with **other client application.**
- Create a TEXT file by add **storage option** as '**STORED AS TEXTFILE**' at the end of a Hive CREATE TABLE command.

### Hive Text File Format Examples

Below is the Hive **CREATE TABLE** command with **storage format** specification:

Create table textfile\_table (column\_specs) **stored as textfile;**

## Hive Sequence File Format

- Sequence files are Hadoop flat files which stores values in **binary key-value pairs**.
- The sequence files are **in binary format** and these files are **able to split**.
- The main **advantages** of using sequence file is to **merge two or more files** into one file.
- Create a sequence file by add storage option as '**STORED AS SEQUENCEFILE**' at the end of a Hive CREATE TABLE command.

### Hive Sequence File Format Example

Below is the Hive **CREATE TABLE** command **with storage format** specification:

**Create table sequencefile\_table (column\_specs) stored as sequencefile;**

## Hive RC File Format

- RCFfile is **row columnar file format**.
- This is another form of Hive file format which **offers high row level compression rates**.
- If you have requirement to **perform multiple rows at a time** then you can use RCFfile format.
- The RCFfile are very much similar to the sequence file format. This file format also stores the **data as key-value pairs**.
- Create RCFfile by specifying '**STORED AS RCFFILE**' option at the end of a CREATE TABLE Command:

### Hive RC File Format Example

Below is the Hive CREATE TABLE command with storage format specification:

**Create table RCfile\_table (column\_specs) stored as rcfle;**

## Hive RC File Format

RC files partitions the table first horizontally, and then vertically to serialize the data

**Table 9.1** A table with four columns

| C1 | C2 | C3 | C4 |
|----|----|----|----|
| 11 | 12 | 13 | 14 |
| 21 | 22 | 23 | 24 |
| 31 | 32 | 33 | 34 |
| 41 | 42 | 43 | 44 |
| 51 | 52 | 53 | 54 |

**Table 9.2** Table with two row groups

| Row Group 1 |    |    |    | Row Group 2 |    |    |    |
|-------------|----|----|----|-------------|----|----|----|
| C1          | C2 | C3 | C4 | C1          | C2 | C3 | C4 |
| 11          | 12 | 13 | 14 | 41          | 42 | 43 | 44 |
| 21          | 22 | 23 | 24 | 51          | 52 | 53 | 54 |
| 31          | 32 | 33 | 34 |             |    |    |    |

**Table 9.3** Table in RCFile Format

| Row Group 1 | Row Group 2 |
|-------------|-------------|
| 11, 21, 31; | 41, 51;     |
| 12, 22, 32; | 42, 52;     |
| 13, 23, 33; | 43, 53;     |
| 14, 24, 34; | 44, 54;     |

## Hive ORC File Format

- The ORC file stands for **Optimized Row Columnar** file format.
- The ORC file format provides a **highly efficient way to store data in Hive table**.
- This file system was actually designed to **overcome limitations of the other Hive file formats**.
- The Use of **ORC files improves performance** when **Hive is reading, writing, and processing data from large tables**.
- Due to their columnar storage format, they can **efficiently skip over irrelevant columns** when executing queries, leading to faster query performance.

Create ORC file by specifying '**STORED AS ORC**' option at the end of a CREATE TABLE Command.

### Hive ORC File Format Examples

Below is the Hive CREATE TABLE command with storage format specification:

**Create table orc\_table (column\_specs) stored as orc;**

## Hive Parquet File Format

- Parquet is a **column-oriented binary file format**.
- The parquet is **highly efficient for the types of large-scale queries**.
- Parquet is especially good for **queries scanning particular columns** within a particular table.
- The Parquet table uses compression **Snappy, gzip**; currently **Snappy by default**.

Create Parquet file by specifying ‘STORED AS PARQUET’ option at the end of a CREATE TABLE Command.

### Hive Parquet File Format Example

Below is the Hive CREATE TABLE command with storage format specification:

**Create table parquet\_table (column\_specs) stored as parquet;**

## Hive Parquet File Sample

File: C:\Users\...\Downloads\yellow\_tripdata\_2022-01.parquet

File Edit Tools Help

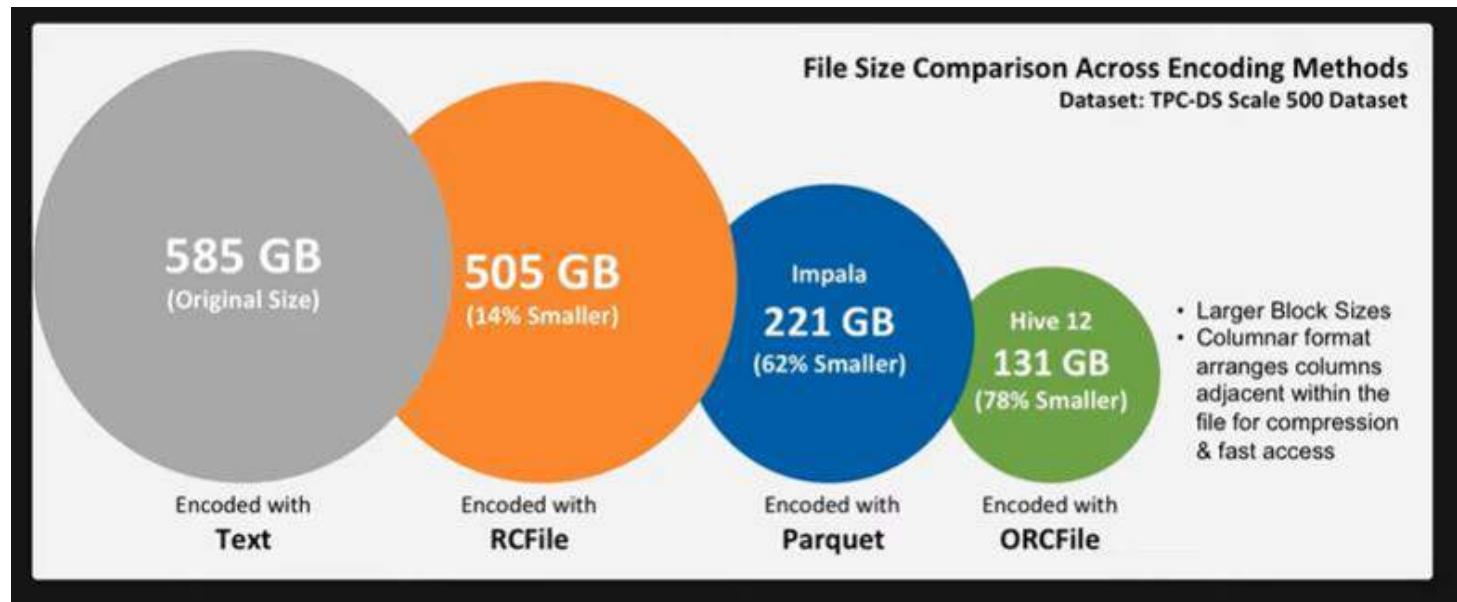
Filter Query (?): WHERE (tip\_amount \* 100) / fare\_amount > 60

Execute Clear Record Offset: 0 Record Count: 1000 ▶

|   | VendorID | fare_amount | tip_amount | tpep_pickup_datetime    | tpep_dropoff_datetime   | passenger_count | trip_distance | RatecodeID | store_and_1 |
|---|----------|-------------|------------|-------------------------|-------------------------|-----------------|---------------|------------|-------------|
| ▶ | 2        | 21          | 15         | 2022-01-01 00:55:48.000 | 2022-01-01 01:14:24.000 | 1               | 6.67          | 1          | N           |
|   | 1        | 7.5         | 20         | 2022-01-01 00:37:13.000 | 2022-01-01 00:44:46.000 | 1               | 1.9           | 1          | N           |
|   | 1        | 5.5         | 4          | 2022-01-01 00:57:44.000 | 2022-01-01 01:02:34.000 | 0               | 0.9           | 1          | N           |
|   | 2        | 4           | 10         | 2022-01-01 00:15:46.000 | 2022-01-01 00:17:41.000 | 2               | 0.6           | 1          | N           |
|   | 2        | 5.5         | 15         | 2022-01-01 00:23:02.000 | 2022-01-01 00:28:20.000 | 1               | 0.75          | 1          | N           |
|   | 1        | 3.5         | 3          | 2022-01-01 00:48:11.000 | 2022-01-01 00:50:08.000 | 1               | 0.4           | 1          | N           |
|   | 2        | 5.5         | 5          | 2022-01-01 00:28:57.000 | 2022-01-01 00:34:16.000 | 1               | 0.67          | 1          | N           |
|   | 2        | 6           | 4          | 2022-01-01 00:15:47.000 | 2022-01-01 00:21:18.000 | 1               | 1.27          | 1          | N           |
|   | 1        | 6           | 20         | 2022-01-01 00:10:22.000 | 2022-01-01 00:15:53.000 | 2               | 1.2           | 1          | N           |
|   | 2        | 3           | 2.04       | 2022-01-01 00:25:52.000 | 2022-01-01 00:27:12.000 | 1               | 0.26          | 1          | N           |

# Memory usage of different file types

- ORC using “Columnar Format of Storage” .
- Columnar oriented storage is always **faster** than row-oriented storage.



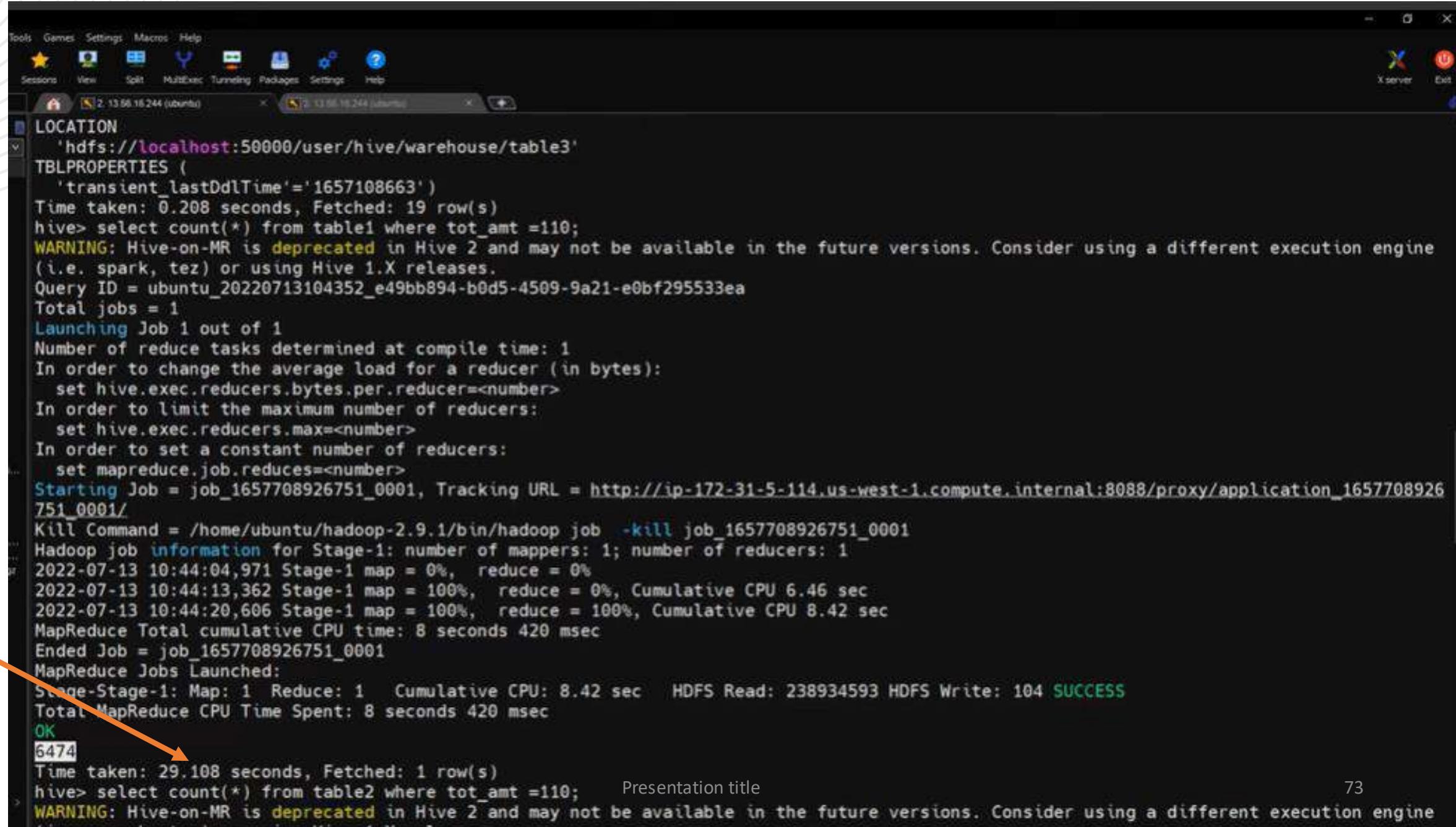
## Example (File Format): Time Difference & volume of Data

```
w20 - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
new 1 new 2 new 3 new 4 new 5 new 6 new 7 new 8 new 9 new 10 new 11 new 12 new 13 new 14 new 15 new 16 new 17 new 18 new 19 new 20
1 create table table1(pid INT, pname STRING, drug STRING, gender STRING, tot_amt INT) row format delimited fields
terminated by ',' stored as textfile;
2
3 load data local inpath '/home/ubuntu/data_301.txt' into table table1;
4 load data local inpath '/home/ubuntu/data_301.txt' into table table1;
5
6
7 show create table table1;
8
9
0 ======
1
2 create table table2(pid INT, pname STRING, drug STRING, gender STRING, tot_amt INT) row format delimited fields
terminated by ',' STORED AS orc;
3
4 insert overwrite table table2 select * from table1;
5
6 show create table table2;
7
8 select count(*) from table1 where tot_amt =110;
9 select count(*) from table2 where tot_amt =110;
0
```

# Table 1: Text Format (60lakh count )

```
essions View Split MultiEdit Tunneling Packages Settings Help
 2:13.56.16.244 (ubuntu) 3:13.56.16.244 (ubuntu)
'hdfs://localhost:50000/user/hive/warehouse/table2'
TBLPROPERTIES (
 'transient_lastDdlTime'='1657108076')
Time taken: 0.049 seconds, Fetched: 19 row(s)
hive> select count(*) from table2;
OK
6000000
Time taken: 0.4 seconds, Fetched: 1 row(s)
hive> select count(*) from table1;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine
(i.e. spark, tez) or using Hive 1.X releases.
Query ID = ubuntu_20220713105649_992f7438-37f2-47cf-87d0-6803e29effd
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
 set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
 set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
 set mapreduce.job.reduces=<number>
Starting Job = job_1657708926751_0003, Tracking URL = http://ip-172-31-5-114.us-west-1.compute.internal:8088/proxy/application_1657708926
751_0003/
Kill Command = /home/ubuntu/hadoop-2.9.1/bin/hadoop job -kill job_1657708926751_0003
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2022-07-13 10:56:56,977 Stage-1 map = 0%, reduce = 0%
2022-07-13 10:57:04,183 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 4.85 sec
2022-07-13 10:57:10,322 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 6.18 sec
MapReduce Total cumulative CPU time: 6 seconds 180 msec
Ended Job = job_1657708926751_0003
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Reduce: 1 Cumulative CPU: 6.18 sec HDFS Read: 238933789 HDFS Write: 107 SUCCESS
Total MapReduce CPU Time Spent: 6 seconds 180 msec
OK
6000000
Time taken: 22.29 seconds, Fetched: 1 row(s)
```

# Text File Format:



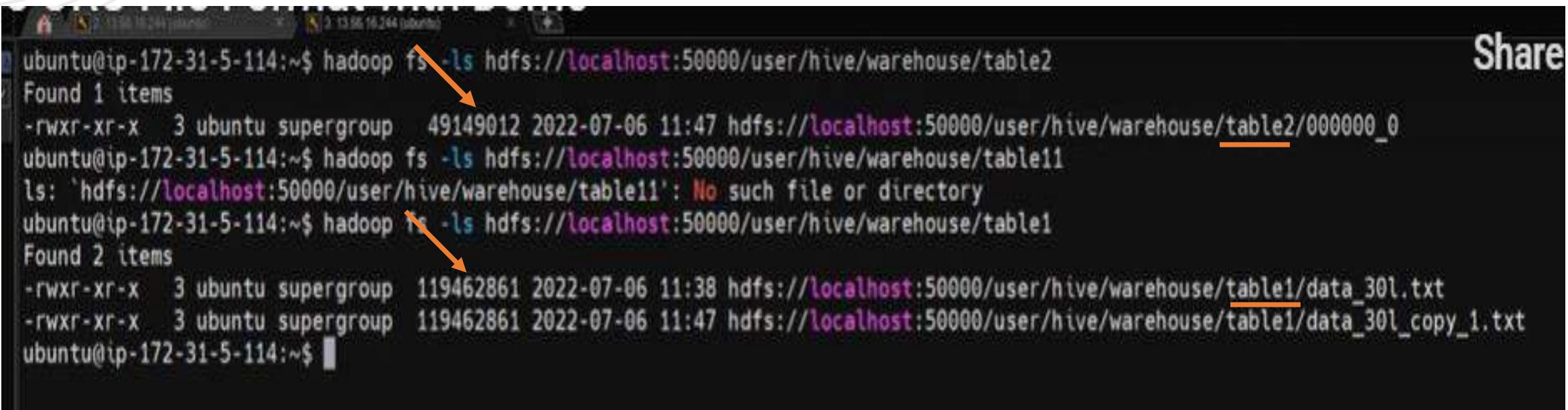
The screenshot shows a terminal window with two tabs open, both titled '13.66.16.244 (ubuntu)'. The terminal is running a Hive session. The user has run a 'select count(\*) from table1 where tot\_amt =110;' query, which returns a single row with a value of 6474. A red arrow points to this value. Below the query results, there is a warning message about Hive-on-MR being deprecated.

```
LOCATION
'hdfs://localhost:50000/user/hive/warehouse/table3'
TBLPROPERTIES (
 'transient_lastDdlTime'='1657108663')
Time taken: 0.208 seconds, Fetched: 19 row(s)
hive> select count(*) from table1 where tot_amt =110;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or using Hive 1.X releases.
Query ID = ubuntu_20220713104352_e49bb894-b0d5-4509-9a21-e0bf295533ea
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
 set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
 set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
 set mapreduce.job.reduces=<number>
Starting Job = job_1657708926751_0001, Tracking URL = http://ip-172-31-5-114.us-west-1.compute.internal:8088/proxy/application_1657708926751_0001/
Kill Command = /home/ubuntu/hadoop-2.9.1/bin/hadoop job -kill job_1657708926751_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2022-07-13 10:44:04,971 Stage-1 map = 0%, reduce = 0%
2022-07-13 10:44:13,362 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 6.46 sec
2022-07-13 10:44:20,606 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 8.42 sec
MapReduce Total cumulative CPU time: 8 seconds 420 msec
Ended Job = job_1657708926751_0001
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Reduce: 1 Cumulative CPU: 8.42 sec HDFS Read: 238934593 HDFS Write: 104 SUCCESS
Total MapReduce CPU Time Spent: 8 seconds 420 msec
OK
6474
Time taken: 29.108 seconds, Fetched: 1 row(s)
hive> select count(*) from table2 where tot_amt =110; Presentation title
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine
```

# ORC : Time Difference

```
OK
6474
Time taken: 29.108 seconds, Fetched: 1 row(s)
hive> select count(*) from table2 where tot_amt =110;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine
(i.e. spark, tez) or using Hive 1.X releases.
Query ID = ubuntu_20220713104557_950d5971-092e-4e66-b4c3-59e5ceb973dc
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
 set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
 set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
 set mapreduce.job.reduces=<number>
Starting Job = job_1657708926751_0002, Tracking URL = http://ip-172-31-5-114.us-west-1.compute.internal:8088/proxy/application_1657708926
751_0002/
Kill Command = /home/ubuntu/hadoop-2.9.1/bin/hadoop job -kill job_1657708926751_0002
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2022-07-13 10:46:03,565 Stage-1 map = 0%, reduce = 0%
2022-07-13 10:46:11,832 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 3.69 sec
2022-07-13 10:46:18,041 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 5.38 sec
MapReduce Total cumulative CPU time: 5 seconds 380 msec
Ended Job = job_1657708926751_0002
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Reduce: 1 Cumulative CPU: 5.38 sec HDFS Read: 9114117 HDFS Write: 104 SUCCESS
Total MapReduce CPU Time Spent: 5 seconds 380 msec
OK
6474
Time taken: 21.902 seconds, Fetched: 1 row(s)
hive>
hive> show create table table1;
OK
CREATE TABLE `table1`(
 `pid` int,
```

# Volume Compression of Text & ORC File Formats:



The screenshot shows a terminal window with the following command and output:

```
ubuntu@ip-172-31-5-114:~$ hadoop fs -ls hdfs://localhost:50000/user/hive/warehouse/table2
Found 1 items
-rwxr-xr-x 3 ubuntu supergroup 49149012 2022-07-06 11:47 hdfs://localhost:50000/user/hive/warehouse/table2/_0
ubuntu@ip-172-31-5-114:~$ hadoop fs -ls hdfs://localhost:50000/user/hive/warehouse/table11
ls: 'hdfs://localhost:50000/user/hive/warehouse/table11': No such file or directory
ubuntu@ip-172-31-5-114:~$ hadoop fs -ls hdfs://localhost:50000/user/hive/warehouse/table1
Found 2 items
-rwxr-xr-x 3 ubuntu supergroup 119462861 2022-07-06 11:38 hdfs://localhost:50000/user/hive/warehouse/table1/_0
-rwxr-xr-x 3 ubuntu supergroup 119462861 2022-07-06 11:47 hdfs://localhost:50000/user/hive/warehouse/table1/_1
```

Two orange arrows point from the text "Table 1" and "Table 2" below to the file names `_0` and `_1` respectively.

- $119462861 * 2 = 238,925,722$  (Text format) [Table 1]
- 49149012 (ORC Format) [Table 2]

# Apache Hive Command Line Options

| Hive Command Line Option | Description                                                                        |
|--------------------------|------------------------------------------------------------------------------------|
| -d,-define <key=value>   | Variable substitution to apply to Hive commands.<br><br>e.g. -d A=B or –define A=B |
| -e <quoted-query-string> | Execute SQL from command line.<br><br>e.g. -e 'select * from table1'               |
| -f <filename>            | Execute SQL from file. e.g. -f '/home/sql_file.sql'                                |
| -H,—help                 | Print Hive help information. Usually, print all command line options.              |

• SQL queries written in a file should be saved with **.hql extension**

• **Silent Mode:** without showing any progress or output information apart from the result set itself.

## Hive Command Line Options Usage Examples

### Execute query using hive command line options

```
$ hive -e 'select * from test';
```

### Execute query using hive command line options in silent mode

```
$ hive -S -e 'select * from test'
```

### Dump data to the file in silent mode

```
$hive -S -e 'select col from tab1' > a.txt
```

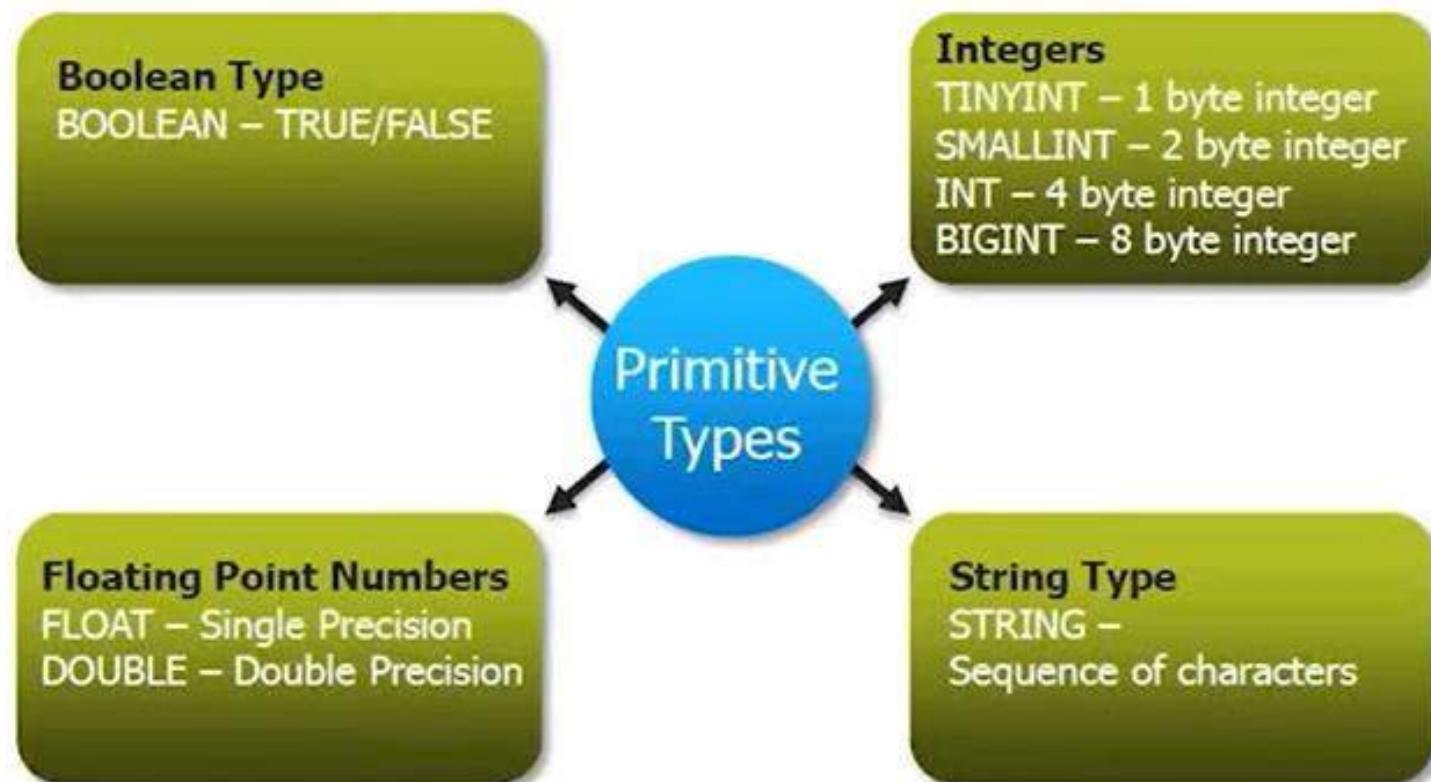
# HIVE V/S RDBMS

| SL.NO | Characteristics      | HIVE               | RDBMS                   |
|-------|----------------------|--------------------|-------------------------|
| 1     | Record Level Queries | No Update & Delete | Insert, Update & Delete |
| 2     | Transaction Support  | No                 | Yes                     |
| 3     | Latency              | Minutes or more    | In fraction of seconds  |
| 4     | Data Size            | Petabytes          | Terabytes               |
| 5     | Data Per Query       | Petabytes          | Gigabytes               |
| 6     | Query Language       | HiveQL             | SQL                     |
| 7     | Support JDBC/ODBC    | Limited            | Full                    |

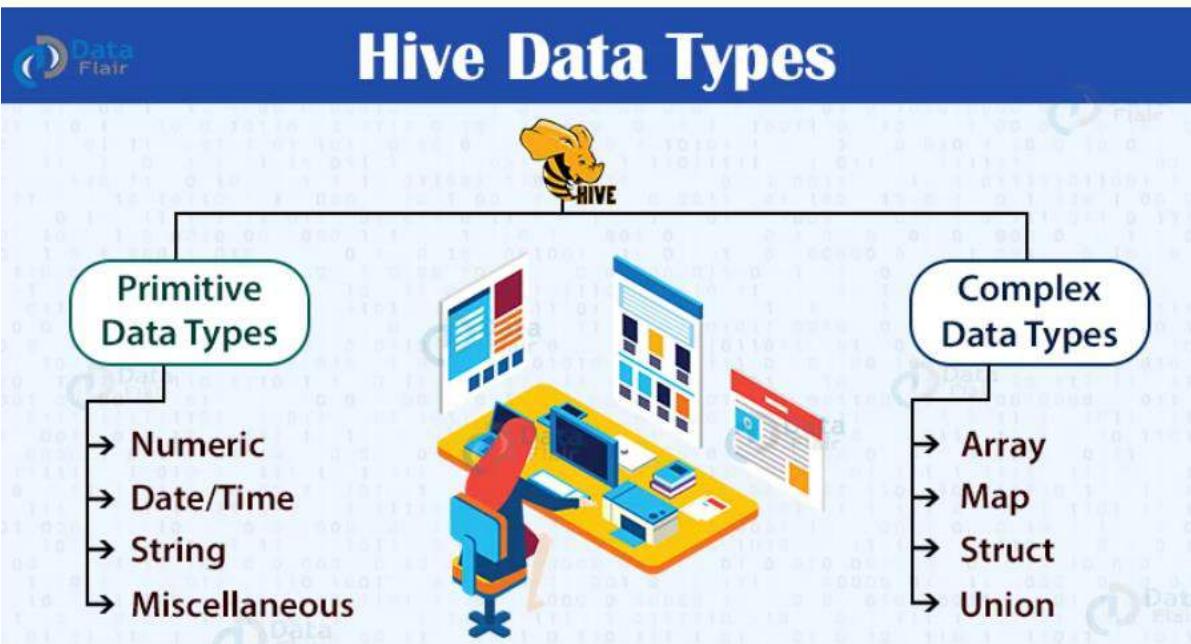
# RDBMS V/S HIVE

- ✓ **Schema on Read vs Schema on Write**
  - ✓ **Hive does not verifies the data when it is loaded**, but rather when a query is issued.
  - ✓ Schema on read makes for a **very fast initial load**, since the data does not have to be read, parsed and serialized to disk in the database's internal format. The load operation is just a file copy or move.
- ✓ **No Updates, Transactions and Indexes.**

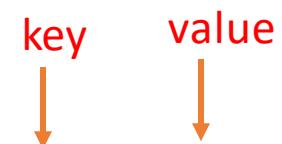
# Data Types - Hive



# Collection Data Types:



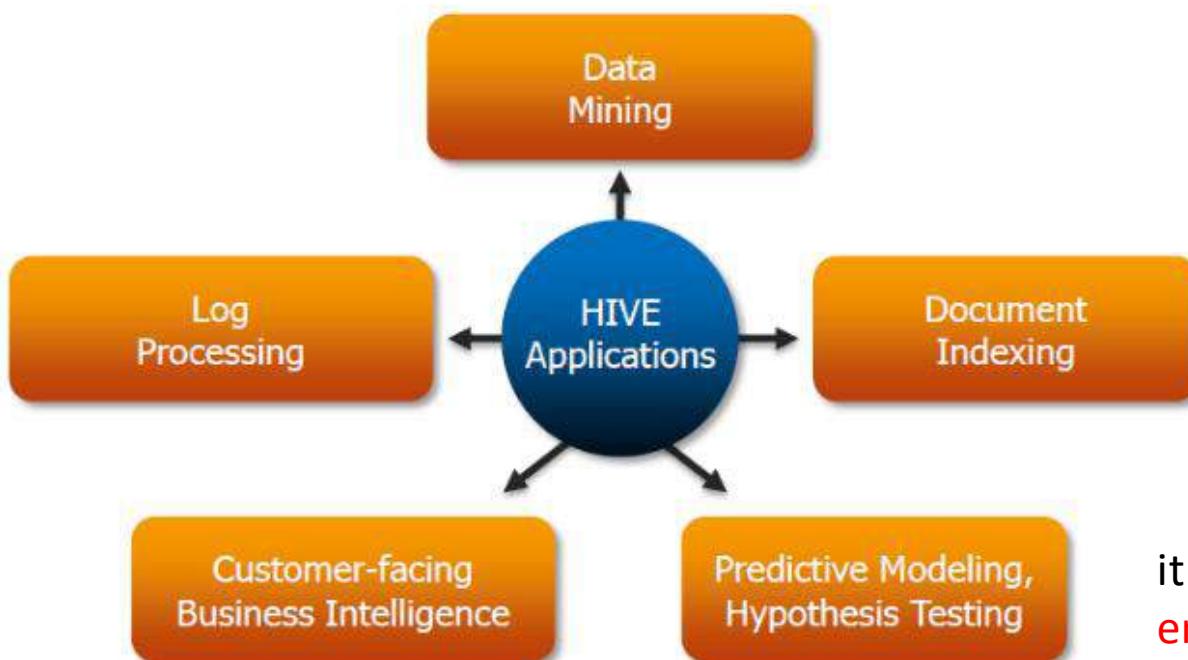
| Name   | Description                                                                                                                                   |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| STRUCT | Similar to 'C' struc, a collection of fields of different data types. An access to field uses dot notation.<br>For example, struct ('a', 'b') |
| MAP    | A collection of key-value pairs. Fields access using [] notation.<br>For example, map ('key1', 'a', 'key2', 'b')                              |
| ARRAY  | Ordered sequence of same types. Accesses to fields using array index.<br>For example, array ('a', 'b')                                        |

- **Map in Hive** is a collection of key-value pairs, where the fields are accessed using array notations of keys (e.g., ['key']).
- **map<primitive\_type, data\_type>**
- **Example:** 'first' -> 'John', 'last' -> 'Deo', represented as map('first', 'John', 'last', 'Deo'). Now 'John' can be accessed with map['first'].
- **STRUCT in Hive** is similar to the STRUCT in C language. It is a record type that encapsulates a set of named fields, which can be any primitive data type.
- We can access the elements in STRUCT type using DOT (.) notation.
- **STRUCT <col\_name : data\_type [ COMMENT col\_comment], ...>**
- Example: For a column c3 of type STRUCT {c1 INTEGER; c2 INTEGER}, the c1 field is accessed by the expression c3.c1.

# Built In Functions: Return Type, Syntax & Description

|        |                                           |                                                                                                                                                                                                            |
|--------|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BIGINT | round(double a)                           | Returns the rounded BIGINT (8 Byte integer) value of the 8 Byte double-precision floating point number a                                                                                                   |
| BIGINT | floor(double a)                           | Returns the maximum BIGINT value that is equal to or less than the double.                                                                                                                                 |
| BIGINT | ceil(double a)                            | Returns the minimum BIGINT value that is equal to or greater than the double.                                                                                                                              |
| double | rand(),<br>rand(int seed)                 | Returns a random number (double) that distributes uniformly from 0 to 1 and that changes in each row. Integer seed ensures that random number sequence is deterministic.                                   |
| string | concat(string str1, string str2, ...)     | Returns the string resulting from concatenating str1 with str2, ....                                                                                                                                       |
| string | substr(string str, int start)             | Returns the substring of str starting from a start position till the end of string str.                                                                                                                    |
| string | substr(string str, int start, int length) | Returns the substring of str starting from the start position with the given length.                                                                                                                       |
| string | upper(string str), ucase (string str)     | Returns the string resulting from converting all characters of str to upper case.                                                                                                                          |
| string | lower(string str), lcase(string str)      | Returns the string resulting from converting all characters of str to lower case.                                                                                                                          |
| string | trim(string str)                          | Returns the string resulting from trimming spaces from both ends. trim ('12A34 56') returns '12A3456'                                                                                                      |
| string | ltrim(string str);<br>rtrim(string str)   | Returns the string resulting from trimming spaces (only one end, left or right hand side or right-handside spaces trimmed). ltrim('12A34 56') returns '12A3456' and rtrim(' 12A34 56 ') returns '12A3456'. |

# Apache Hive – Applications



- Data Mining (If u want to identify Buying behaviour)
- Predictive Modeling (To enhance performance)
- Document Indexing
- Custom Facing UI

it can play a crucial role in the **data preparation and feature engineering stages** of the **predictive modeling pipeline**

# Apache Hive Use Case – Facebook:



# Facebook – Use Case

- **Objective – To store & process large amount of data.**
- **In 2007 , FB used “Hadoop” – To store and process the generated data. ETL (Via Python)**
- **But In Hadoop – Programming became difficult, as Hadoop default based is map reduce (java based), to execute small query larger codes need to be written (Structured Data type).**
- **As a solution, Hive was developed by FB.**
  
- FB Stats: RDBMS was not suitable
- Hadoop usage Started
- 500 B/Day
- 70k Queries /Day
- 300 Million Photos/Day

# Hive QL:

- HiveQL: **Querying the large dataset** which reside in the **HDFS** environment.
- HiveQL script commands enable **data definition, data manipulation and query processing**.
- Supports a large base of SQL users who are acquainted with SQL to “**Extract information from data warehouse**”

# HiveQL : Data Definition Language

HiveQL database commands for data definition for DBs and Tables are CREATE DATABASE, SHOW DATABASE (list of all DBs), CREATE SCHEMA, CREATE TABLE. Following are HiveQL commands which create a table:

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [<database name>.]
<table name>
[(<column name> <data type> [COMMENT <column comment>], ...)]
[COMMENT <table comment>]
[ROW FORMAT <row format>]
[STORED AS <file format>]
```

DELIMITED	Specifies a delimiter at the table level for structured fields. This is default. Syntax: FIELDS TERMINATED BY, LINES TERMINATED BY
SERDE	Stands for Serializer/Deserializer. SYNTAX: SERDE 'serde.class.name'

# Commands: DDL

- Commands : CREATE DATABASE, SHOW DATABASE, CREATE SCHEMA, CREATE TABLE

How do you create a database named toys\_companyDB and table named toys\_tbl?

## SOLUTION

```
$HIVE_HOME/binhive - service cli
hive>set hive.cli.print.current.db=true;
hive> CREATE DATABASE toys_companyDB
hive>USE toys_companyDB
hive (toys_companyDB)> CREATE TABLE toys_tbl (
>puzzle_code STRING,
>pieces SMALLINT
>cost FLOAT);
hive (toys_company)> quit;
&ls/home/binadmin/Hive/warehouse/toys_companyDB.db
```

# Partition: Reducing Query Processing Time

Assume that following file contains toys\_tbl.

```
/table/toy_tbl/file1
Category, id, name, price
Toy_Airplane, 10725, Lost Temple, 1.25
Toy_Airplane, 31047, Propeller Plane, 2.10
Toy_Airplane, 31049, Twin Spin Helicopter, 3.45
```

```
Toy_Train, 31054, Blue Express, 4.25
Toy_Train, 10254, Winter Holiday Toy_Train, 2.75
```

A table *toy\_tbl* contains many values for categories of toys. Query is required to identify all *toy\_airplane* fields. Give reasons why partitioning reduces query processing time.

## SOLUTION

Here, a table named *toy\_tbl* contains several toy details (category, id, name and price). Suppose it is required to identify all the airplanes. A query searches the whole table for the required information. However, if a partition is created on the *toy\_tbl*, based on category and stores it in a separate file, then it will reduce the query processing time.

Let the data partitions into two files, file 2 and file 3, using category.

```
/table/toys/toy_airplane/file2
toy_airplane, 10725, Lost Temple, TP, 1.25
toy_airplane, 31047, Propeller Plane, 2.10
toy_airplane, 31049, Lost Temple, 3.45


```

# Partition : Advantages & Limitations

## Advantages of Partition

1. Distributes execution load horizontally.
2. **Query response time becomes faster** when processing a small part of the data instead of searching the entire dataset.

## Limitations of Partition

1. Creating a large number of partitions in a table leads to a large number of files and directories in HDFS, which is an overhead to NameNode, since it must keep all metadata for the file system in memory only.
2. Partitions may optimize some queries based on Where clauses, but they may be less responsive for other important queries on grouping clauses.
3. A large number of **partitions will lead to a large number of tasks** (which will run in separate JVM) in each MapReduce job, thus creating a lot of overhead in maintaining JVM start up and tear down (A separate task will be used for each file). The overhead of JVM start up and tear down can exceed the actual processing time in the worst case.

# Bucketing:

A table `toy_tbl` contains many values for categories of toys. Assume the number of buckets to be created = 5. Assume a table for `Toy_Airplane` of product code 10725.

1. How will the bucketing enforce?
2. How will the bucketed table partition `toy_airplane_10725` create five buckets?
3. How will the bucket column load into `toy_tbl`?
4. How will the bucket data display?

## SOLUTION

#Enforce bucketing

```
set hive.enforce.bucketing=true;
```

#Create bucketed Table for `toy_airplane` of product code 10725 and create cluster of 5 buckets

```
CREATE TABLE IF NOT EXISTS
toy_airplane_10725(ProductCategory STRING,
ProductId INT, ProductName STRING, PrdocutMfgDate
YYYY-MM-DD, ProductPrice_US$ FLOAT) CLUSTERED BY
(Price) into 5 buckets;
```

# Load data to bucketed table.

```
FROM toy_airplane_10725 INSERT OVERWRITE TABLE
toy_tbl SELECT ProductCategory, ProductId,
ProductName, PrdocutMfgDate, ProductPrice;
```

- To display the contents for `Price_US$` selected for the `ProductId` from the second bucket.

Pres

```
SELECT DISTINCT ProductId FROM toy_tbl_buckets
TABLE FOR 10725(BUCKET 2 OUT OF 5 ON Price_US$);
```

## Arithmetic Operators in Hive

Operators	Description
A + B	This is used to add A and B.
A - B	This is used to subtract B from A.
A * B	This is used to multiply A and B.
A / B	This is used to divide A and B and returns the quotient of the operands.
A % B	This returns the remainder of A / B.
A   B	This is used to determine the bitwise OR of A and B.
A & B	This is used to determine the bitwise AND of A and B.
A ^ B	This is used to determine the bitwise XOR of A and B.
~A	This is used to determine the bitwise NOT of A.

```
hive> select id, name, salary + 50 from employee;
```

```
hive> select id, name, salary - 50 from employee;
```

```
hive> select id, name, (salary * 10) /100 from employee;
```

## Relational Operators in Hive

Operator	Description
A=B	It returns true if A equals B, otherwise false.
A <> B, A !=B	It returns null if A or B is null; true if A is not equal to B, otherwise false.
A<B	It returns null if A or B is null; true if A is less than B, otherwise false.
A>B	It returns null if A or B is null; true if A is greater than B, otherwise false.
A<=B	It returns null if A or B is null; true if A is less than or equal to B, otherwise false.
A>=B	It returns null if A or B is null; true if A is greater than or equal to B, otherwise false.
A IS NULL	It returns true if A evaluates to null, otherwise false.
A IS NOT NULL	It returns false if A evaluates to null, otherwise true.

```
hive> select * from employee where salary >= 25000;
```

```
hive> select * from employee where salary < 25000;
```

# Aggregate Functions in Hive

In Hive, the aggregate function returns a single value resulting from computation over many rows. Let's see some commonly used aggregate functions: -

Return Type	Operator	Description
BIGINT	count(*)	It returns the count of the number of rows present in the file.
DOUBLE	sum(col)	It returns the sum of values.
DOUBLE	sum(DISTINCT col)	It returns the sum of distinct values.
DOUBLE	avg(col)	It returns the average of values.
DOUBLE	avg(DISTINCT col)	It returns the average of distinct values.
DOUBLE	min(col)	It compares the values and returns the minimum one form it.
DOUBLE	max(col)	It compares the values and returns the maximum one form it.

# HIVE V/S HBASE

Hive	HBase
Hive is a query engine.	Hbase is data storage mainly for unstructured data.
Hive allows most of the SQL queries.	HBase does not allow SQL queries.
Hive is mainly used for batch processing.	Hbase is mainly used for transactional processing.
Hive is not real-time processing.	HBase is real-time processing.
Hive is only used for analytical queries.	HBase is used for real-time querying.
Hive runs on the top of MapReduce.	HBase runs on the top of HDFS (Hadoop distributed file system).
Hive is not a full database. It is a data warehouse framework	HBase supports the NoSQL database.
Hive provides SQL features to Spark/Hadoop data.	HBase is used to store and process Hadoop data in real-time.
Hive has a schema model.	HBase is free from the schema model.
Hive is made for high latency operations.	HBase is made for low-level latency operations.
Hive is not suited for real-time querying.	HBase is used for real-time querying of Big Data.



A dark green background featuring various tropical leaves, including palm fronds and smaller greenery, creating a lush, jungle-like atmosphere.

Thank you

# Part 8 :

# RHA DOOP

## R:

R is an open-source programming language that is extensively used for **statistical and graphical analysis**.

One major quality of R's is that it produces well-designed **quality plots** with greater ease, including mathematical **symbols and formulae where needed**.

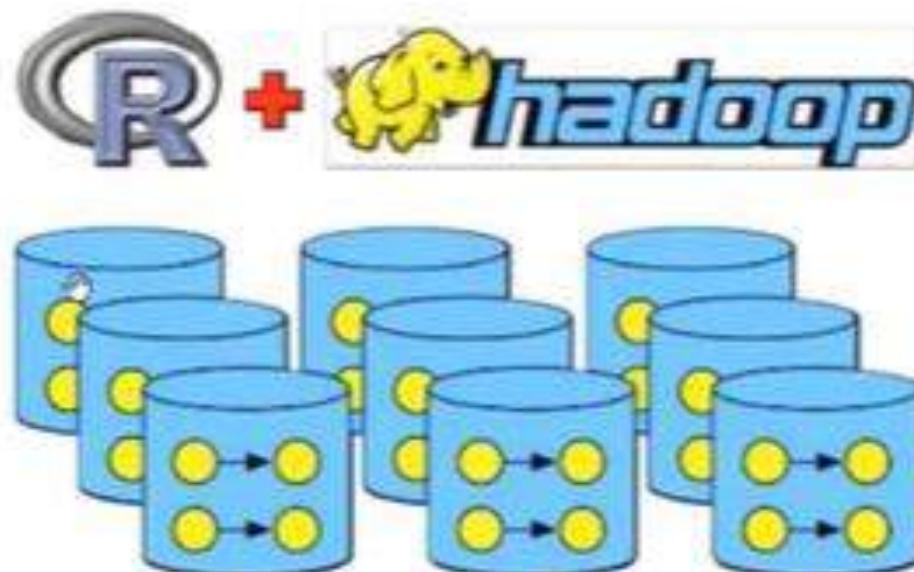
Some reasons for which R is considered the best fit for data analytics :

- A robust collection of **packages**
- Powerful **data visualization** techniques
- Commendable **Statistical and graphical** programming features
- Object-oriented programming language
- It has a wide smart **collection of operators** for calculations of arrays, particular matrices, etc
- Graphical representation capability on display or on hard copy.

# R:

R is very powerful for statistical analysis, However:

- ✓ All the calculations are performed by loading entire data in RAM
- ✓ Scaling up RAM has an upper limit
- ✓ Hadoop framework allows parallel processing of massive amount of data
- ✓ Using R with Hadoop facilitates horizontal scalability of statistical calculations

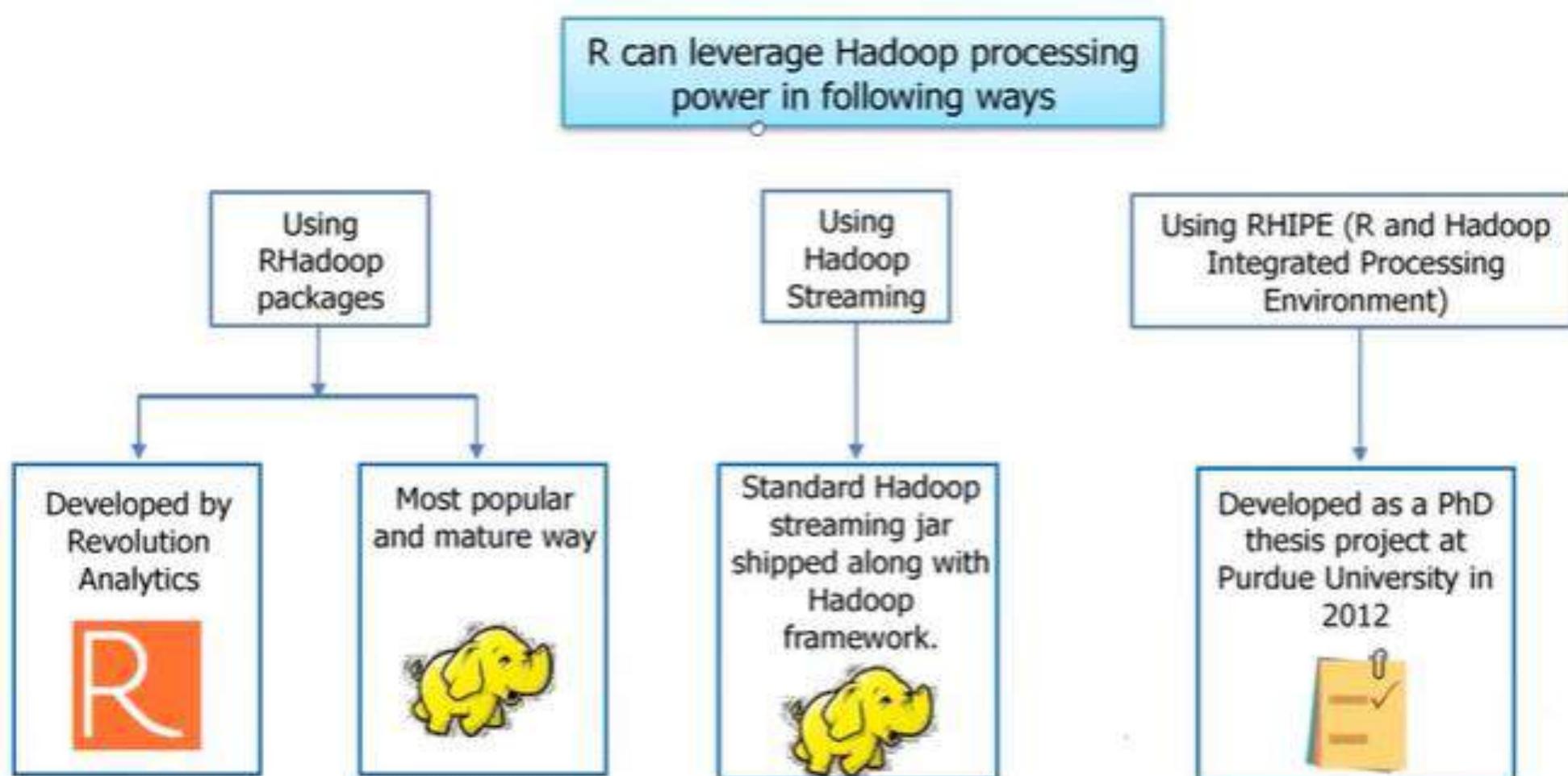


- **Rhadoop:** Collection of 3 packages for performing large data operations with an R environment.

# Hadoop & R

- Hadoop for R
  - Datasets that do not fit in memory but can be naturally partitioned
  - Harness well known infrastructure for embarrassingly parallel tasks
- R for Hadoop
  - Very large number of implementations of different functions and libraries (machine learning, text processing etc)
  - Native C implementation – very fast number crunching, matrix processing
  - All in-memory calculations

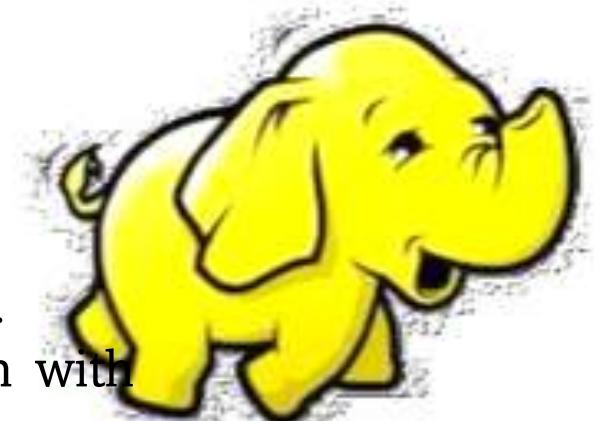
# Integrate R and Hadoop: Different Ways



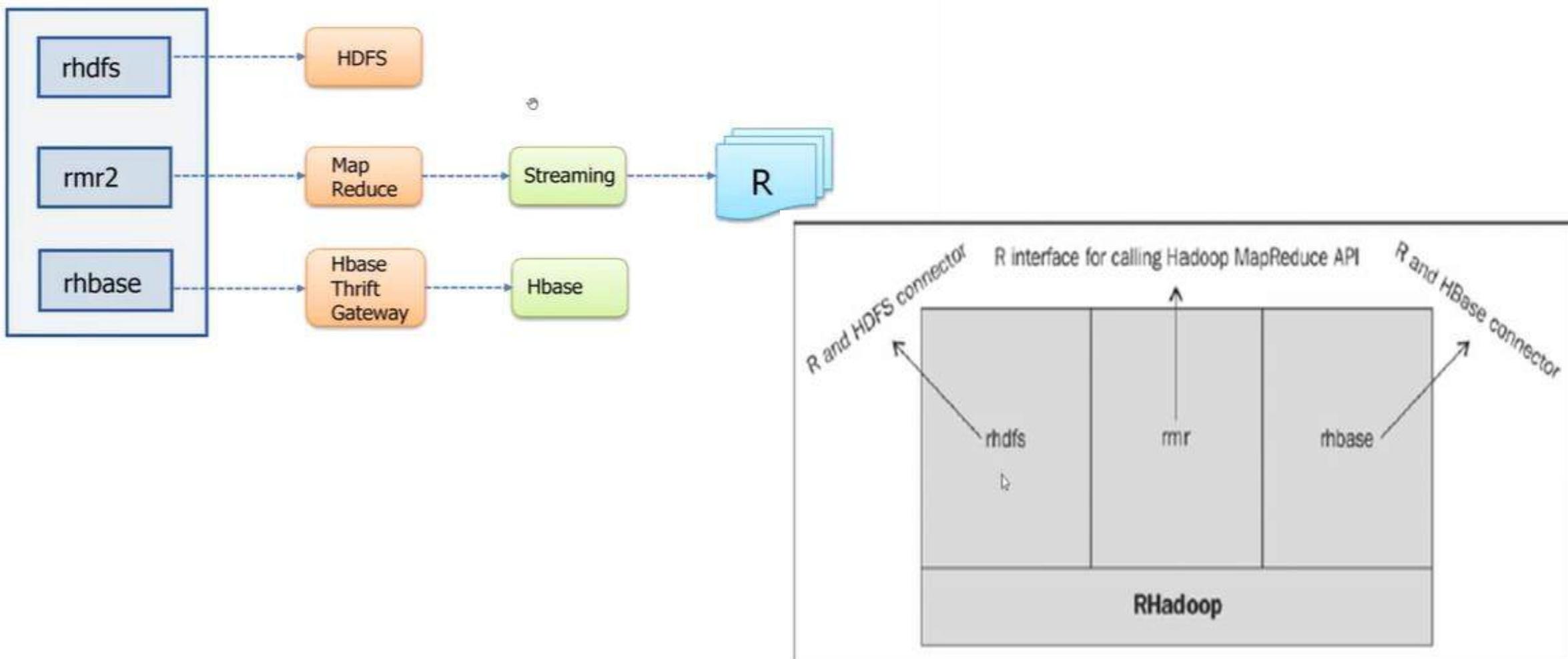
# Exploring RHadoop

- ✓ Development lead by Revolution Analytics. Available on github:  
<https://github.com/RevolutionAnalytics/RHadoop>
- ✓ Mostly implemented in native R. Some pieces written in C++ and Java
- ✓ Contains the following packages:
  - **rmr**
    - Interface for running map/reduce jobs via Hadoop Streaming in R
  - **rbase**
    - Interface to read/write data to/from HBase tables
  - **rhdfs**
    - Provides file manipulation capabilities for HDFS

- rmr: Helps to execute the Mapping & Reducing codes in R
- rhdfs: Provides file management capabilities by integrating with HDFS.
- rbase: provides R database management capability with integration with Hbase.



# RHadoop Architecture



# rmr2



rmr2

- Enables writing MapReduce jobs using R
- Ability to parallelize algorithms
- Ability to use big data sets without needing to sample data
- Mapreduce(input, output, map, reduce, ...)
- Reduces takes a key and a collection of values which could be vector, list, data frame or matrix
- 2.2.1 adds support for Windows (using HDP)

# Functionalities of rmr:

- ✓ **Convenience**

- keyval(): used in generating output from input formatters, mappers, and reducers

- ✓ **Input/Output**

- to.dfs()
  - from.dfs()
  - make.input.format()
  - make.output.format()

- ✓ **Job Execution**

- mapreduce(): submits job, returns a HDFS path pointing to output data



# rmr Advantages:

## ✓ Well designed API

- User code needs only the manipulation of R objects (data frames, lists, vectors, strings, etc.)

- It provides a **convenient interface for R users** to write **MapReduce** programs and execute them on Hadoop clusters **without needing to write Java code directly.**

## ✓ Flexible I/O subsystem

- Supports common input formats like CSV
- Provides control for input parsing (buffered, line-by-line w/o having to interact with stdin and stdout directly)

- users have **full control over how input data is processed and transformed into key-value pairs** for further processing.
- Users can define **custom output formats** or select from predefined formats like text or sequence file.

## ✓ Capability to construct chains of mapreduce jobs

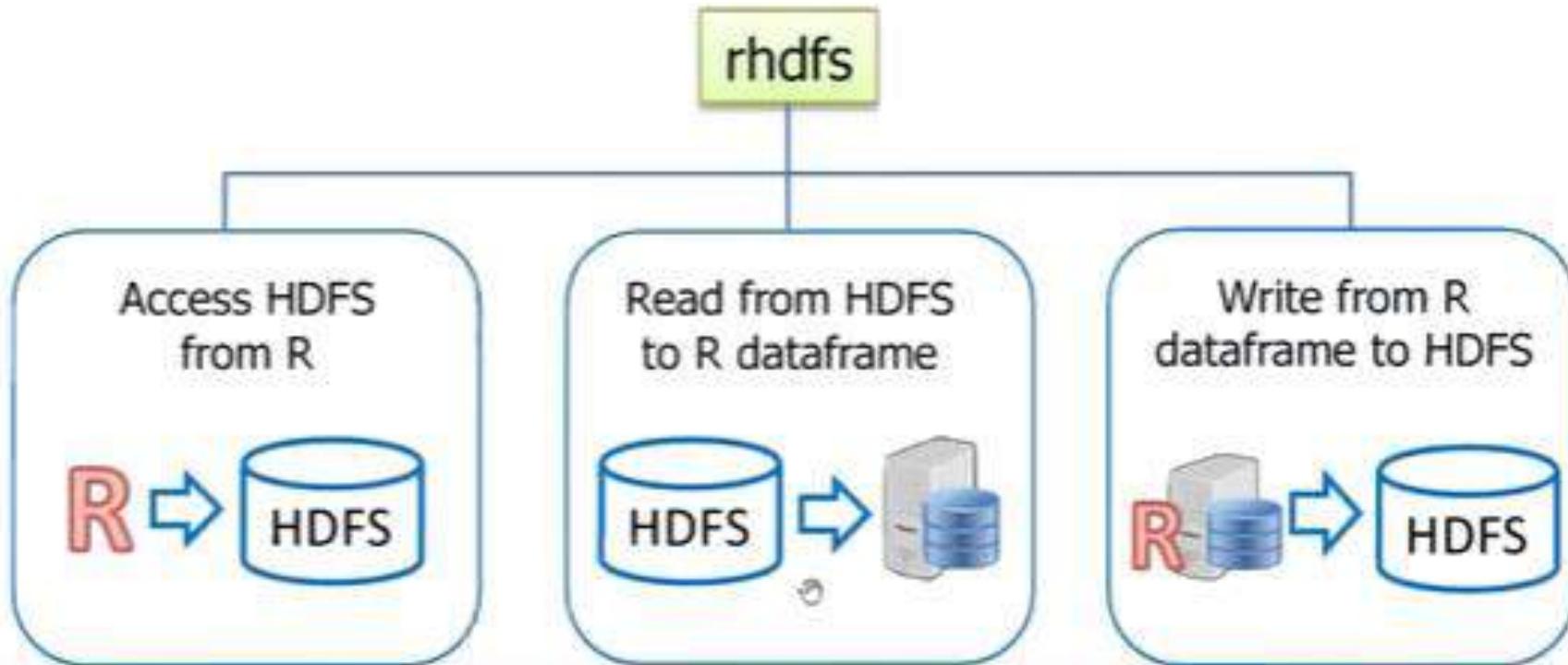
- Result of mapreduce() is simply the HDFS path of job's output

- MapReduce job will be executed **locally** within the R environment on the same machine where R is running

## ✓ Local backend for quick prototyping

- The "local backend" refers to **running MapReduce jobs on a single machine**, utilizing the **local file system** instead of Hadoop's distributed file system (HDFS) and the **local computing resources** instead of a Hadoop cluster. This setup is primarily used for **development, testing, and prototyping purposes.**

# rhdfs



# rhdfs

- ✓ Hadoop CLI Commands & rhdfs equivalent
- ✓ hadoop fs -ls /
  - hdfs.ls("/")
- ✓ hadoop fs -mkdir /user/rhdfs/ppt
  - hdfs.mkdir("/user/rhdfs/ppt")
- ✓ hadoop fs -put 1.txt /user/rhdfs/ppt/
  - localData <- system.file(file.path("unitTestData", "1.txt"), package="rhdfs")
  - hdfs.put(localData, "/user/rhdfs/ppt/1.txt")
- ✓ hadoop fs -get /user/rhdfs/ppt/1.txt 1.txt
  - hdfs.get("/user/rhdfs/ppt/1.txt","test")
- ✓ hadoop fs -rm /user/rhdfs/ppt/1.txt
  - hdfs.delete("/user/rhdfs/ppt/1.txt")

# Functionalities in HDFS

## ✓ File manipulations

- hdfs.copy, hdfs.move, hdfs.rename,
- hdfs.delete, hdfs.rm, hdfs.del,
- hdfs.chown, hdfs.put, hdfs.get

## ✓ File read/write

- hdfs.file, hdfs.write, hdfs.read,
- hdfs.close, hdfs.flush, hdfs.seek, hdfs.tell,
- hdfs.line.reader, hdfs.read.text.file

## ✓ Directory

- hdfs.dircreate, hdfs.mkdir

## ✓ Utility

- hdfs.ls, hdfs.list.files, hdfs.file.info, hdfs.exists

## ✓ Initialization

- hdfs.init and hdfs.defaults

`hdfs.copy`, `hdfs.move`, `hdfs.rename`, `hdfs.delete`, `hdfs.rm`, `hdfs.del`,  
`hdfs.chown`, `hdfs.put`, `hdfs.get`

- File Read/Write

`hdfs.file`, `hdfs.write`, `hdfs.close`, `hdfs.flush`, `hdfs.read`, `hdfs.seek`,  
`hdfs.tell`, `hdfs.line.reader`, `hdfs.read.text.file`

- Directory

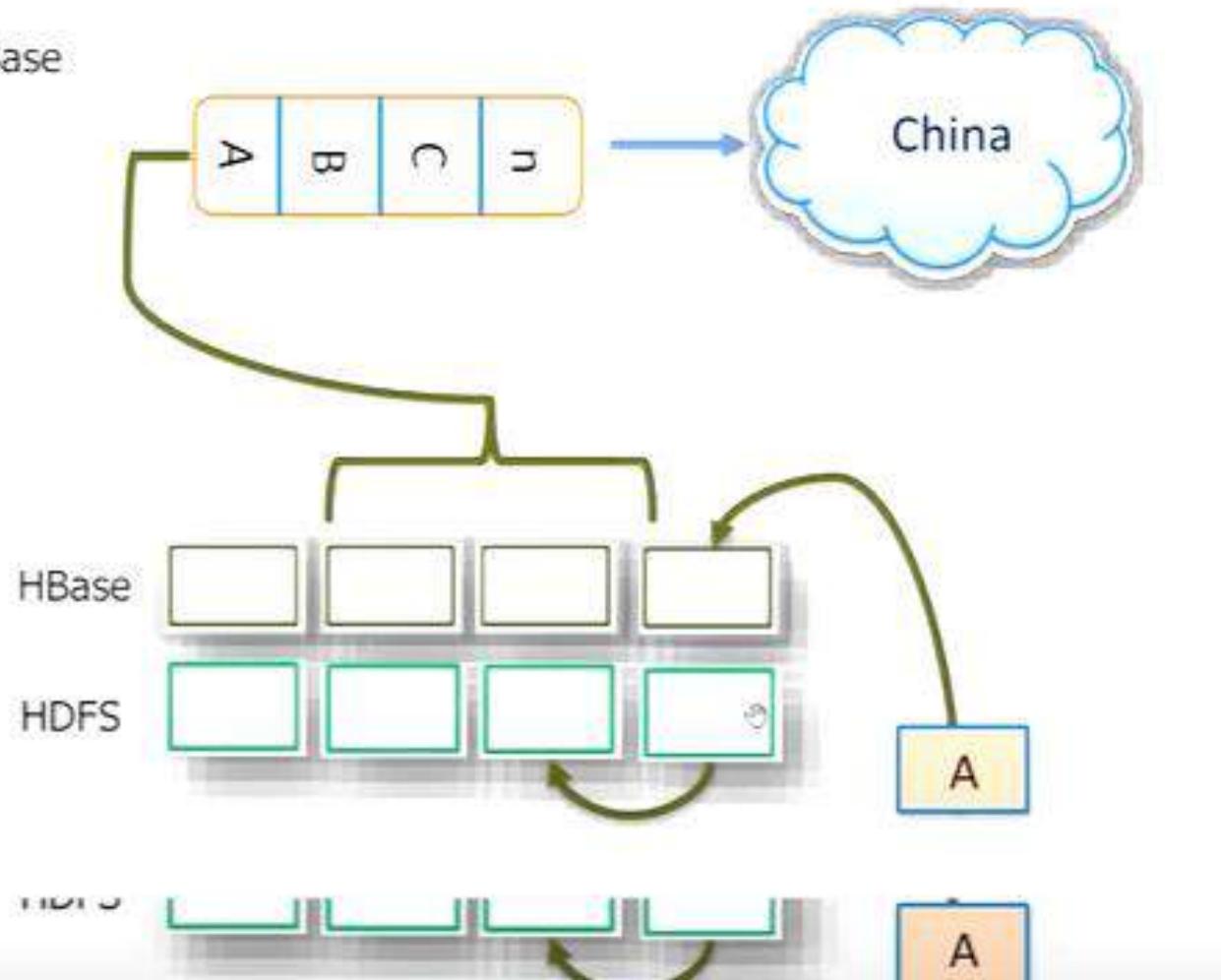
`hdfs.dircreate`, `hdfs.mkdir`

- Utility

`hdfs.ls`, `hdfs.list.files`, `hdfs.file.info`, `hdfs.exists`

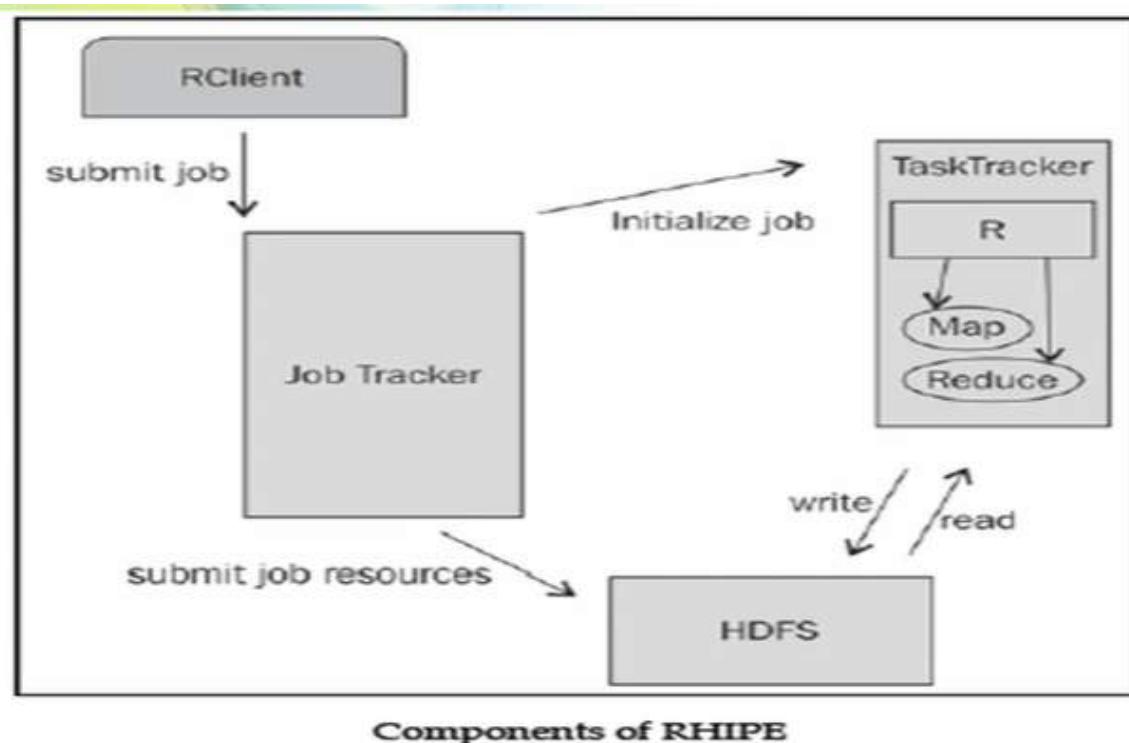
# rbase

- ✓ Access and change data within HBase
- ✓ Uses Thrift API
- ✓ Command Examples
  - hb.new.table
  - hb.insert
  - hb.scan.ex
  - hb.scan





# RHIPE:



- RHIPE: Stands for R and Hadoop Integrated Programming Environment
- RHIPE involves working with R and Hadoop Integrated Programming Environment
- **R client**: submits Query (Job)to job tracker.
- **Job Tracker** : Main coordinator – Responsible to allocating job/assign task/job to task tracker.
- **Task Tracker** : May be present in same location or different nodes.
- **Task Tracker** : Perform Map Reduce - Responsible for “Processing of Data”.
- Parallel to distributed computing.
- **HDFS: During Processing** : It can read and write data from HDFS

## RHIPE Components:

### **RClient:**

RClient is an R application that calls the JobTracker to execute the job with an indication of several MapReduce job resources such as Mapper, Reducer, input format, output format, input file, output file, and other several parameters that can handle the MapReduce jobs with RClient.

### **• JobTracker:**

A JobTracker is the master node of the Hadoop Map Reduce operations for initializing and monitoring the MapReduce jobs over the Hadoop cluster..

## RHIPE Components:

### **Task Tracker:**

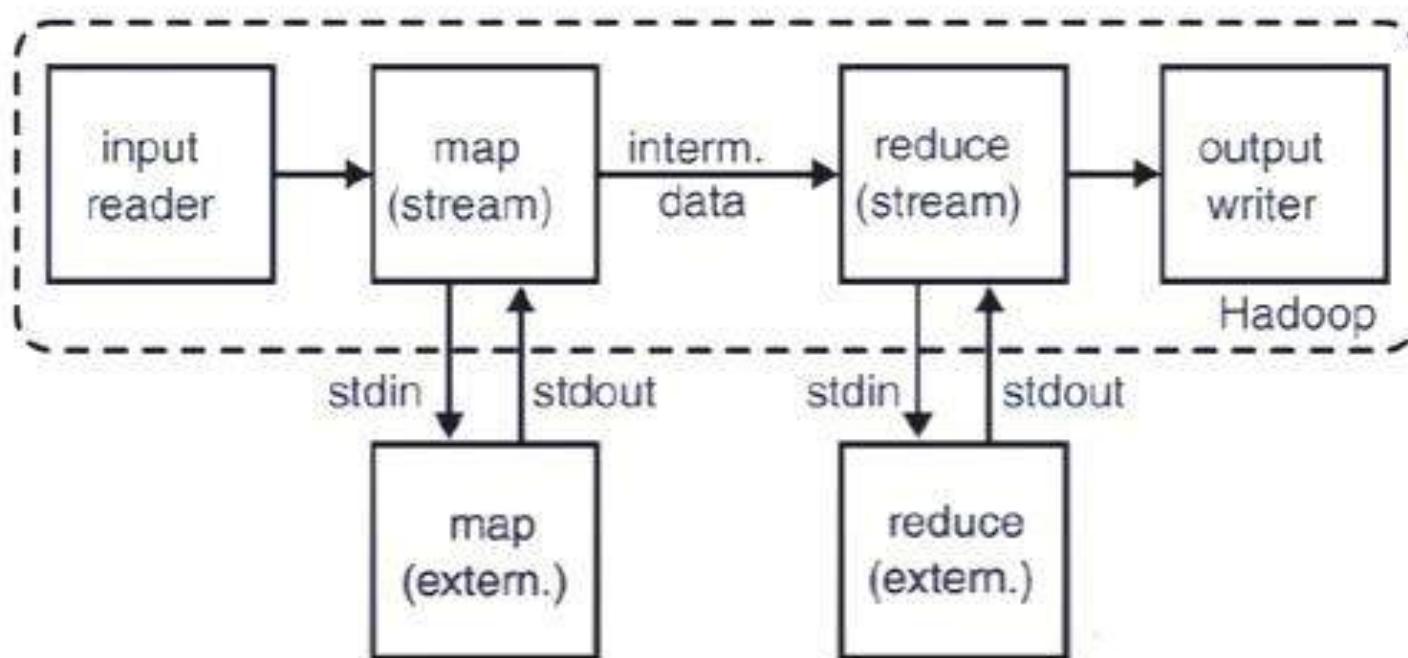
Task Tracker is a slave node in the Hadoop cluster. It executes the Map Reduce jobs as per the orders given by Job Tracker, retrieve the input data chunks, and run R-specific Mapper and Reducer over it. Finally, the output will be written on the HDFS directory.

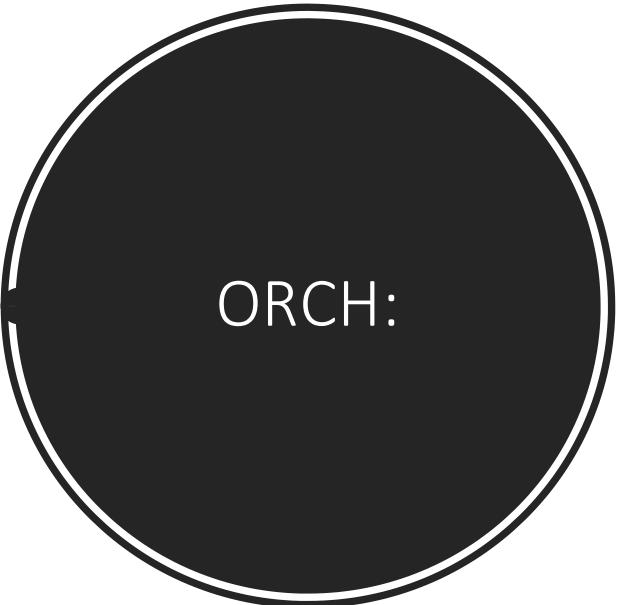
### **• HDFS:**

HDFS is a file system distributed over Hadoop clusters with several data nodes. It provides data services for various data operations.

# Hadoop Streaming:

- ✓ A utility that allows one to `run map/reduce` jobs on the cluster with any `executable`; e.g. shell scripts, Perl, Python, etc
- ✓ Executable reads input from `STDIN` and writes output to `STDOUT`
- ✓ Maps input line data from `STDIN` to `(key, value)` pairs in `STDOUT`, in our case





ORCH:

- ORCH is known as Oracle R Connector.
- It helps in accessing the Hadoop cluster with the help of R.
- It allows us to manipulate the data residing in the Hadoop Distributed File System.

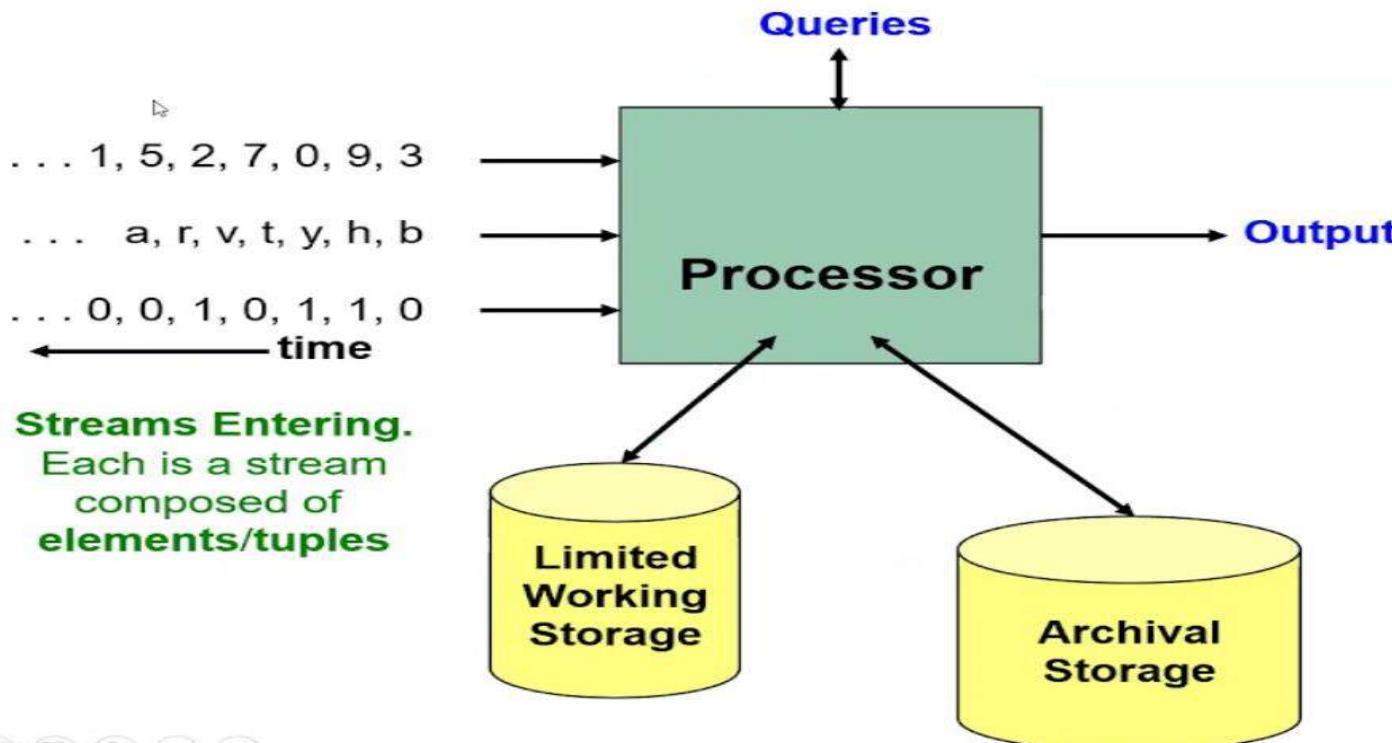
# MINING DATA STREAMS

## Data Stream ?

- Stream in English “a continuous flow”
- In computing
  - transmit or receive (data, especially video and audio material) over the Internet as a steady, continuous flow.

- | DBMS                                                            | versus | DSMS                                                      |
|-----------------------------------------------------------------|--------|-----------------------------------------------------------|
| ■ One-time queries                                              |        | ■ Continuous queries                                      |
| ■ Random access                                                 |        | ■ Sequential access                                       |
| ■ “Unbounded” disk store                                        |        | ■ Bounded main memory                                     |
| ■ Only current state matters                                    |        | ■ Historical data is important                            |
| ■ No real-time services                                         |        | ■ Real-time requirements                                  |
| ■ Relatively low update rate                                    |        | ■ Possibly multi-GB arrival rate                          |
| ■ Data at any granularity                                       |        | ■ Data at fine granularity                                |
| ■ Assume precise data                                           |        | ■ Data stale/imprecise                                    |
| ■ Access plan determined by query processor, physical DB design |        | ■ Unpredictable/variable data arrival and characteristics |

# General Stream Processing Model



## Challenges in Working with Streaming Data

- A storage layer
- A processing layer

Needs plan for scalability, data durability, and fault tolerance in both the storage and processing layers.

# Stream Processing

Parameter	Stream processing
Data scope	Queries or processing over data within a rolling time window, or on just the most recent data record.
Data size	Individual records or micro batches consisting of a few records.
Performance	Requires latency in the order of seconds or milliseconds.
Analyses	Simple response functions, aggregates, and rolling metrics.

## Streaming Data Examples

- Sensors in transportation vehicles, industrial equipment, and farm machinery.
- A financial institution
- A real-estate website
- A solar power company
- A media publisher
- An online gaming company

# Types of queries one wants answer on a data stream

## – Filtering a data stream

- Select elements with property  $x$  from the stream

## – Counting distinct elements

- Number of distinct elements in the last  $k$  elements of the stream

## – Estimating moments

- Estimate avg./std. dev. of last  $k$  elements

## – Finding frequent elements

## Applications (1)

### • Mining query streams

- Google wants to know what queries are more frequent today than yesterday

### • Mining click streams

- Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour

### • Mining social network news feeds

- E.g., look for trending topics on Twitter, Facebook

## STREAM DATA PROCESSING METHODS

- ❖ Random sampling (but without knowing the total length in advance)

- ✓ Reservoir sampling: maintain a set of  $s$  candidates in the reservoir, which form a true random sample of the element seen so far in the stream. As the data stream flow, every new element has a certain probability ( $s/N$ ) of replacing an old element in the reservoir.

- ❖ Sliding windows

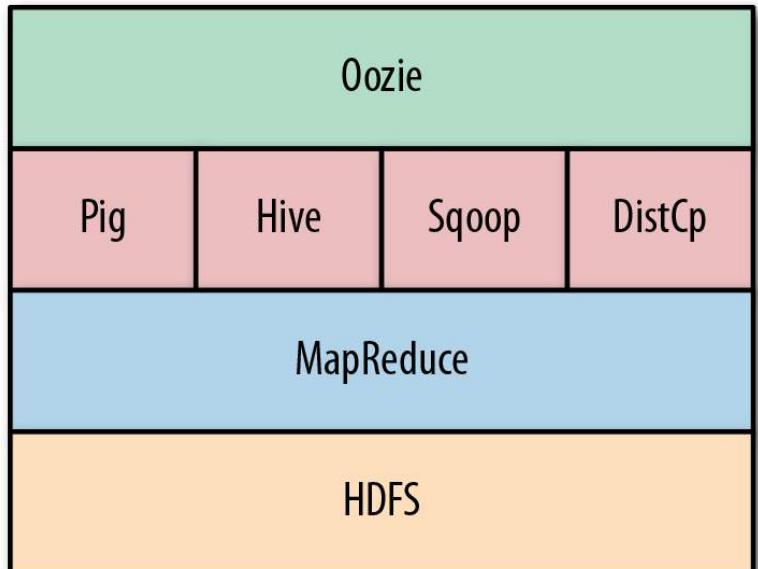
- ✓ Make decisions based only on **recent** data of sliding window size  $w$
  - ✓ An element arriving at time  $t$  expires at time  $t + w$

- ❖ Histograms

- ✓ Approximate the frequency distribution of element values in a stream
  - ✓ Partition data into a set of contiguous buckets
  - ✓ Equal-width (equal value range for buckets) vs. V-optimal (minimizing frequency variance within each bucket)

# Apache OOZIE

- Apache Oozie is a workflow director designed to run and manage multiple related Apache Jobs.
- For instance, complete data input and analysis may require several related Hadoop jobs to be run as a workflow in which the output of one job may input to the successive job.
- Oozie is designed to construct and manage these workflows.
- Oozie is not a substitute for YARN scheduler.
- YARN manages the resources of individual jobs and Oozie provides a mechanism to connect and control Hadoop jobs on



# Apache OOZIE

Oozie workflow jobs are represented using **Directed Acyclic Graphs**.

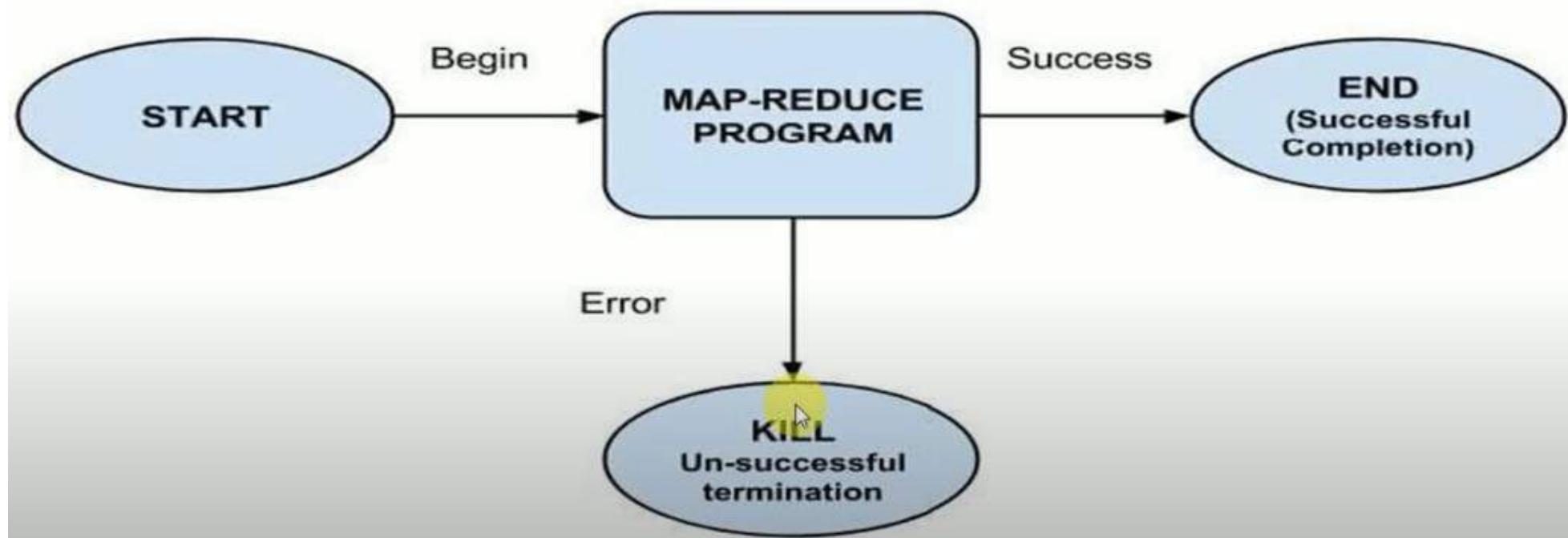
Oozie Supports three types of Jobs:

- **Workflow engine:** Responsibility of a workflow engine is to store and run workflows composed of Hadoop jobs e.g., MapReduce, Pig, Hive.
- **Coordinator engine:** It runs workflow jobs based on predefined schedules and availability of data.
- **Bundle:** Higher level abstraction that will batch a set of coordinator jobs

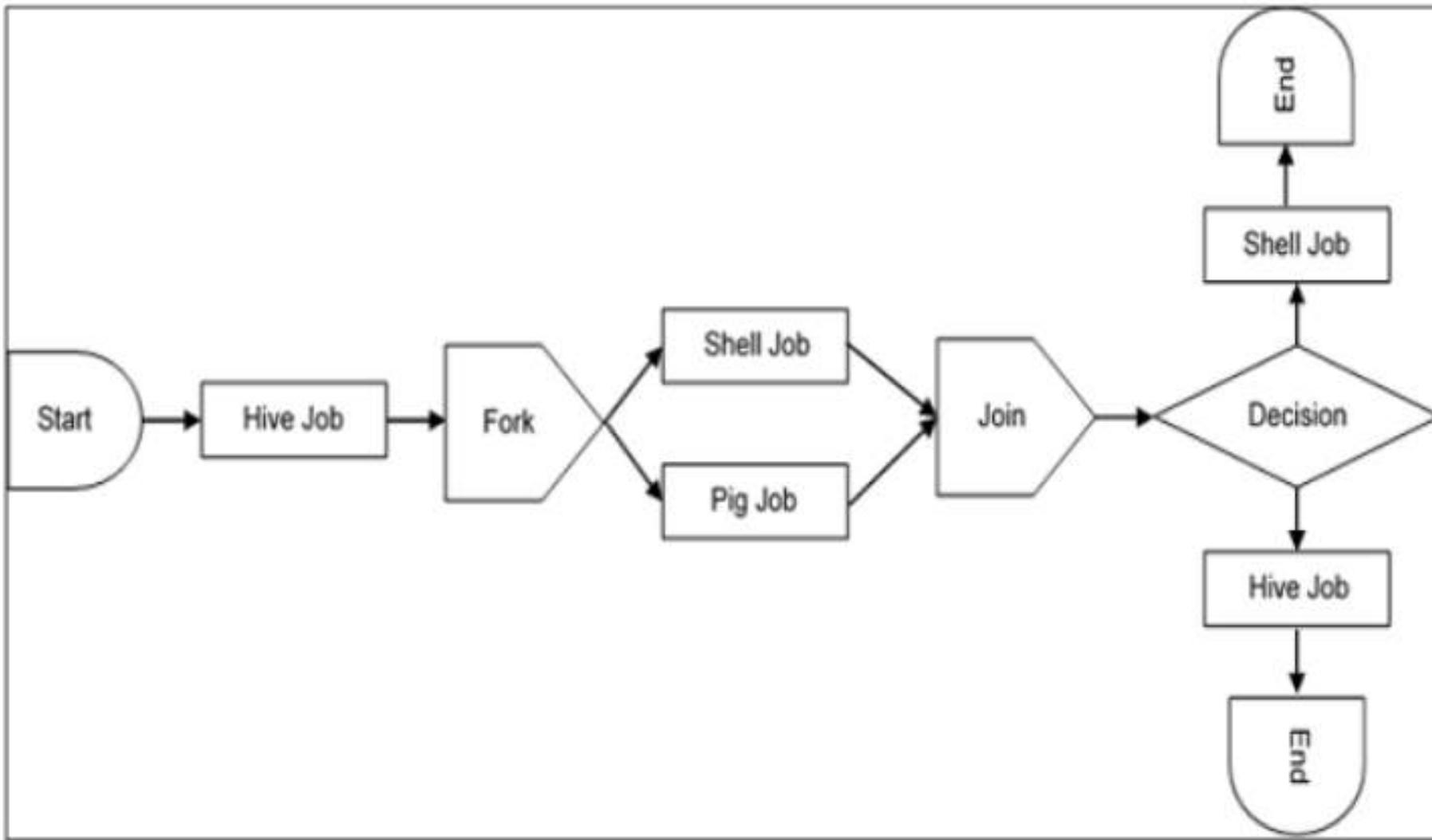
# Apache Oozie: Types of Nodes

- **Start and end nodes:** define the beginning and the end of the workflow. They include optional **fail** nodes.
- **Action Nodes:** are where the actual processing tasks are defined. When an action node finishes, the remote systems notify Oozie and the next node in the workflow is executed. Action nodes can also includes HDFS commands.
- **Fork/Join nodes:** enable parallel execution of tasks in the workflow. The fork node enables two or more tasks to run at the same time. A join node represents a rendezvous point that must wait until all forked tasks complete.
- **Control flow nodes:** enable decisions to be made about the previous task. Control decisions are based on the results of the previous actions. Decision nodes are essentially switch-case statement that use JSP EL (Java Server Pages – Expression Language) that evaluate to either true or false.

# Apache OOZIE: workflow



# Apache Oozie: Complex workflow



THANK  
YOU.

\* ALL THE BEST \*



# BIG DATA ANALYTICS

Part – 8

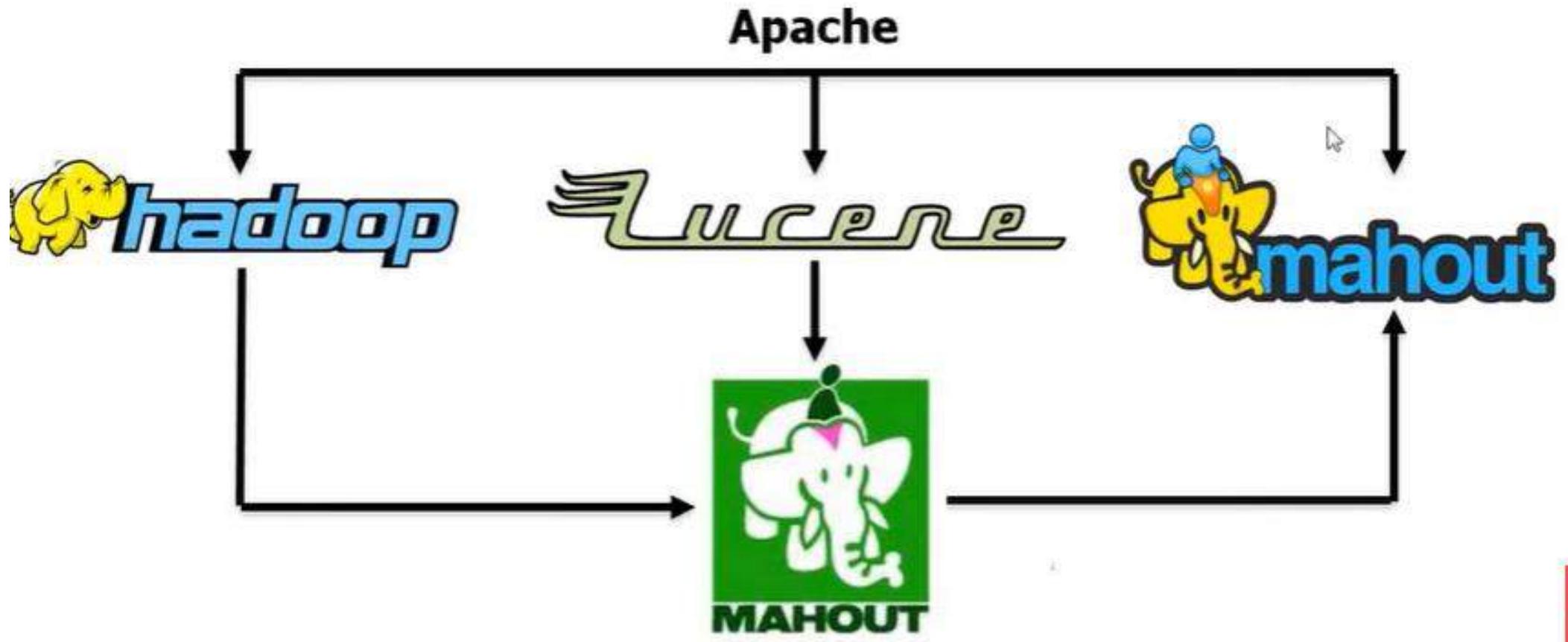
**“MAHOUT”**

# Mahout

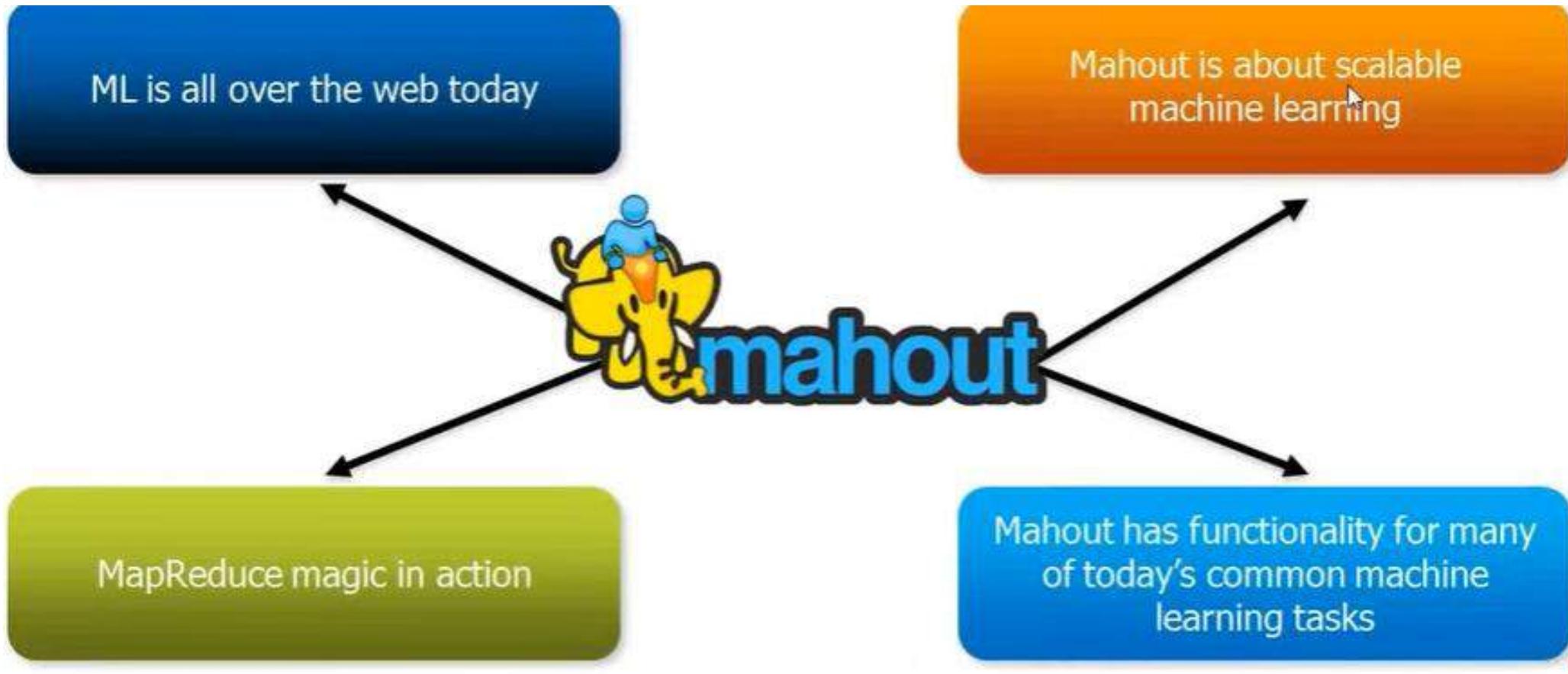
- ✓ Mahout began life in 2008 as a subproject of Apache's Lucene project, which provides the well-known open source search engine of the same name.
- ✓ Lucene provides advanced implementations of search, text mining, and information-retrieval techniques.
- ✓ In the universe of computer science, these concepts are adjacent to machine learning techniques like clustering and, to an extent, classification.
- ✓ As a result, some of the work of the Lucene committers that fell more into these machine learning areas was spun off into its own subproject.
- ✓ Soon after, Mahout absorbed the Taste open source collaborative filtering project



## Apache Mahout and its related projects within the Apache Software Foundation



# Mahout



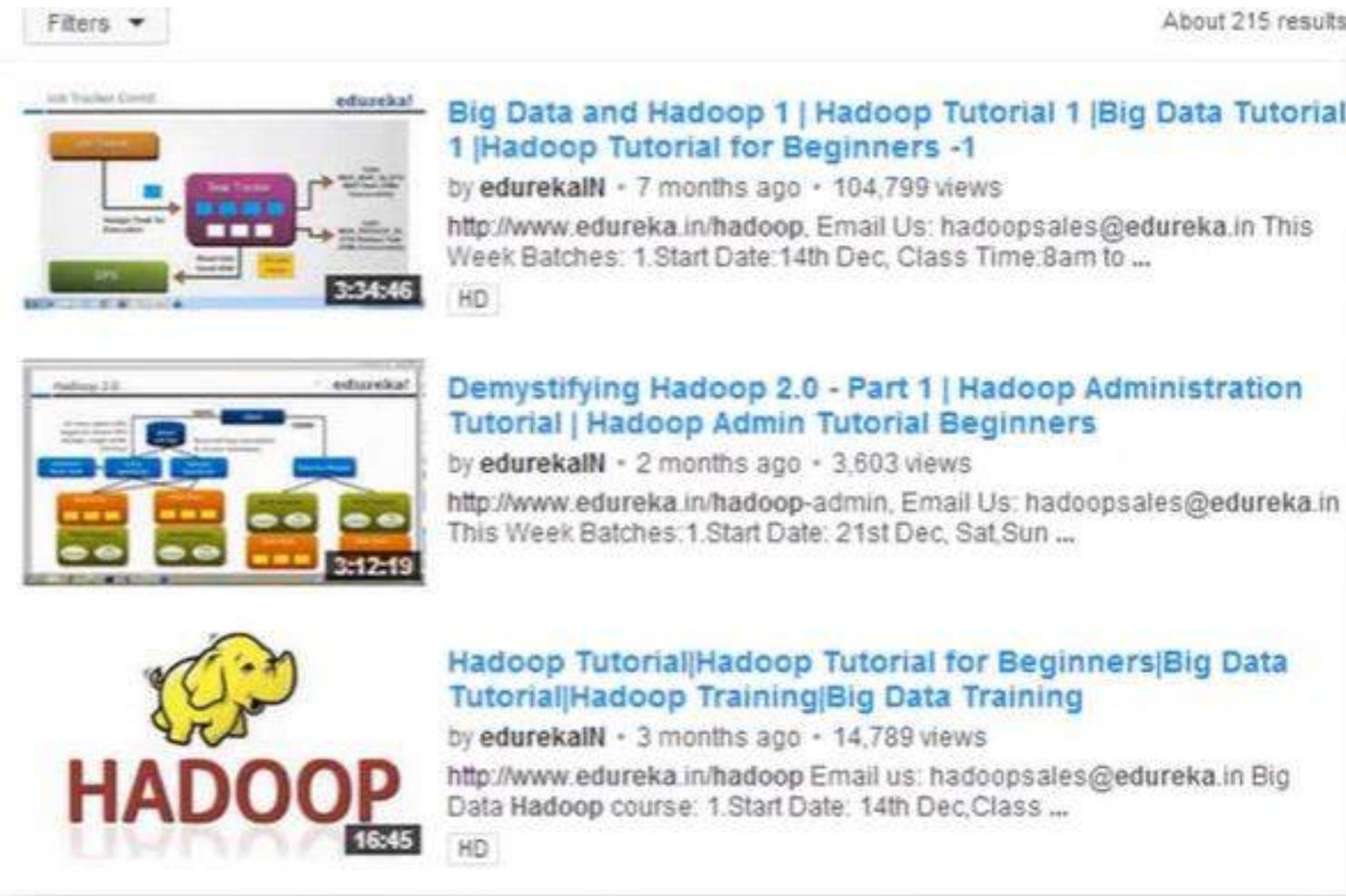
# Use Case:

**YouTube** utilizes recommendation systems to bring videos to a user that it believes the user will be interested in.

They are designed to:

- ✓ Increase the numbers of videos the user will watch
- ✓ Increase the length of time he spends on the site, and
- ✓ Maximize the enjoyment of his YouTube experience.

Filters ▾ About 215 results



**Big Data and Hadoop 1 | Hadoop Tutorial 1 |Big Data Tutorial 1 |Hadoop Tutorial for Beginners -1**  
by edurekaIN • 7 months ago • 104,799 views  
<http://www.edureka.in/hadoop>. Email Us: [hadoopsales@edureka.in](mailto:hadoopsales@edureka.in) This Week Batches: 1. Start Date: 14th Dec, Class Time: 8am to ...  
HD

**Demystifying Hadoop 2.0 - Part 1 | Hadoop Administration Tutorial | Hadoop Admin Tutorial Beginners**  
by edurekaIN • 2 months ago • 3,603 views  
<http://www.edureka.in/hadoop-admin>, Email Us: [hadoopsales@edureka.in](mailto:hadoopsales@edureka.in) This Week Batches: 1. Start Date: 21st Dec, Sat,Sun ...

**Hadoop Tutorial|Hadoop Tutorial for Beginners|Big Data Tutorial|Hadoop Training|Big Data Training**  
by edurekaIN • 3 months ago • 14,789 views  
<http://www.edureka.in/hadoop> Email us: [hadoopsales@edureka.in](mailto:hadoopsales@edureka.in) Big Data Hadoop course: 1. Start Date: 14th Dec,Class ...  
HD

# Use case: Biometric

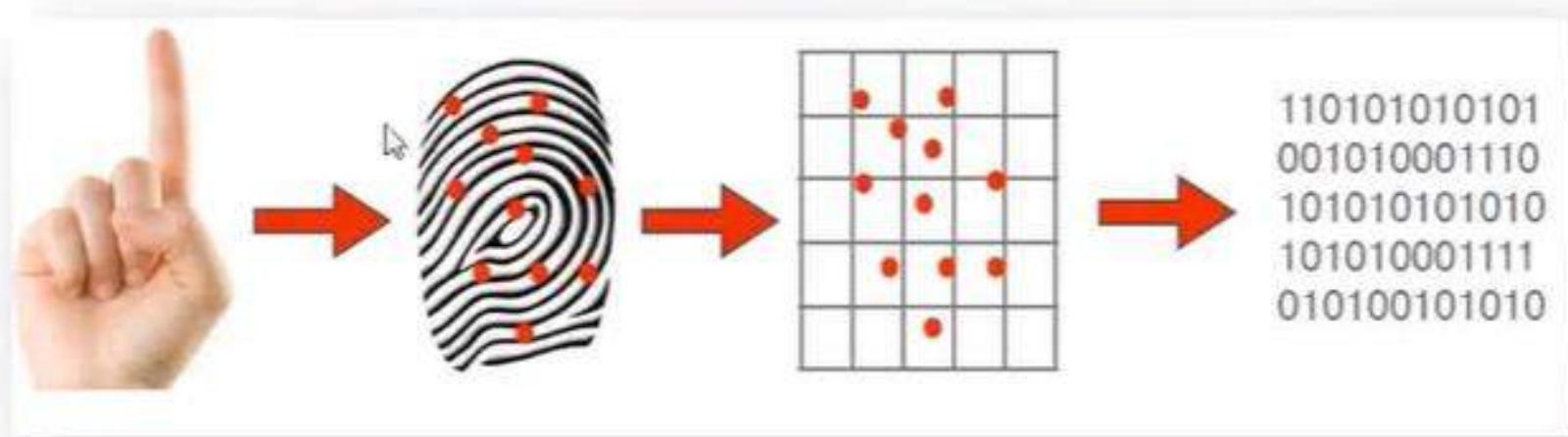


**Biometrics**: The Science of establishing the identity of an individual based on the physical, chemical or behavioral attributes of the person.

## Why is it Important?

- ✓ Identify Individual credentials
- ✓ Identify and prevent banking fraud
- ✓ Enforcement of law and security

# Finger Print Scanner Work:



A **fingerprint scanner system** has two basic jobs

- ✓ Get an image of your finger
- ✓ Determine whether the pattern of ridges and valleys in this image matches the pattern of ridges and valleys in pre-scanned images

## Process

- ✓ Only specific characteristics, which are **unique to every fingerprint**, are filtered and saved as an encrypted **biometric key or mathematical representation**.
- ✓ No image of a fingerprint is ever saved, only a series of numbers (a binary code), which is used for verification. The algorithm cannot be reconverted to an image, so no one can duplicate your fingerprints

# Mahout:

- ✓ What is Learning?
- ✓ Can a Machine learn?
- ✓ How to do it ?

## **Mahout : Scalable Machine learning Library**

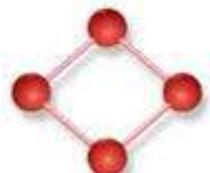
Machine Learning is Programming Computers to optimize Performance Criterion using Example Data or Past experience.

- ✓ A branch of artificial intelligence
- ✓ Systems that learn from data
- ✓ Classify data after learning
- ✓ Learn on test data sets
- ✓ Generalisation – the ability to classify unseen data sets

# Mahout : Perform



Collaborative Filtering



Clustering



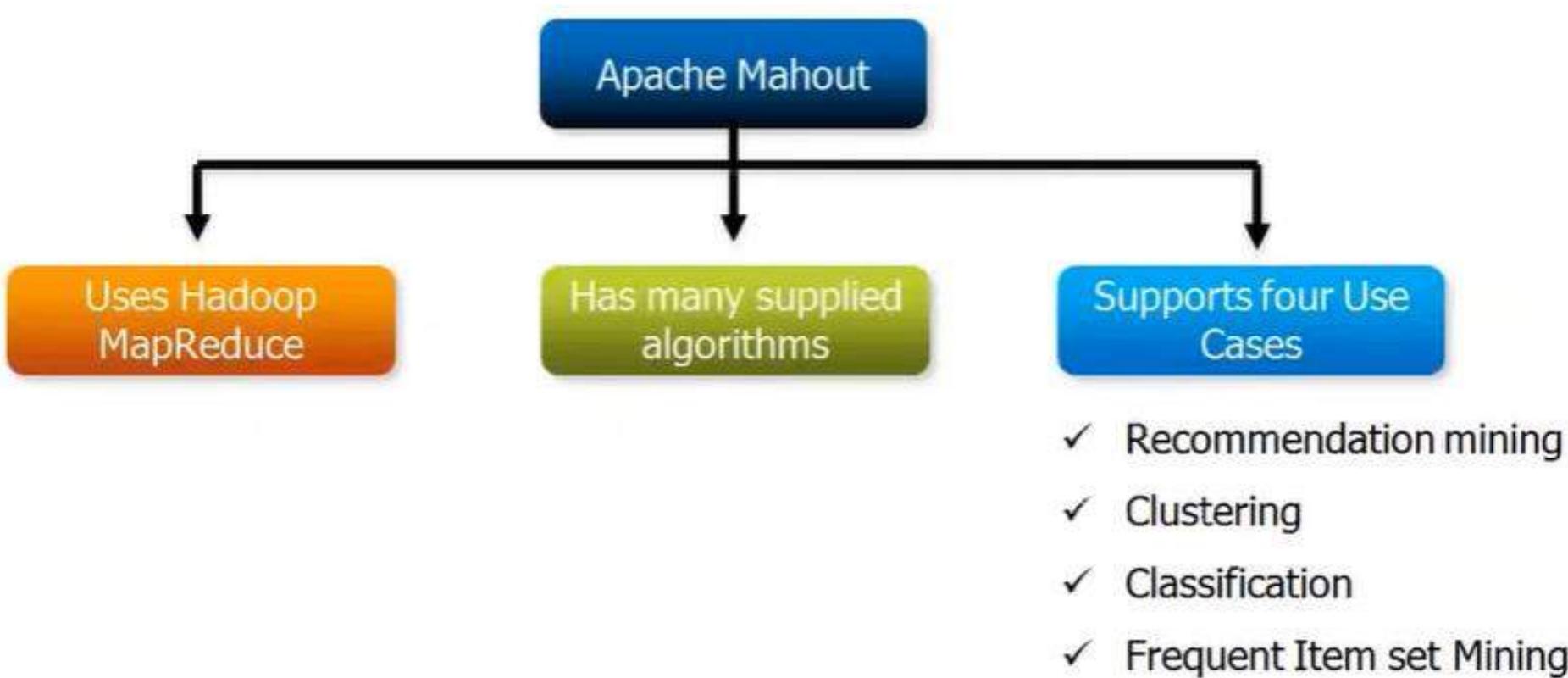
- ✓ Apache Mahout is an Apache project to produce open source implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the areas of collaborative filtering, clustering and classification, often leveraging, but not limited to, the Hadoop platform.

- ✓ The Apache Mahout project aims to make building intelligent applications easier and faster. Mahout co-founder Grant Ingersoll introduces the basic concepts of machine learning and then demonstrates how to use Mahout to cluster documents, make recommendations, and organize content.

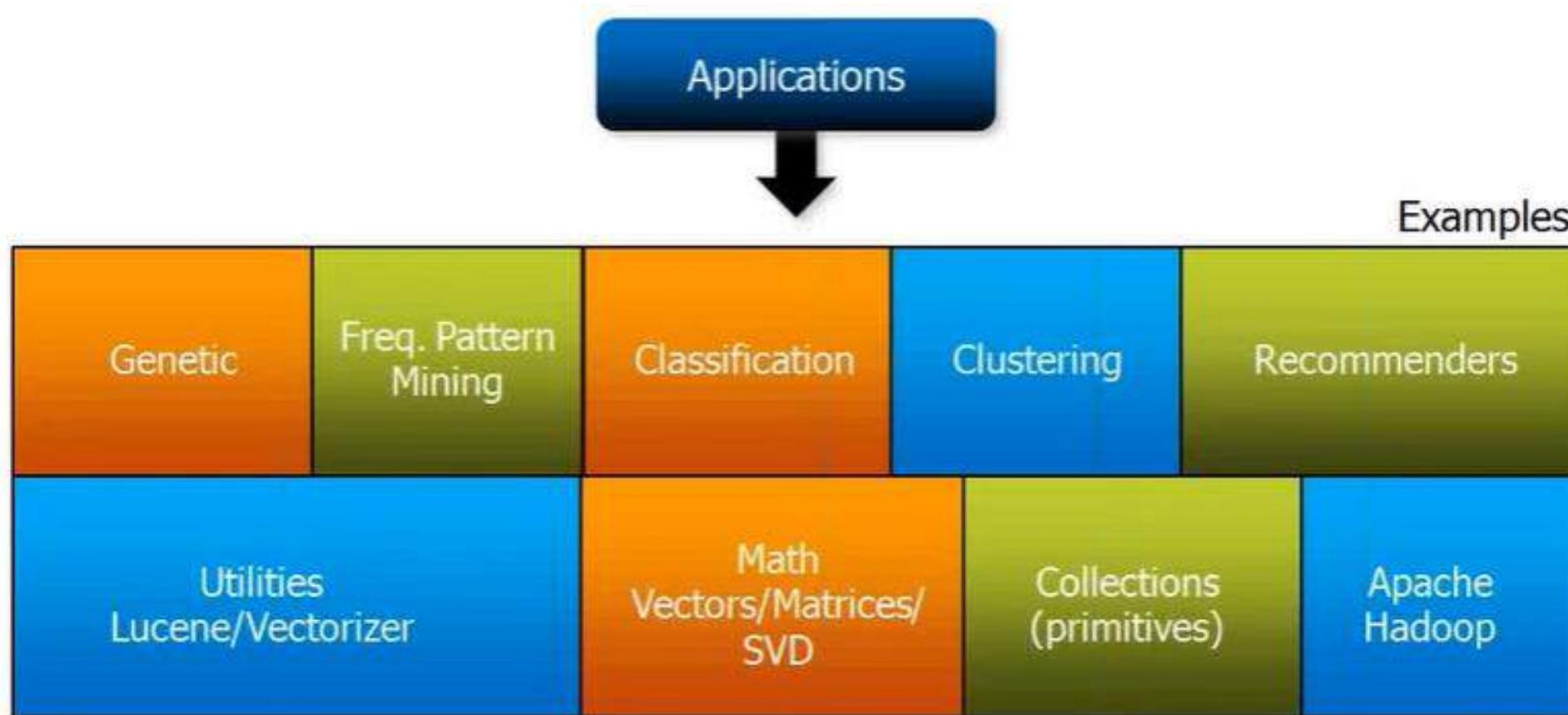
**Three specific machine-learning tasks that Mahout currently implements are:**

- ✓ Collaborative Filtering
- ✓ Clustering
- ✓ Classification

# Mahout- How Does It work

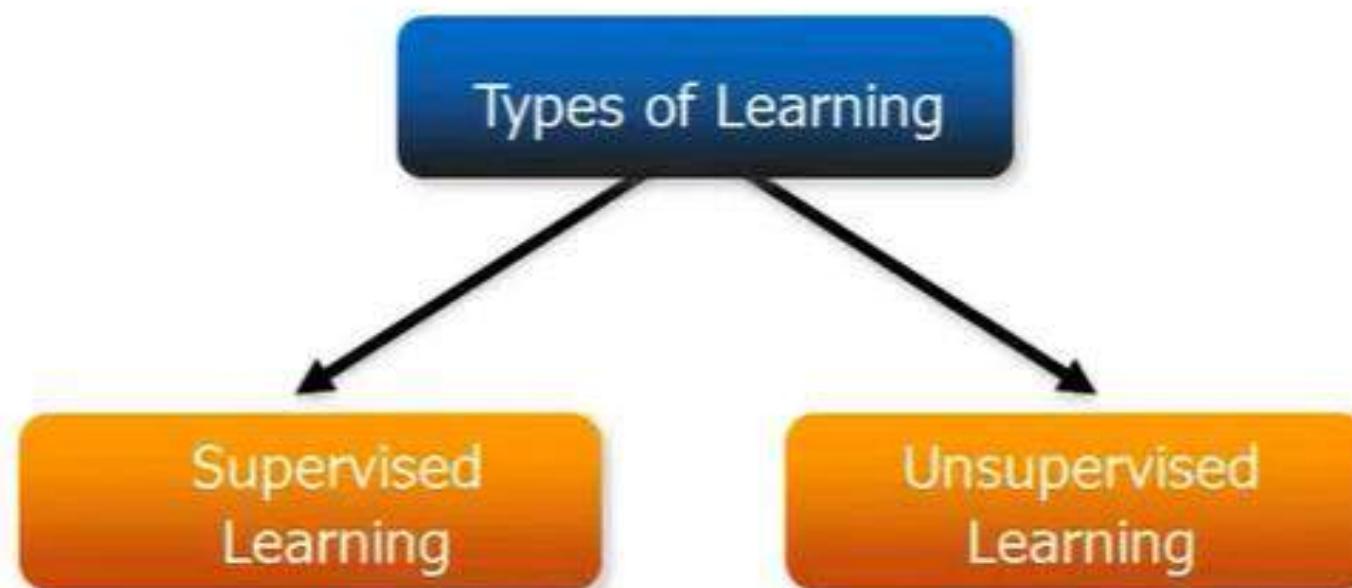


# Mahout: Applications



# Learning Technique: Types of Learning

**Attain knowledge by study, experience, or by being taught.**



# Supervised Learning:

**Supervised learning : Training data includes both the input and the desired results.**

- ✓ For some examples, the correct results (targets) are known and are given in input to the model during the learning process.
- ✓ The construction of a proper training, validation and test set is crucial.
- ✓ These methods are usually fast and accurate.
- ✓ Have to be able to generalize: give the correct results when new data are given in input without knowing a priori the target.

# Unsupervised Learning:

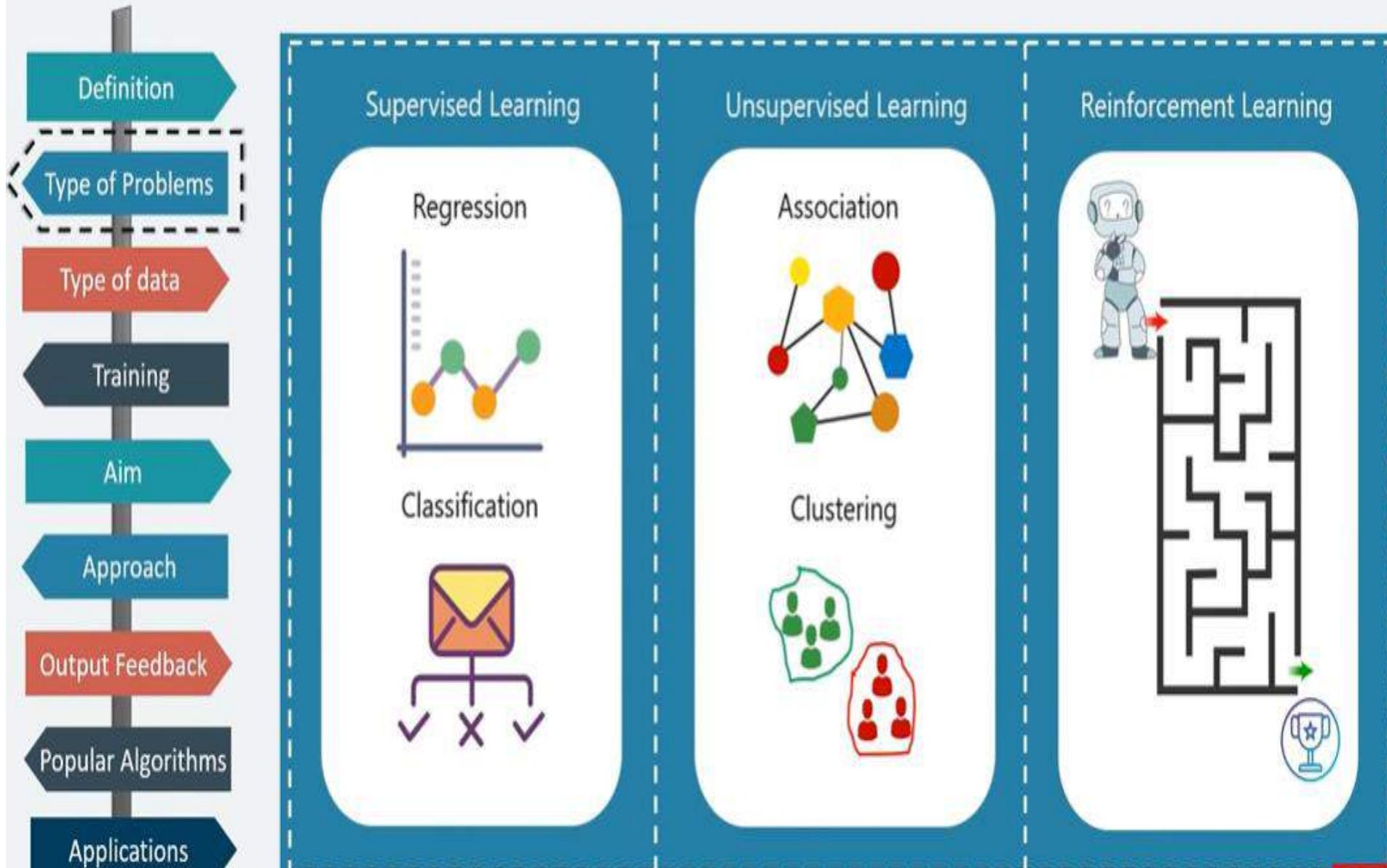
## Unsupervised Learning:

- ✓ The model is not provided with the correct results during the training.
- ✓ Can be used to cluster the input data in classes on the basis of their statistical properties only Cluster significance and labeling.
- ✓ The labeling can be carried out even if the labels are only available for a small number of objects representative of the desired classes.

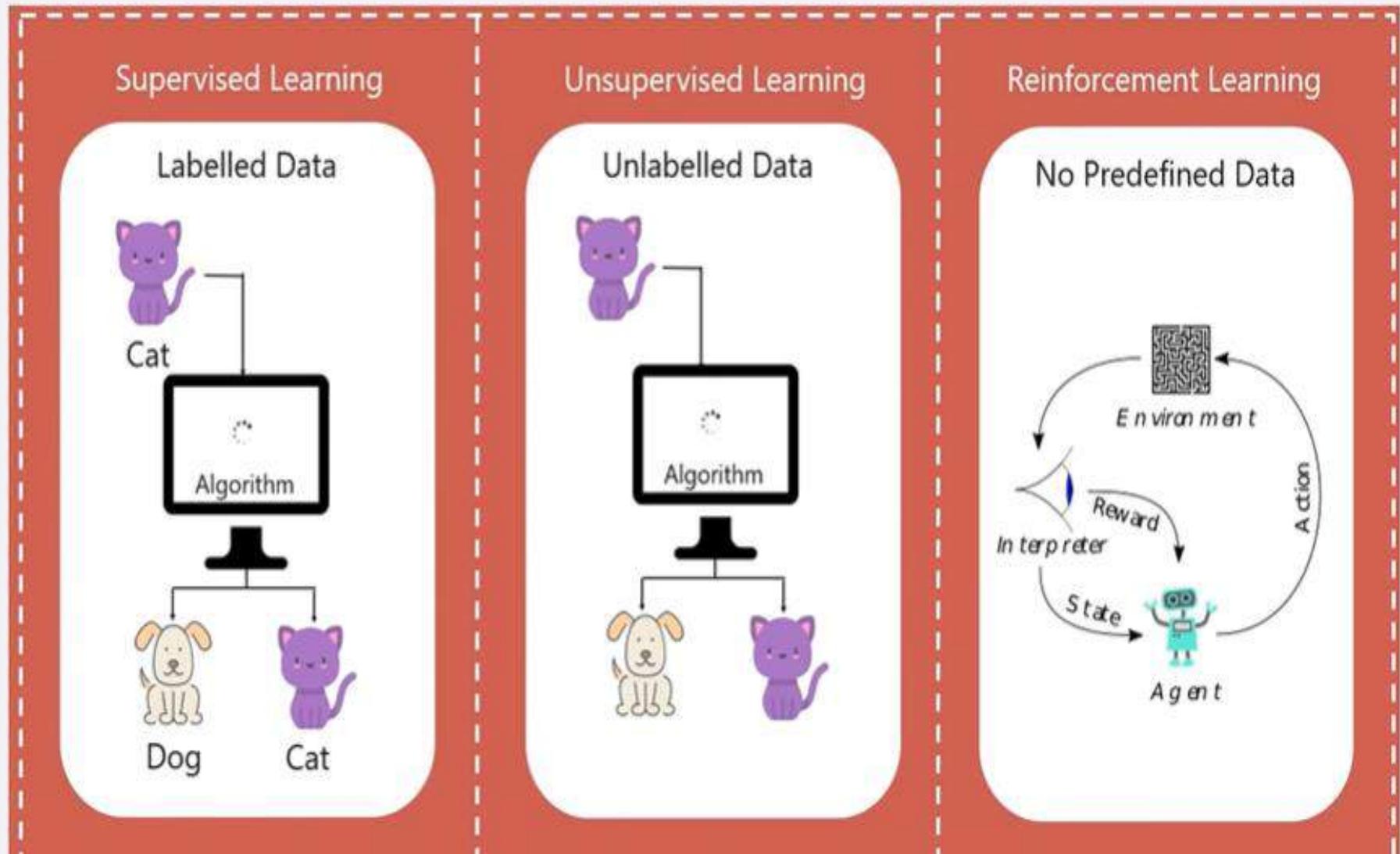
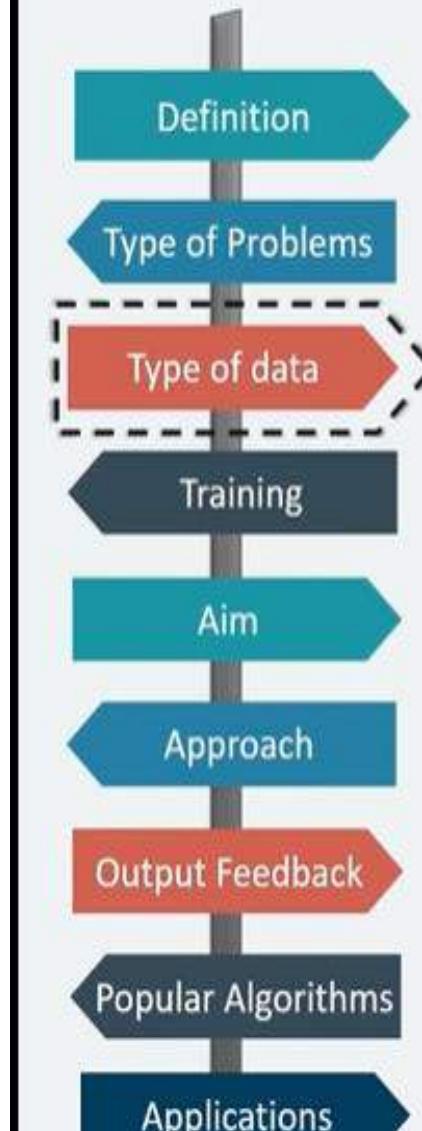
# Supervised & Unsupervised Learning

Parameters	Supervised machine learning	Unsupervised machine learning
Input Data	Algorithms are trained using labeled data.	Algorithms are used against data that is not labeled
Computational Complexity	Simpler method	Computationally complex
Accuracy	Highly accurate	Less accurate
No. of classes	No. of classes is known	No. of classes is not known
Data Analysis	Uses offline analysis	Uses real-time analysis of data
Algorithms used	Linear and Logistics regression, Random forest, Support Vector Machine, Neural Network, etc.	K-Means clustering, Hierarchical clustering, Apriori algorithm, etc.

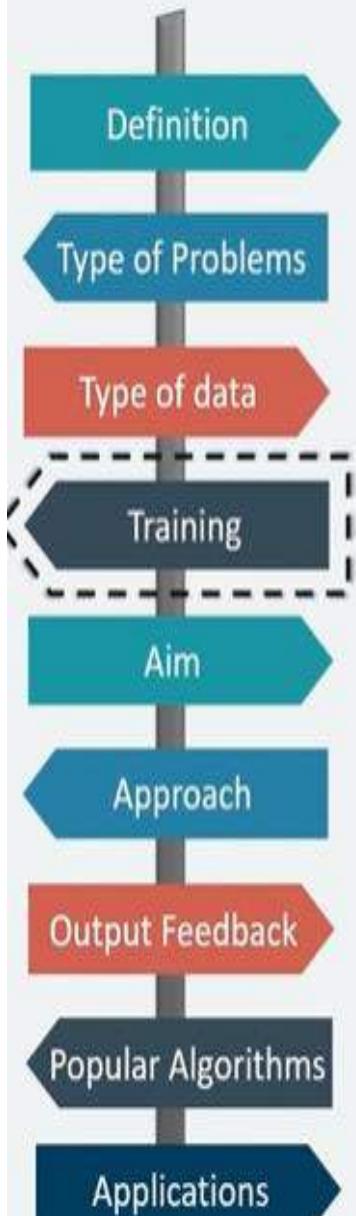
# Problem Type



# Type of data



# Training



## Supervised Learning

External supervision



## Unsupervised Learning

No supervision

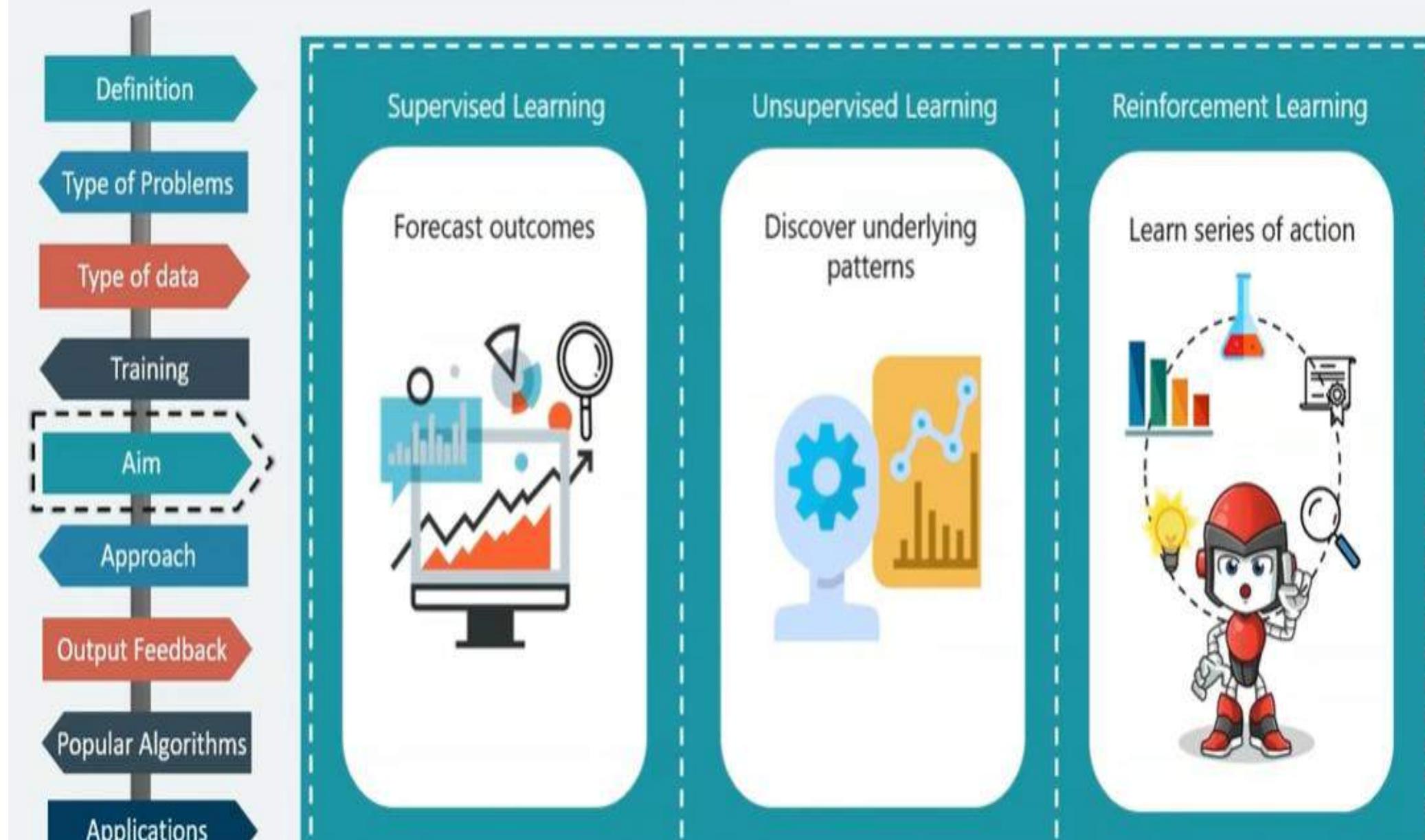


## Reinforcement Learning

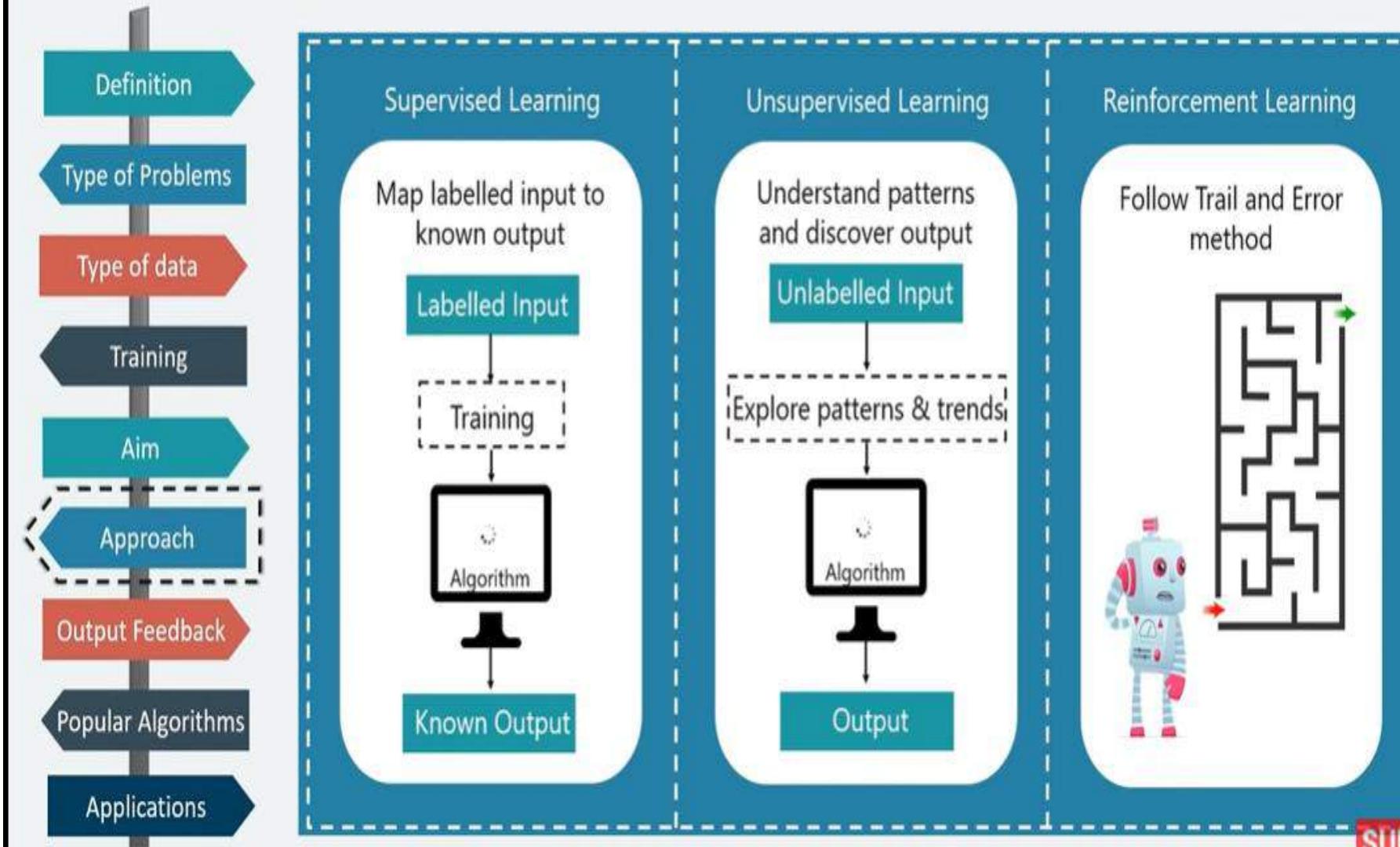
No supervision



# Aim

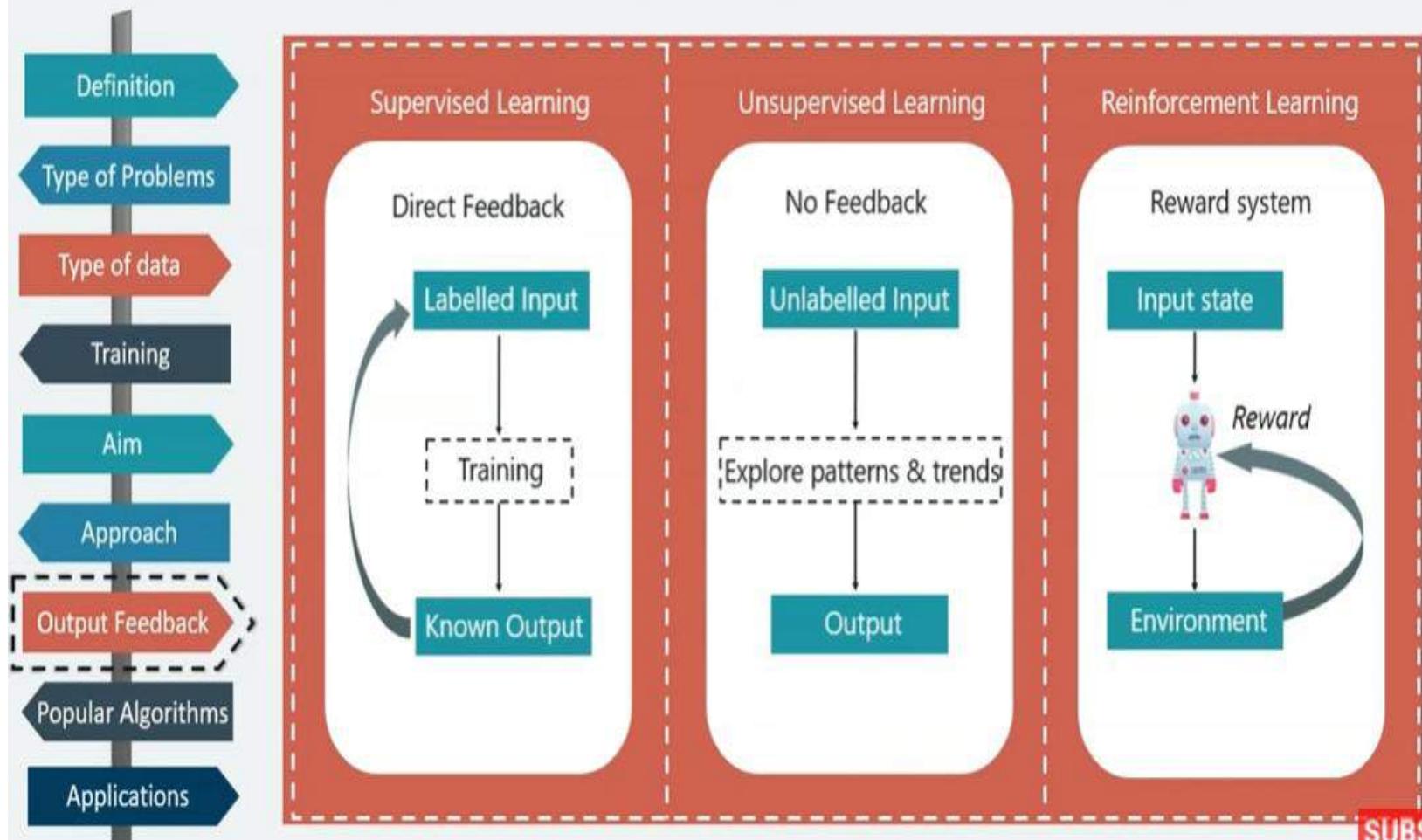


# Approach

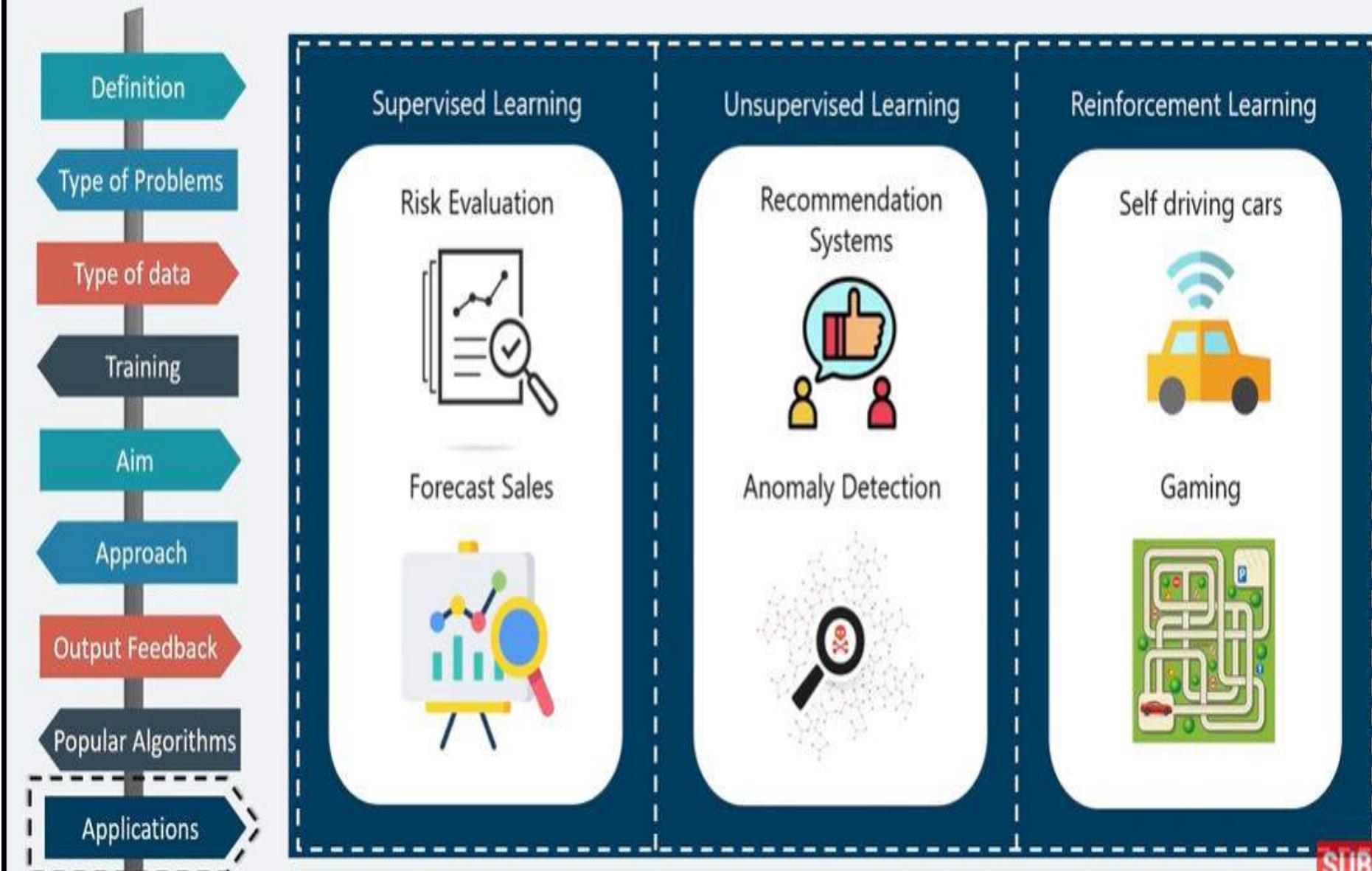


SUF

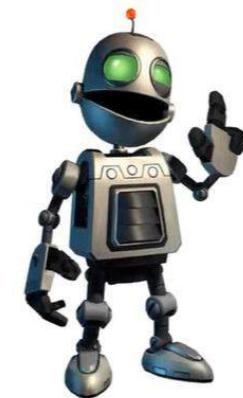
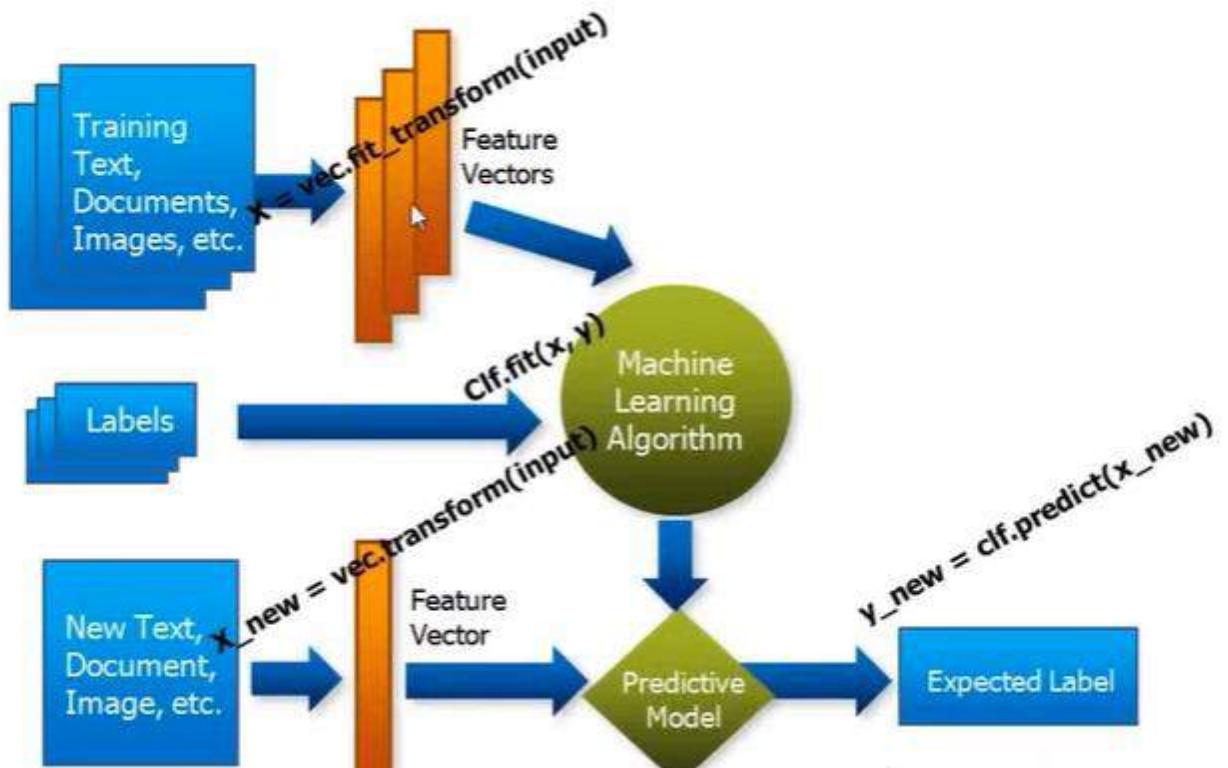
# Output Feedback



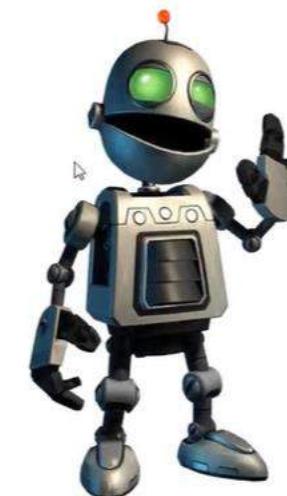
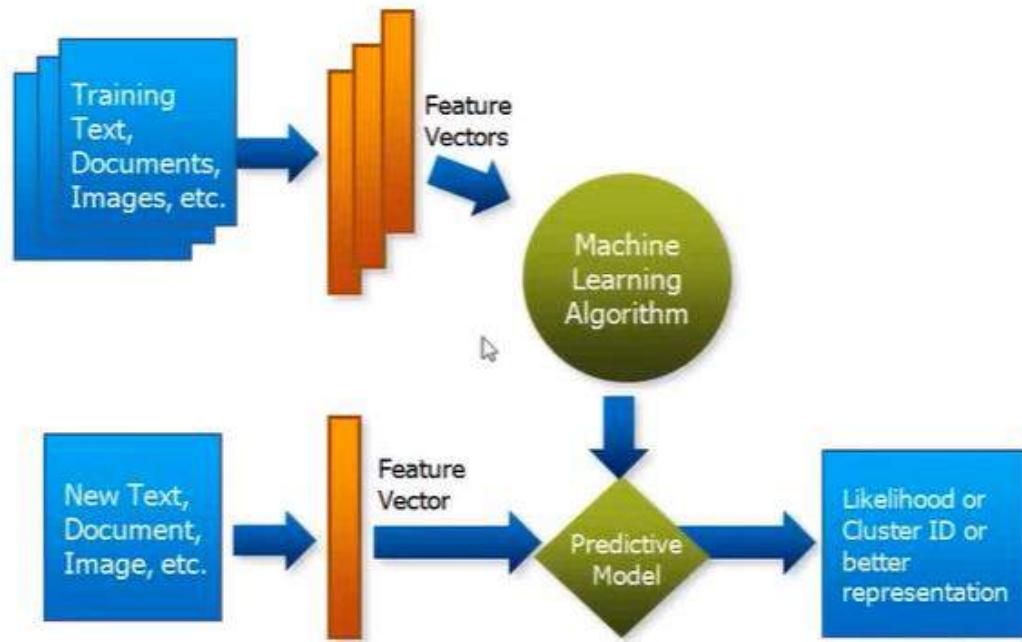
# Applications



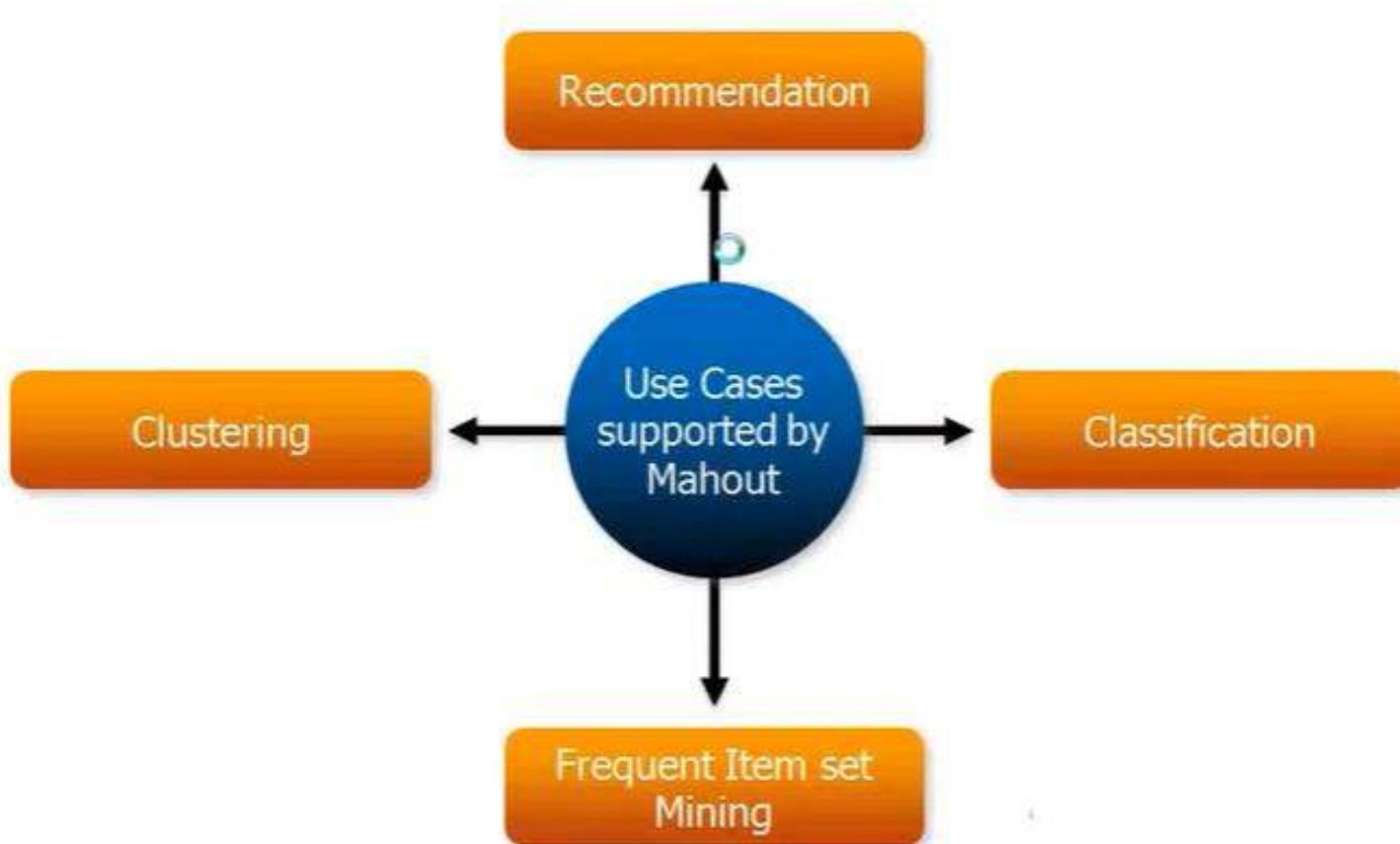
# Supervised Learning:



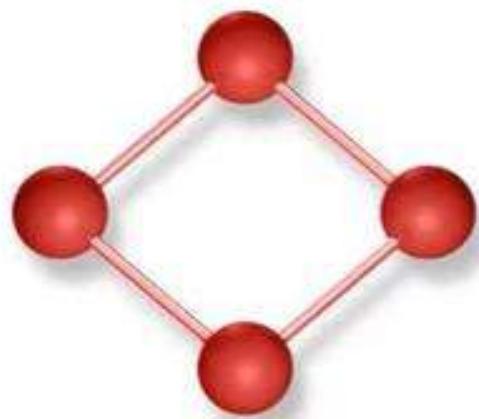
# Unsupervised Learning:



# Applications Of Mahout:



# Clustering:



Clustering

Organizing data into *clusters* such that there is:

- ✓ High intra-cluster similarity
- ✓ Low inter-cluster similarity
- ✓ Informally, finding natural groupings among objects.

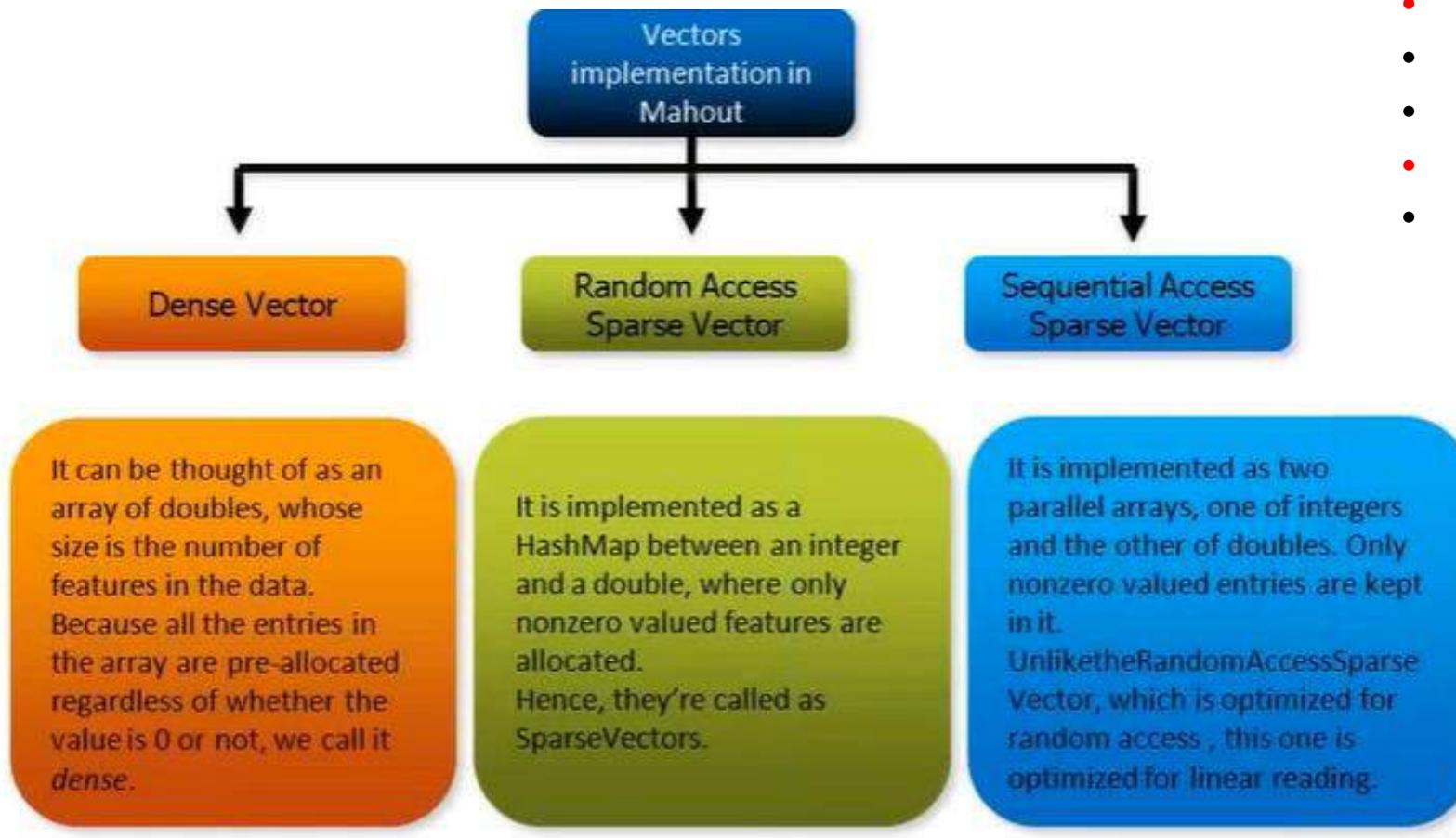


Why do we want to do it??

# Why Clustering?

- ✓ Organizing data into clusters shows internal structure of the data  
*Ex. Clusty and clustering genes*
- ✓ Sometimes the partitioning is the goal  
*Ex. Market segmentation*
- ✓ Prepare for other AI techniques  
*Ex. Summarize news (cluster and then find centroid)*
- ✓ Techniques for clustering is useful in knowledge discovery in data  
*Ex. Underlying rules, reoccurring patterns, topics, etc.*

# Vector Implementations:



- **Dense:** Assume 50 Dimension Vector.
- All values should exist.
- If some value is missing replace with 0.
- **Sparse (Random):**
- If any value is not allocated to specific feature then don't consider or ignore them or exclude that variable which has no value assigned to it..

{x, y, z} - {12.0, 0, 87.9} Dense vector

Sparse Vector{x:12.0, z:87.9}  
Sequential Access sparseVector  
[0,1,2]  
[12.0, 34.5, 67.8]

## **Similarity measurement definition**

Similarity by Correlation

Similarity by Distance

# Similarity by distance

Euclidean distance measure

Manhattan distance measure

Cosine distance measure

Tanimoto distance measure

Squared Euclidean distance measure

# Recommendation or Collaborative Filtering

A Mahout-based collaborative filtering engine **takes users' preferences for items ("tastes") and returns estimated preferences for other items.**

- Suppose you want to purchase the book "**Mahout in Action**" from Amazon:

The screenshot shows the product page for "Mahout in Action" on Amazon. At the top, there's a search bar with "mahout" and a "Go" button. Below the search bar, the product title "Mahout in Action Paperback – Import, 17 Oct 2011" is displayed. The main image of the book cover features a man in a hat and coat. To the right of the image, there's a price box showing "recomendation1.png" for ₹ 2,069.88. Below the image, there are sections for "Delivery to pincode 500001 - Hyderabad : within 1 - 2 weeks.", "Details", and "Summary". The summary text describes the book as a hands-on introduction to machine learning with Apache Mahout, mentioning real-world examples.

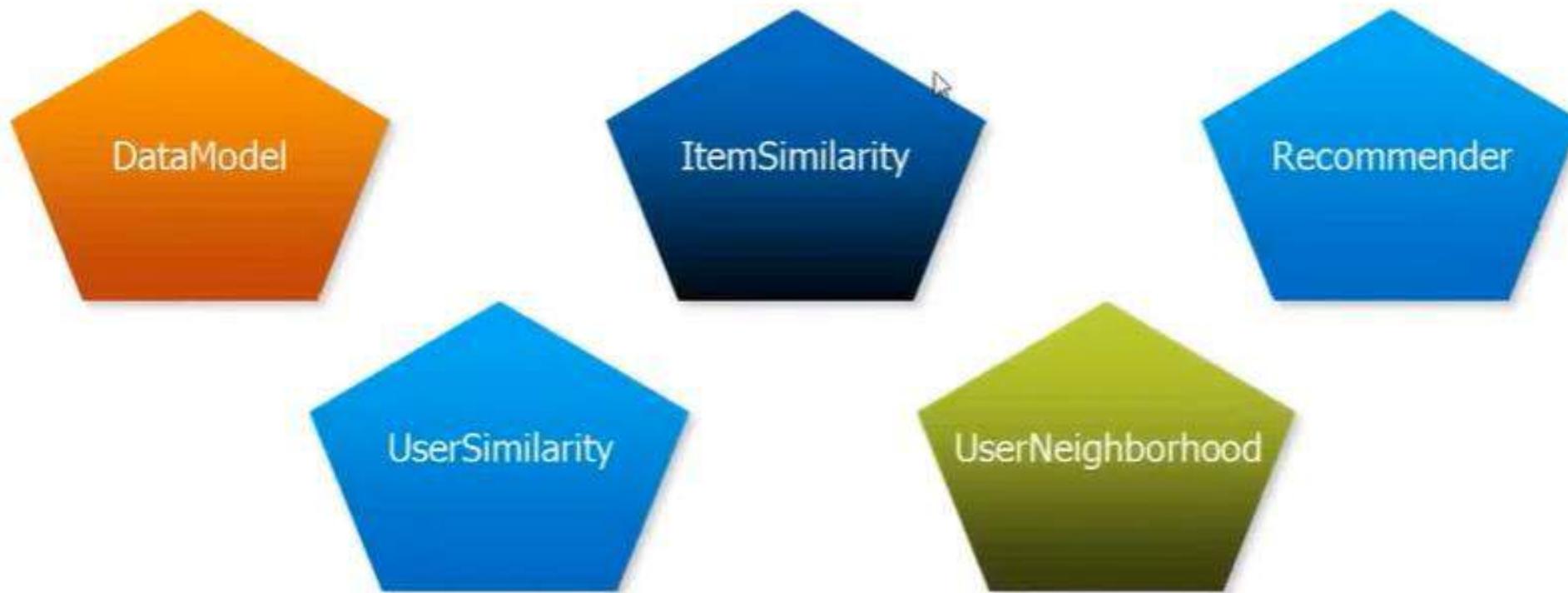
The screenshot shows two recommendation sections on the same product page. The first section, "Frequently Bought Together", displays two related books: "Mahout in Action" and "Taming Text: How to Find, Organize, and Manipulate It by Grant S. Ingersoll". It shows the combined price of ₹ 4,139.76 and a "Add both to Cart" button. The second section, "Customers Who Viewed This Item Also Viewed", lists four other books: "Hadoop: The Definitive Guide" by White, "Hadoop Operations" by Eric Sammer, "Programming Pig" by Alan Gates, and "Hadoop in Action" by Chuck Lam. Each listing includes the book cover, title, author, rating, and price.

- Such recommendation lists are produced with the help of **recommender engines**. Mahout provides recommender engines of several types such as: **user-based , item-based**

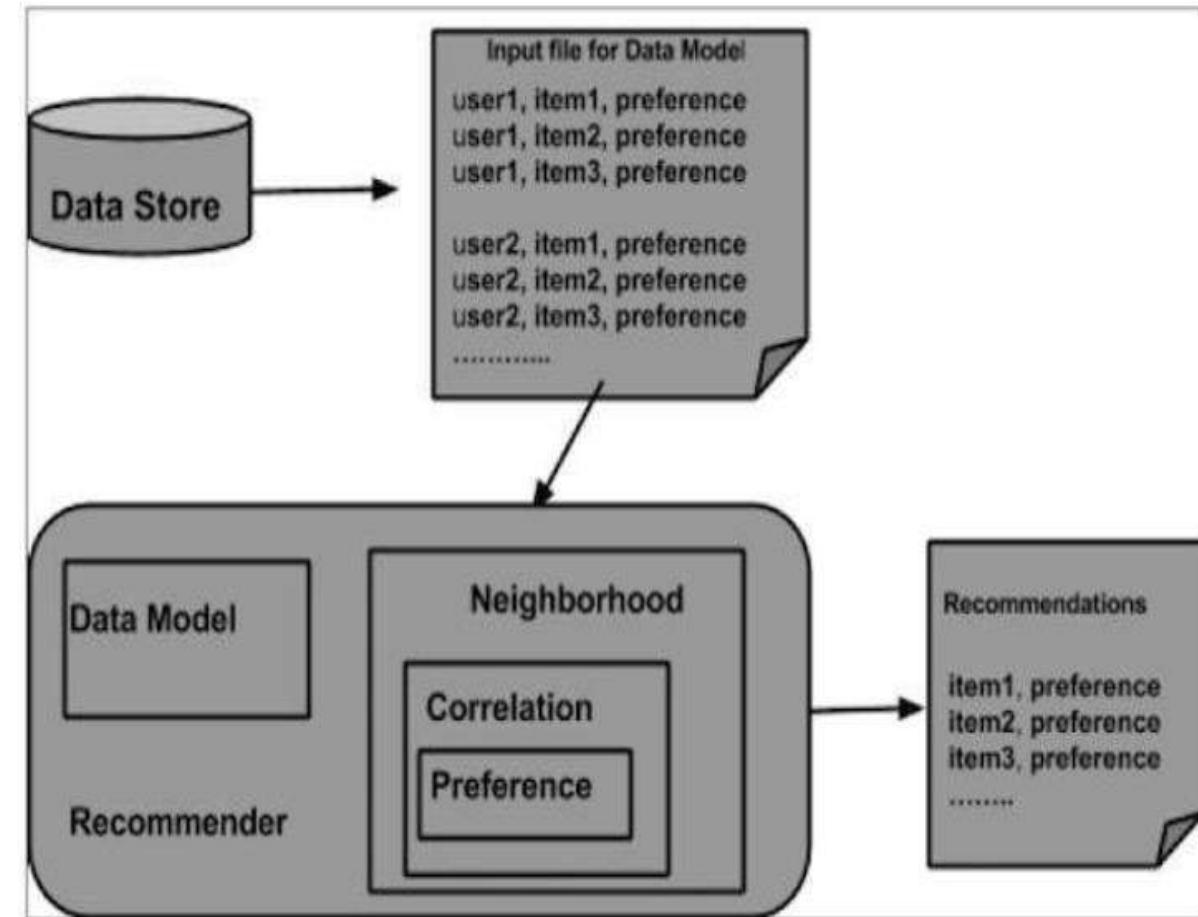
# Mahout Packages:

- The components provided by Mahout to build a recommender engine are as follows:

**Top-level packages define the Mahout interfaces to these key abstractions:**



# Architecture of Recommender Engine Step1: Create DataModel Object



- The DataModel object requires the **file object**, which contains the **path of the input file**. Create the DataModel object as shown below.

```
DataModel datamodel = new FileDataModel(new File("input file"));
```

## Step2: Create UserSimilarity Object

Create **UserSimilarity** object using **PearsonCorrelationSimilarity** class as shown below:

```
UserSimilarity similarity = new
PearsonCorrelationSimilarity(datamodel);
```

## Step3: Create UserNeighborhood object

This object computes a "neighborhood" of users like a given user. There are two types of neighborhoods:

- **ThresholdUserNeighborhood**
- **NearestNUserNeighborhood**

```
UserNeighborhood neighborhood = new ThresholdUserNeighborhood(3.0, similarity, model);
```

#### Step4: Create Recommender Object

- Create **UserbasedRecomender** object. Pass all the above created objects to its constructor as shown below.

```
UserBasedRecommender recommender = new GenericUserBasedRecommender(model, neighborhood, similarity);
```

#### Step5: Recommend Items to a User

- Recommend products to a user using the **recommend()** method of Recommender interface.
- This method requires **two parameters**. The first represents the **user id** of the user to whom we need to send the recommendations, and the second represents the **number of recommendations to be sent**.

```
List<RecommendedItem> recommendations = recommender.recommend(2, 3);

for (RecommendedItem recommendation : recommendations){
 System.out.println(recommendation);
}
```

# Machine Learning Tools:

DATA SIZE	CLASSIFICATION	TOOLS
Lines Sample Data	Analysis and Visualization	Whiteboard,...
KBs - low MBs Prototype Data	Analysis and Visualization	Matlab, Octave, R, Processing,
MBs - low GBs Online Data	Analysis	NumPy, SciPy, Weka, BLAS/LAPACK
	Visualization	Flare, AmCharts, Raphael, Protovis
GBs - TBs - PBs Big Data	Analysis	<b>Mahout</b> , Giraph MLib

# STEPS TO BE FOLLOWED IN MAHOUT

1. Getting the data
2. Copying text files to The Hadoop Distributed File System (HDFS)
3. Convert our dataset into a SequenceFiles
4. Convert sequenceFiles to sparse vector file format
5. Running k-means text clustering algorithm
6. Interpreting the clustering final result

## STEP 1

The first step is to get our dataset that will eventually represent our raw material on which we will test our clustering algorithm.

## STEP 2

- After downloading our text collections locally, and in order to be able to handle it with mahout, it's time to copy it to our HDFS.

# STEP 3

`mahout seqdirectory -i <I> -o <O> -c UTF-8 -chunk 5`

`-i` : specifying the input directory

`-o` : specifying the output directory

`UTF-8` : specifying the encoding of our input files

`-chunk` : specifying the size of each block of data

File Edit View Search Terminal Help

```
amrit@amrit-HP-Notebook:~$ hadoop fs -mkdir clustering
2020-04-06 22:55:45,500 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
amrit@amrit-HP-Notebook:~$ mahout seqdirectory -i clustering/ -o tragedy-seqfiles -c UTF-8 -chunk 5
```

## STEP 4

- In order to be able to run properly, most algorithms in text mining require a numerical representation of texts.

## STEP 4

- That's why, we should turn the collections of texts we had in the previous steps into numerical feature vectors.
- Therefore, every document is represented as a vector where each element of the vector is a word and its weight respectively.

## STEP 4

```
mahout seq2sparse -nv -i tragedy-seqfiles -o tragedy-vectors
```



-i : specifying the input directory

-o : specifying the output directory

-nv: very important option that keeps the files names for later use when displaying the result of text clustering

## STEP 5

- Before passing to action by applying k-means clustering algorithm on our textual data, there is a simple step left.
- In order to have initial centroids values, we should, in the first place, run canopy clustering on our data.

```
mahout canopy -i <input vectors directory>
```

```
-o <output directory>
```

```
-t1 <threshold value 1>
```

```
-t2 <threshold value 2>
```

```
-dm
```

```
amrit@amrit-HP-Notebook:~$ mahout canopy -i tragedy-vectors/tf-vectors -o tragedy-vectors/tragedy-canopy-centroids -dm org.apache.mahout.common.distance.CosineDistanceMeasure -t1 1500 -t2 2000
```

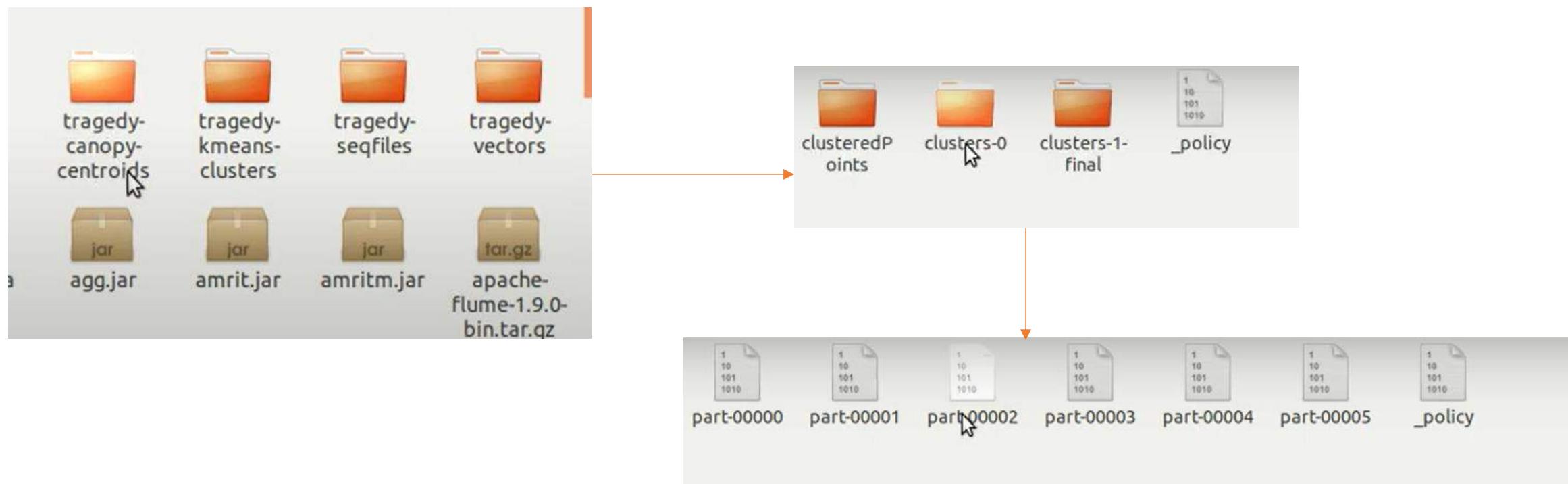
## STEP 5

- Once we have generated initial centroids values we can finally run k-means algorithm on our documents.

## STEP 5 I

```
mahout kmeans -i <INPUT> -c <CENTROID
DIRECTORY> -o <OUTPUT> -dm <DISTANCE
MEASURE> --clustering -cl -cd <convergence
delta parameter> -ow -x <MAX NO OF
ITERATIONS> -k <NO OF CLUSTERS>
```

```
amrit@amrit-HP-Notebook:~/mahout$ mahout kmeans -i tragedy-vectors/tfidf-vectors -c tragedy-canopy-centroids -o tragedy-kmeans-clusters -dm org.apache.mahout.common.distance.CosineDistanceMeasure --clustering -cl -cd 0.1 -ow -x 20 -k 10
```





## Chapter 4

# Mining Data Streams

Most of the algorithms described in this book assume that we are mining a database. That is, all our data is available when and if we want it. In this chapter, we shall make another assumption: data arrives in a stream or streams, and if it is not processed immediately or stored, then it is lost forever. Moreover, we shall assume that the data arrives so rapidly that it is not feasible to store it all in active storage (i.e., in a conventional database), and then interact with it at the time of our choosing.

The algorithms for processing streams each involve summarization of the stream in some way. We shall start by considering how to make a useful sample of a stream and how to filter a stream to eliminate most of the “undesirable” elements. We then show how to estimate the number of different elements in a stream using much less storage than would be required if we listed all the elements we have seen.

Another approach to summarizing a stream is to look at only a fixed-length “window” consisting of the last  $n$  elements for some (typically large)  $n$ . We then query the window as if it were a relation in a database. If there are many streams and/or  $n$  is large, we may not be able to store the entire window for every stream, so we need to summarize even the windows. We address the fundamental problem of maintaining an approximate count on the number of 1’s in the window of a bit stream, while using much less space than would be needed to store the entire window itself. This technique generalizes to approximating various kinds of sums.

### 4.1 The Stream Data Model

Let us begin by discussing the elements of streams and stream processing. We explain the difference between streams and databases and the special problems that arise when dealing with streams. Some typical applications where the stream model applies will be examined.

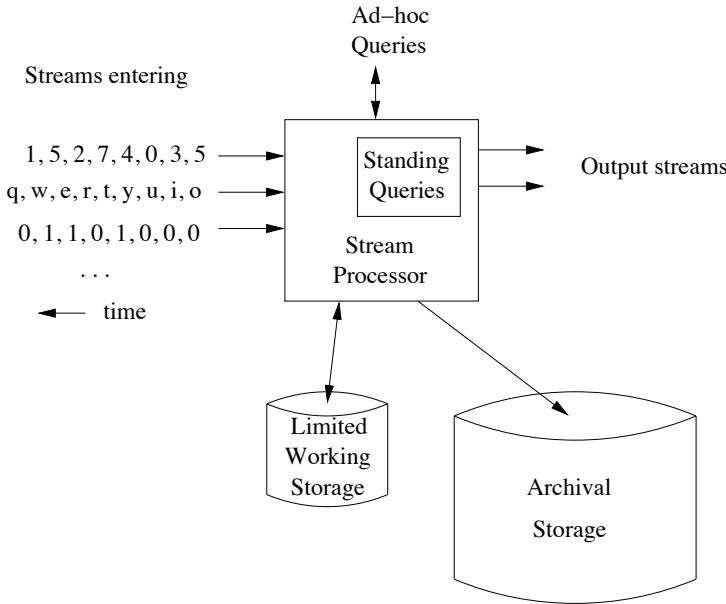


Figure 4.1: A data-stream-management system

#### 4.1.1 A Data-Stream-Management System

In analogy to a database-management system, we can view a stream processor as a kind of data-management system, the high-level organization of which is suggested in Fig. 4.1. Any number of streams can enter the system. Each stream can provide elements at its own schedule; they need not have the same data rates or data types, and the time between elements of one stream need not be uniform. The fact that the rate of arrival of stream elements is not under the control of the system distinguishes stream processing from the processing of data that goes on within a database-management system. The latter system controls the rate at which data is read from the disk, and therefore never has to worry about data getting lost as it attempts to execute queries.

Streams may be archived in a large *archival store*, but we assume it is not possible to answer queries from the archival store. It could be examined only under special circumstances using time-consuming retrieval processes. There is also a *working store*, into which summaries or parts of streams may be placed, and which can be used for answering queries. The working store might be disk, or it might be main memory, depending on how fast we need to process queries. But either way, it is of sufficiently limited capacity that it cannot store all the data from all the streams.

### 4.1.2 Examples of Stream Sources

Before proceeding, let us consider some of the ways in which stream data arises naturally.

#### Sensor Data

Imagine a temperature sensor bobbing about in the ocean, sending back to a base station a reading of the surface temperature each hour. The data produced by this sensor is a stream of real numbers. It is not a very interesting stream, since the data rate is so low. It would not stress modern technology, and the entire stream could be kept in main memory, essentially forever.

Now, give the sensor a GPS unit, and let it report surface height instead of temperature. The surface height varies quite rapidly compared with temperature, so we might have the sensor send back a reading every tenth of a second. If it sends a 4-byte real number each time, then it produces 3.5 megabytes per day. It will still take some time to fill up main memory, let alone a single disk.

But one sensor might not be that interesting. To learn something about ocean behavior, we might want to deploy a million sensors, each sending back a stream, at the rate of ten per second. A million sensors isn't very many; there would be one for every 150 square miles of ocean. Now we have 3.5 terabytes arriving every day, and we definitely need to think about what can be kept in working storage and what can only be archived.

#### Image Data

Satellites often send down to earth streams consisting of many terabytes of images per day. Surveillance cameras produce images with lower resolution than satellites, but there can be many of them, each producing a stream of images at intervals like one second. London is said to have six million such cameras, each producing a stream.

#### Internet and Web Traffic

A switching node in the middle of the Internet receives streams of IP packets from many inputs and routes them to its outputs. Normally, the job of the switch is to transmit data and not to retain it or query it. But there is a tendency to put more capability into the switch, e.g., the ability to detect denial-of-service attacks or the ability to reroute packets based on information about congestion in the network.

Web sites receive streams of various types. For example, Google receives several hundred million search queries per day. Yahoo! accepts billions of “clicks” per day on its various sites. Many interesting things can be learned from these streams. For example, an increase in queries like “sore throat” enables us to track the spread of viruses. A sudden increase in the click rate for a link could

indicate some news connected to that page, or it could mean that the link is broken and needs to be repaired.

### 4.1.3 Stream Queries

There are two ways that queries get asked about streams. We show in Fig. 4.1 a place within the processor where *standing queries* are stored. These queries are, in a sense, permanently executing, and produce outputs at appropriate times.

**Example 4.1:** The stream produced by the ocean-surface-temperature sensor mentioned at the beginning of Section 4.1.2 might have a standing query to output an alert whenever the temperature exceeds 25 degrees centigrade. This query is easily answered, since it depends only on the most recent stream element.

Alternatively, we might have a standing query that, each time a new reading arrives, produces the average of the 24 most recent readings. That query also can be answered easily, if we store the 24 most recent stream elements. When a new stream element arrives, we can drop from the working store the 25th most recent element, since it will never again be needed (unless there is some other standing query that requires it).

Another query we might ask is the maximum temperature ever recorded by that sensor. We can answer this query by retaining a simple summary: the maximum of all stream elements ever seen. It is not necessary to record the entire stream. When a new stream element arrives, we compare it with the stored maximum, and set the maximum to whichever is larger. We can then answer the query by producing the current value of the maximum. Similarly, if we want the average temperature over all time, we have only to record two values: the number of readings ever sent in the stream and the sum of those readings. We can adjust these values easily each time a new reading arrives, and we can produce their quotient as the answer to the query.  $\square$

The other form of query is *ad-hoc*, a question asked once about the current state of a stream or streams. If we do not store all streams in their entirety, as normally we can not, then we cannot expect to answer arbitrary queries about streams. If we have some idea what kind of queries will be asked through the ad-hoc query interface, then we can prepare for them by storing appropriate parts or summaries of streams as in Example 4.1.

If we want the facility to ask a wide variety of ad-hoc queries, a common approach is to store a *sliding window* of each stream in the working store. A sliding window can be the most recent  $n$  elements of a stream, for some  $n$ , or it can be all the elements that arrived within the last  $t$  time units, e.g., one day. If we regard each stream element as a tuple, we can treat the window as a relation and query it with any SQL query. Of course the stream-management system must keep the window fresh, deleting the oldest elements as new ones come in.

**Example 4.2:** Web sites often like to report the number of unique users over the past month. If we think of each login as a stream element, we can maintain a window that is all logins in the most recent month. We must associate the arrival time with each login, so we know when it no longer belongs to the window. If we think of the window as a relation `Logins(name, time)`, then it is simple to get the number of unique users over the past month. The SQL query is:

```
SELECT COUNT(DISTINCT(name))
FROM Logins
WHERE time >= t;
```

Here,  $t$  is a constant that represents the time one month before the current time.

Note that we must be able to maintain the entire stream of logins for the past month in working storage. However, for even the largest sites, that data is not more than a few terabytes, and so surely can be stored on disk.  $\square$

#### 4.1.4 Issues in Stream Processing

Before proceeding to discuss algorithms, let us consider the constraints under which we work when dealing with streams. First, streams often deliver elements very rapidly. We must process elements in real time, or we lose the opportunity to process them at all, without accessing the archival storage. Thus, it often is important that the stream-processing algorithm is executed in main memory, without access to secondary storage or with only rare accesses to secondary storage. Moreover, even when streams are “slow,” as in the sensor-data example of Section 4.1.2, there may be many such streams. Even if each stream by itself can be processed using a small amount of main memory, the requirements of all the streams together can easily exceed the amount of available main memory.

Thus, many problems about streaming data would be easy to solve if we had enough memory, but become rather hard and require the invention of new techniques in order to execute them at a realistic rate on a machine of realistic size. Here are two generalizations about stream algorithms worth bearing in mind as you read through this chapter:

- Often, it is much more efficient to get an approximate answer to our problem than an exact solution.
- As in Chapter 3, a variety of techniques related to hashing turn out to be useful. Generally, these techniques introduce useful randomness into the algorithm’s behavior, in order to produce an approximate answer that is very close to the true result.

## 4.2 Sampling Data in a Stream

As our first example of managing streaming data, we shall look at extracting reliable samples from a stream. As with many stream algorithms, the “trick” involves using hashing in a somewhat unusual way.

### 4.2.1 A Motivating Example

The general problem we shall address is selecting a subset of a stream so that we can ask queries about the selected subset and have the answers be statistically representative of the stream as a whole. If we know what queries are to be asked, then there are a number of methods that might work, but we are looking for a technique that will allow ad-hoc queries on the sample. We shall look at a particular problem, from which the general idea will emerge.

Our running example is the following. A search engine receives a stream of queries, and it would like to study the behavior of typical users.<sup>1</sup> We assume the stream consists of tuples (user, query, time). Suppose that we want to answer queries such as “What fraction of the typical user’s queries were repeated over the past month?” Assume also that we wish to store only 1/10th of the stream elements.

The obvious approach would be to generate a random number, say an integer from 0 to 9, in response to each search query. Store the tuple if and only if the random number is 0. If we do so, each user has, on average, 1/10th of their queries stored. Statistical fluctuations will introduce some noise into the data, but if users issue many queries, the law of large numbers will assure us that most users will have a fraction quite close to 1/10th of their queries stored.

However, this scheme gives us the wrong answer to the query asking for the average number of duplicate queries for a user. Suppose a user has issued  $s$  search queries one time in the past month,  $d$  search queries twice, and no search queries more than twice. If we have a 1/10th sample, of queries, we shall see in the sample for that user an expected  $s/10$  of the search queries issued once. Of the  $d$  search queries issued twice, only  $d/100$  will appear twice in the sample; that fraction is  $d$  times the probability that both occurrences of the query will be in the 1/10th sample. Of the queries that appear twice in the full stream,  $18d/100$  will appear exactly once. To see why, note that  $18/100$  is the probability that one of the two occurrences will be in the 1/10th of the stream that is selected, while the other is in the 9/10th that is not selected.

The correct answer to the query about the fraction of repeated searches is  $d/(s+d)$ . However, the answer we shall obtain from the sample is  $d/(10s+19d)$ . To derive the latter formula, note that  $d/100$  appear twice, while  $s/10+18d/100$  appear once. Thus, the fraction appearing twice in the sample is  $d/100$  divided

---

<sup>1</sup>While we shall refer to “users,” the search engine really receives IP addresses from which the search query was issued. We shall assume that these IP addresses identify unique users, which is approximately true, but not exactly true.

by  $d/100 + s/10 + 18d/100$ . This ratio is  $d/(10s + 19d)$ . For no positive values of  $s$  and  $d$  is  $d/(s+d) = d/(10s+19d)$ .

### 4.2.2 Obtaining a Representative Sample

The query of Section 4.2.1, like many queries about the statistics of typical users, cannot be answered by taking a sample of each user's search queries. Thus, we must strive to pick 1/10th of the users, and take all their searches for the sample, while taking none of the searches from other users. If we can store a list of all users, and whether or not they are in the sample, then we could do the following. Each time a search query arrives in the stream, we look up the user to see whether or not they are in the sample. If so, we add this search query to the sample, and if not, then not. However, if we have no record of ever having seen this user before, then we generate a random integer between 0 and 9. If the number is 0, we add this user to our list with value "in," and if the number is other than 0, we add the user with the value "out."

That method works as long as we can afford to keep the list of all users and their in/out decision in main memory, because there isn't time to go to disk for every search that arrives. By using a hash function, one can avoid keeping the list of users. That is, we hash each user name to one of ten buckets, 0 through 9. If the user hashes to bucket 0, then accept this search query for the sample, and if not, then not.

Note we do not actually store the user in the bucket; in fact, there is no data in the buckets at all. Effectively, we use the hash function as a random-number generator, with the important property that, when applied to the same user several times, we always get the same "random" number. That is, without storing the in/out decision for any user, we can reconstruct that decision any time a search query by that user arrives.

More generally, we can obtain a sample consisting of any rational fraction  $a/b$  of the users by hashing user names to  $b$  buckets, 0 through  $b - 1$ . Add the search query to the sample if the hash value is less than  $a$ .

### 4.2.3 The General Sampling Problem

The running example is typical of the following general problem. Our stream consists of tuples with  $n$  components. A subset of the components are the *key* components, on which the selection of the sample will be based. In our running example, there are three components – user, query, and time – of which only *user* is in the key. However, we could also take a sample of queries by making *query* be the key, or even take a sample of user-query pairs by making both those components form the key.

To take a sample of size  $a/b$ , we hash the key value for each tuple to  $b$  buckets, and accept the tuple for the sample if the hash value is less than  $a$ . If the key consists of more than one component, the hash function needs to combine the values for those components to make a single hash-value. The

result will be a sample consisting of all tuples with certain key values. The selected key values will be approximately  $a/b$  of all the key values appearing in the stream.

#### 4.2.4 Varying the Sample Size

Often, the sample will grow as more of the stream enters the system. In our running example, we retain all the search queries of the selected 1/10th of the users, forever. As time goes on, more searches for the same users will be accumulated, and new users that are selected for the sample will appear in the stream.

If we have a budget for how many tuples from the stream can be stored as the sample, then the fraction of key values must vary, lowering as time goes on. In order to assure that at all times, the sample consists of all tuples from a subset of the key values, we choose a hash function  $h$  from key values to a very large number of values  $0, 1, \dots, B-1$ . We maintain a *threshold*  $t$ , which initially can be the largest bucket number,  $B-1$ . At all times, the sample consists of those tuples whose key  $K$  satisfies  $h(K) \leq t$ . New tuples from the stream are added to the sample if and only if they satisfy the same condition.

If the number of stored tuples of the sample exceeds the allotted space, we lower  $t$  to  $t-1$  and remove from the sample all those tuples whose key  $K$  hashes to  $t$ . For efficiency, we can lower  $t$  by more than 1, and remove the tuples with several of the highest hash values, whenever we need to throw some key values out of the sample. Further efficiency is obtained by maintaining an index on the hash value, so we can find all those tuples whose keys hash to a particular value quickly.

#### 4.2.5 Exercises for Section 4.2

**Exercise 4.2.1:** Suppose we have a stream of tuples with the schema

Grades(university, courseID, studentID, grade)

Assume universities are unique, but a courseID is unique only within a university (i.e., different universities may have different courses with the same ID, e.g., “CS101”) and likewise, studentID’s are unique only within a university (different universities may assign the same ID to different students). Suppose we want to answer certain queries approximately from a 1/20th sample of the data. For each of the queries below, indicate how you would construct the sample. That is, tell what the key attributes should be.

- (a) For each university, estimate the average number of students in a course.
- (b) Estimate the fraction of students who have a GPA of 3.5 or more.
- (c) Estimate the fraction of courses where at least half the students got “A.”

## 4.3 Filtering Streams

Another common process on streams is selection, or filtering. We want to accept those tuples in the stream that meet a criterion. Accepted tuples are passed to another process as a stream, while other tuples are dropped. If the selection criterion is a property of the tuple that can be calculated (e.g., the first component is less than 10), then the selection is easy to do. The problem becomes harder when the criterion involves lookup for membership in a set. It is especially hard, when that set is too large to store in main memory. In this section, we shall discuss the technique known as “Bloom filtering” as a way to eliminate most of the tuples that do not meet the criterion.

### 4.3.1 A Motivating Example

Again let us start with a running example that illustrates the problem and what we can do about it. Suppose we have a set  $S$  of one billion allowed email addresses – those that we will allow through because we believe them not to be spam. The stream consists of pairs: an email address and the email itself. Since the typical email address is 20 bytes or more, it is not reasonable to store  $S$  in main memory. Thus, we can either use disk accesses to determine whether or not to let through any given stream element, or we can devise a method that requires no more main memory than we have available, and yet will filter most of the undesired stream elements.

Suppose for argument’s sake that we have one gigabyte of available main memory. In the technique known as *Bloom filtering*, we use that main memory as a bit array. In this case, we have room for eight billion bits, since one byte equals eight bits. Devise a hash function  $h$  from email addresses to eight billion buckets. Hash each member of  $S$  to a bit, and set that bit to 1. All other bits of the array remain 0.

Since there are one billion members of  $S$ , approximately 1/8th of the bits will be 1. The exact fraction of bits set to 1 will be slightly less than 1/8th, because it is possible that two members of  $S$  hash to the same bit. We shall discuss the exact fraction of 1’s in Section 4.3.3. When a stream element arrives, we hash its email address. If the bit to which that email address hashes is 1, then we let the email through. But if the email address hashes to a 0, we are certain that the address is not in  $S$ , so we can drop this stream element.

Unfortunately, some spam email will get through. Approximately 1/8th of the stream elements whose email address is not in  $S$  will happen to hash to a bit whose value is 1 and will be let through. Nevertheless, since the majority of emails are spam (about 80% according to some reports), eliminating 7/8th of the spam is a significant benefit. Moreover, if we want to eliminate every spam, we need only check for membership in  $S$  those good and bad emails that get through the filter. Those checks will require the use of secondary memory to access  $S$  itself. There are also other options, as we shall see when we study the general Bloom-filtering technique. As a simple example, we could use a cascade

of filters, each of which would eliminate 7/8th of the remaining spam.

### 4.3.2 The Bloom Filter

A *Bloom filter* consists of:

1. An array of  $n$  bits, initially all 0's.
2. A collection of hash functions  $h_1, h_2, \dots, h_k$ . Each hash function maps “key” values to  $n$  buckets, corresponding to the  $n$  bits of the bit-array.
3. A set  $S$  of  $m$  key values.

The purpose of the Bloom filter is to allow through all stream elements whose keys are in  $S$ , while rejecting most of the stream elements whose keys are not in  $S$ .

To initialize the bit array, begin with all bits 0. Take each key value in  $S$  and hash it using each of the  $k$  hash functions. Set to 1 each bit that is  $h_i(K)$  for some hash function  $h_i$  and some key value  $K$  in  $S$ .

To test a key  $K$  that arrives in the stream, check that all of

$$h_1(K), h_2(K), \dots, h_k(K)$$

are 1's in the bit-array. If all are 1's, then let the stream element through. If one or more of these bits are 0, then  $K$  could not be in  $S$ , so reject the stream element.

### 4.3.3 Analysis of Bloom Filtering

If a key value is in  $S$ , then the element will surely pass through the Bloom filter. However, if the key value is not in  $S$ , it might still pass. We need to understand how to calculate the probability of a *false positive*, as a function of  $n$ , the bit-array length,  $m$  the number of members of  $S$ , and  $k$ , the number of hash functions.

The model to use is throwing darts at targets. Suppose we have  $x$  targets and  $y$  darts. Any dart is equally likely to hit any target. After throwing the darts, how many targets can we expect to be hit at least once? The analysis is similar to the analysis in Section 3.4.2, and goes as follows:

- The probability that a given dart will not hit a given target is  $(x - 1)/x$ .
- The probability that none of the  $y$  darts will hit a given target is  $(\frac{x-1}{x})^y$ . We can write this expression as  $(1 - \frac{1}{x})^{x(\frac{y}{x})}$ .
- Using the approximation  $(1 - \epsilon)^{1/\epsilon} = 1/e$  for small  $\epsilon$  (recall Section 1.3.5), we conclude that the probability that none of the  $y$  darts hit a given target is  $e^{-y/x}$ .

**Example 4.3:** Consider the running example of Section 4.3.1. We can use the above calculation to get the true expected number of 1's in the bit array. Think of each bit as a target, and each member of  $S$  as a dart. Then the probability that a given bit will be 1 is the probability that the corresponding target will be hit by one or more darts. Since there are one billion members of  $S$ , we have  $y = 10^9$  darts. As there are eight billion bits, there are  $x = 8 \times 10^9$  targets. Thus, the probability that a given target is not hit is  $e^{-y/x} = e^{-1/8}$  and the probability that it *is* hit is  $1 - e^{-1/8}$ . That quantity is about 0.1175. In Section 4.3.1 we suggested that  $1/8 = 0.125$  is a good approximation, which it is, but now we have the exact calculation.  $\square$

We can apply the rule to the more general situation, where set  $S$  has  $m$  members, the array has  $n$  bits, and there are  $k$  hash functions. The number of targets is  $x = n$ , and the number of darts is  $y = km$ . Thus, the probability that a bit remains 0 is  $e^{-km/n}$ . We want the fraction of 0 bits to be fairly large, or else the probability that a nonmember of  $S$  will hash at least once to a 0 becomes too small, and there are too many false positives. For example, we might choose  $k$ , the number of hash functions to be  $n/m$  or less. Then the probability of a 0 is at least  $e^{-1}$  or 37%. In general, the probability of a false positive is the probability of a 1 bit, which is  $1 - e^{-km/n}$ , raised to the  $k$ th power, i.e.,  $(1 - e^{-km/n})^k$ .

**Example 4.4:** In Example 4.3 we found that the fraction of 1's in the array of our running example is 0.1175, and this fraction is also the probability of a false positive. That is, a nonmember of  $S$  will pass through the filter if it hashes to a 1, and the probability of it doing so is 0.1175.

Suppose we used the same  $S$  and the same array, but used two different hash functions. This situation corresponds to throwing two billion darts at eight billion targets, and the probability that a bit remains 0 is  $e^{-1/4}$ . In order to be a false positive, a nonmember of  $S$  must hash twice to bits that are 1, and this probability is  $(1 - e^{-1/4})^2$ , or approximately 0.0493. Thus, adding a second hash function for our running example is an improvement, reducing the false-positive rate from 0.1175 to 0.0493.  $\square$

#### 4.3.4 Exercises for Section 4.3

**Exercise 4.3.1:** For the situation of our running example (8 billion bits, 1 billion members of the set  $S$ ), calculate the false-positive rate if we use three hash functions? What if we use four hash functions?

**! Exercise 4.3.2:** Suppose we have  $n$  bits of memory available, and our set  $S$  has  $m$  members. Instead of using  $k$  hash functions, we could divide the  $n$  bits into  $k$  arrays, and hash once to each array. As a function of  $n$ ,  $m$ , and  $k$ , what is the probability of a false positive? How does it compare with using  $k$  hash functions into a single array?

**!! Exercise 4.3.3:** As a function of  $n$ , the number of bits and  $m$  the number of members in the set  $S$ , what number of hash functions minimizes the false-positive rate?

## 4.4 Counting Distinct Elements in a Stream

In this section we look at a third simple kind of processing we might want to do on a stream. As with the previous examples – sampling and filtering – it is somewhat tricky to do what we want in a reasonable amount of main memory, so we use a variety of hashing and a randomized algorithm to get approximately what we want with little space needed per stream.

### 4.4.1 The Count-Distinct Problem

Suppose stream elements are chosen from some universal set. We would like to know how many different elements have appeared in the stream, counting either from the beginning of the stream or from some known time in the past.

**Example 4.5:** As a useful example of this problem, consider a Web site gathering statistics on how many unique users it has seen in each given month. The universal set is the set of logins for that site, and a stream element is generated each time someone logs in. This measure is appropriate for a site like Amazon, where the typical user logs in with their unique login name.

A similar problem is a Web site like Google that does not require login to issue a search query, and may be able to identify users only by the IP address from which they send the query. There are about 4 billion IP addresses,<sup>2</sup> sequences of four 8-bit bytes will serve as the universal set in this case.  $\square$

The obvious way to solve the problem is to keep in main memory a list of all the elements seen so far in the stream. Keep them in an efficient search structure such as a hash table or search tree, so one can quickly add new elements and check whether or not the element that just arrived on the stream was already seen. As long as the number of distinct elements is not too great, this structure can fit in main memory and there is little problem obtaining an exact answer to the question how many distinct elements appear in the stream.

However, if the number of distinct elements is too great, or if there are too many streams that need to be processed at once (e.g., Yahoo! wants to count the number of unique users viewing each of its pages in a month), then we cannot store the needed data in main memory. There are several options. We could use more machines, each machine handling only one or several of the streams. We could store most of the data structure in secondary memory and batch stream elements so whenever we brought a disk block to main memory there would be many tests and updates to be performed on the data in that block. Or we could use the strategy to be discussed in this section, where we

---

<sup>2</sup>At least that will be the case until IPv6 becomes the norm.

only estimate the number of distinct elements but use much less memory than the number of distinct elements.

#### 4.4.2 The Flajolet-Martin Algorithm

It is possible to estimate the number of distinct elements by hashing the elements of the universal set to a bit-string that is sufficiently long. The length of the bit-string must be sufficient that there are more possible results of the hash function than there are elements of the universal set. For example, 64 bits is sufficient to hash URL's. We shall pick many different hash functions and hash each element of the stream using these hash functions. The important property of a hash function is that when applied to the same element, it always produces the same result. Notice that this property was also essential for the sampling technique of Section 4.2.

The idea behind the Flajolet-Martin Algorithm is that the more different elements we see in the stream, the more different hash-values we shall see. As we see more different hash-values, it becomes more likely that one of these values will be "unusual." The particular unusual property we shall exploit is that the value ends in many 0's, although many other options exist.

Whenever we apply a hash function  $h$  to a stream element  $a$ , the bit string  $h(a)$  will end in some number of 0's, possibly none. Call this number the *tail length* for  $a$  and  $h$ . Let  $R$  be the maximum tail length of any  $a$  seen so far in the stream. Then we shall use estimate  $2^R$  for the number of distinct elements seen in the stream.

This estimate makes intuitive sense. The probability that a given stream element  $a$  has  $h(a)$  ending in at least  $r$  0's is  $2^{-r}$ . Suppose there are  $m$  distinct elements in the stream. Then the probability that none of them has tail length at least  $r$  is  $(1 - 2^{-r})^m$ . This sort of expression should be familiar by now. We can rewrite it as  $((1 - 2^{-r})^{2^r})^{m2^{-r}}$ . Assuming  $r$  is reasonably large, the inner expression is of the form  $(1 - \epsilon)^{1/\epsilon}$ , which is approximately  $1/e$ . Thus, the probability of not finding a stream element with as many as  $r$  0's at the end of its hash value is  $e^{-m2^{-r}}$ . We can conclude:

1. If  $m$  is much larger than  $2^r$ , then the probability that we shall find a tail of length at least  $r$  approaches 1.
2. If  $m$  is much less than  $2^r$ , then the probability of finding a tail length at least  $r$  approaches 0.

We conclude from these two points that the proposed estimate of  $m$ , which is  $2^R$  (recall  $R$  is the largest tail length for any stream element) is unlikely to be either much too high or much too low.

### 4.4.3 Combining Estimates

Unfortunately, there is a trap regarding the strategy for combining the estimates of  $m$ , the number of distinct elements, that we obtain by using many different hash functions. Our first assumption would be that if we take the average of the values  $2^R$  that we get from each hash function, we shall get a value that approaches the true  $m$ , the more hash functions we use. However, that is not the case, and the reason has to do with the influence an overestimate has on the average.

Consider a value of  $r$  such that  $2^r$  is much larger than  $m$ . There is some probability  $p$  that we shall discover  $r$  to be the largest number of 0's at the end of the hash value for any of the  $m$  stream elements. Then the probability of finding  $r+1$  to be the largest number of 0's instead is at least  $p/2$ . However, if we do increase by 1 the number of 0's at the end of a hash value, the value of  $2^R$  doubles. Consequently, the contribution from each possible large  $R$  to the expected value of  $2^R$  grows as  $R$  grows, and the expected value of  $2^R$  is actually infinite.<sup>3</sup>

Another way to combine estimates is to take the median of all estimates. The median is not affected by the occasional outsized value of  $2^R$ , so the worry described above for the average should not carry over to the median. Unfortunately, the median suffers from another defect: it is always a power of 2. Thus, no matter how many hash functions we use, should the correct value of  $m$  be between two powers of 2, say 400, then it will be impossible to obtain a close estimate.

There is a solution to the problem, however. We can combine the two methods. First, group the hash functions into small groups, and take their average. Then, take the median of the averages. It is true that an occasional outsized  $2^R$  will bias some of the groups and make them too large. However, taking the median of group averages will reduce the influence of this effect almost to nothing. Moreover, if the groups themselves are large enough, then the averages can be essentially any number, which enables us to approach the true value  $m$  as long as we use enough hash functions. In order to guarantee that any possible average can be obtained, groups should be of size at least a small multiple of  $\log_2 m$ .

### 4.4.4 Space Requirements

Observe that as we read the stream it is not necessary to store the elements seen. The only thing we need to keep in main memory is one integer per hash function; this integer records the largest tail length seen so far for that hash function and any stream element. If we are processing only one stream, we could use millions of hash functions, which is far more than we need to get a

---

<sup>3</sup>Technically, since the hash value is a bit-string of finite length, there is no contribution to  $2^R$  for  $R$ 's that are larger than the length of the hash value. However, this effect is not enough to avoid the conclusion that the expected value of  $2^R$  is much too large.

close estimate. Only if we are trying to process many streams at the same time would main memory constrain the number of hash functions we could associate with any one stream. In practice, the time it takes to compute hash values for each stream element would be the more significant limitation on the number of hash functions we use.

#### 4.4.5 Exercises for Section 4.4

**Exercise 4.4.1:** Suppose our stream consists of the integers 3, 1, 4, 1, 5, 9, 2, 6, 5. Our hash functions will all be of the form  $h(x) = ax + b \bmod 32$  for some  $a$  and  $b$ . You should treat the result as a 5-bit binary integer. Determine the tail length for each stream element and the resulting estimate of the number of distinct elements if the hash function is:

- (a)  $h(x) = 2x + 1 \bmod 32$ .
- (b)  $h(x) = 3x + 7 \bmod 32$ .
- (c)  $h(x) = 4x \bmod 32$ .

**! Exercise 4.4.2:** Do you see any problems with the choice of hash functions in Exercise 4.4.1? What advice could you give someone who was going to use a hash function of the form  $h(x) = ax + b \bmod 2^k$ ?

## 4.5 Estimating Moments

In this section we consider a generalization of the problem of counting distinct elements in a stream. The problem, called computing “moments,” involves the distribution of frequencies of different elements in the stream. We shall define moments of all orders and concentrate on computing second moments, from which the general algorithm for all moments is a simple extension.

### 4.5.1 Definition of Moments

Suppose a stream consists of elements chosen from a universal set. Assume the universal set is ordered so we can speak of the  $i$ th element for any  $i$ . Let  $m_i$  be the number of occurrences of the  $i$ th element for any  $i$ . Then the  $k$ th-order moment (or just  $k$ th moment) of the stream is the sum over all  $i$  of  $(m_i)^k$ .

**Example 4.6:** The 0th moment is the sum of 1 for each  $m_i$  that is greater than 0.<sup>4</sup> That is, the 0th moment is a count of the number of distinct elements in the stream. We can use the method of Section 4.4 to estimate the 0th moment of a stream.

---

<sup>4</sup>Technically, since  $m_i$  could be 0 for some elements in the universal set, we need to make explicit in the definition of “moment” that  $0^0$  is taken to be 0. For moments 1 and above, the contribution of  $m_i$ ’s that are 0 is surely 0.

The 1st moment is the sum of the  $m_i$ 's, which must be the length of the stream. Thus, first moments are especially easy to compute; just count the length of the stream seen so far.

The second moment is the sum of the squares of the  $m_i$ 's. It is sometimes called the *surprise number*, since it measures how uneven the distribution of elements in the stream is. To see the distinction, suppose we have a stream of length 100, in which eleven different elements appear. The most even distribution of these eleven elements would have one appearing 10 times and the other ten appearing 9 times each. In this case, the surprise number is  $10^2 + 10 \times 9^2 = 910$ . At the other extreme, one of the eleven elements could appear 90 times and the other ten appear 1 time each. Then, the surprise number would be  $90^2 + 10 \times 1^2 = 8110$ .  $\square$

As in Section 4.4, there is no problem computing moments of any order if we can afford to keep in main memory a count for each element that appears in the stream. However, also as in that section, if we cannot afford to use that much memory, then we need to estimate the  $k$ th moment by keeping a limited number of values in main memory and computing an estimate from these values. For the case of distinct elements, each of these values were counts of the longest tail produced by a single hash function. We shall see another form of value that is useful for second and higher moments.

### 4.5.2 The Alon-Matias-Szegedy Algorithm for Second Moments

For now, let us assume that a stream has a particular length  $n$ . We shall show how to deal with growing streams in the next section. Suppose we do not have enough space to count all the  $m_i$ 's for all the elements of the stream. We can still estimate the second moment of the stream using a limited amount of space; the more space we use, the more accurate the estimate will be. We compute some number of *variables*. For each variable  $X$ , we store:

1. A particular element of the universal set, which we refer to as  $X.\text{element}$ , and
2. An integer  $X.\text{value}$ , which is the *value* of the variable. To determine the value of a variable  $X$ , we choose a position in the stream between 1 and  $n$ , uniformly and at random. Set  $X.\text{element}$  to be the element found there, and initialize  $X.\text{value}$  to 1. As we read the stream, add 1 to  $X.\text{value}$  each time we encounter another occurrence of  $X.\text{element}$ .

**Example 4.7:** Suppose the stream is  $a, b, c, b, d, a, c, d, a, b, d, c, a, a, b$ . The length of the stream is  $n = 15$ . Since  $a$  appears 5 times,  $b$  appears 4 times, and  $c$  and  $d$  appear three times each, the second moment for the stream is  $5^2 + 4^2 + 3^2 + 3^2 = 59$ . Suppose we keep three variables,  $X_1$ ,  $X_2$ , and  $X_3$ . Also,

assume that at “random” we pick the 3rd, 8th, and 13th positions to define these three variables.

When we reach position 3, we find element  $c$ , so we set  $X_1.element = c$  and  $X_1.value = 1$ . Position 4 holds  $b$ , so we do not change  $X_1$ . Likewise, nothing happens at positions 5 or 6. At position 7, we see  $c$  again, so we set  $X_1.value = 2$ .

At position 8 we find  $d$ , and so set  $X_2.element = d$  and  $X_2.value = 1$ . Positions 9 and 10 hold  $a$  and  $b$ , so they do not affect  $X_1$  or  $X_2$ . Position 11 holds  $d$  so we set  $X_2.value = 2$ , and position 12 holds  $c$  so we set  $X_1.value = 3$ . At position 13, we find element  $a$ , and so set  $X_3.element = a$  and  $X_3.value = 1$ . Then, at position 14 we see another  $a$  and so set  $X_3.value = 2$ . Position 15, with element  $b$  does not affect any of the variables, so we are done, with final values  $X_1.value = 3$  and  $X_2.value = X_3.value = 2$ .  $\square$

We can derive an estimate of the second moment from any variable  $X$ . This estimate is  $n(2X.value - 1)$ .

**Example 4.8:** Consider the three variables from Example 4.7. From  $X_1$  we derive the estimate  $n(2X_1.value - 1) = 15 \times (2 \times 3 - 1) = 75$ . The other two variables,  $X_2$  and  $X_3$ , each have value 2 at the end, so their estimates are  $15 \times (2 \times 2 - 1) = 45$ . Recall that the true value of the second moment for this stream is 59. On the other hand, the average of the three estimates is 55, a fairly close approximation.  $\square$

### 4.5.3 Why the Alon-Matias-Szegedy Algorithm Works

We can prove that the expected value of any variable constructed as in Section 4.5.2 is the second moment of the stream from which it is constructed. Some notation will make the argument easier to follow. Let  $e(i)$  be the stream element that appears at position  $i$  in the stream, and let  $c(i)$  be the number of times element  $e(i)$  appears in the stream among positions  $i, i+1, \dots, n$ .

**Example 4.9:** Consider the stream of Example 4.7.  $e(6) = a$ , since the 6th position holds  $a$ . Also,  $c(6) = 4$ , since  $a$  appears at positions 9, 13, and 14, as well as at position 6. Note that  $a$  also appears at position 1, but that fact does not contribute to  $c(6)$ .  $\square$

The expected value of  $n(2X.value - 1)$  is the average over all positions  $i$  between 1 and  $n$  of  $n(2c(i) - 1)$ , that is

$$E(n(2X.value - 1)) = \frac{1}{n} \sum_{i=1}^n n(2c(i) - 1)$$

We can simplify the above by canceling factors  $1/n$  and  $n$ , to get

$$E(n(2X.value - 1)) = \sum_{i=1}^n (2c(i) - 1)$$

However, to make sense of the formula, we need to change the order of summation by grouping all those positions that have the same element. For instance, concentrate on some element  $a$  that appears  $m_a$  times in the stream. The term for the last position in which  $a$  appears must be  $2 \times 1 - 1 = 1$ . The term for the next-to-last position in which  $a$  appears is  $2 \times 2 - 1 = 3$ . The positions with  $a$  before that yield terms 5, 7, and so on, up to  $2m_a - 1$ , which is the term for the first position in which  $a$  appears. That is, the formula for the expected value of  $2X.value - 1$  can be written:

$$E(n(2X.value - 1)) = \sum_a 1 + 3 + 5 + \cdots + (2m_a - 1)$$

Note that  $1 + 3 + 5 + \cdots + (2m_a - 1) = (m_a)^2$ . The proof is an easy induction on the number of terms in the sum. Thus,  $E(n(2X.value - 1)) = \sum_a (m_a)^2$ , which is the definition of the second moment.

#### 4.5.4 Higher-Order Moments

We estimate  $k$ th moments, for  $k > 2$ , in essentially the same way as we estimate second moments. The only thing that changes is the way we derive an estimate from a variable. In Section 4.5.2 we used the formula  $n(2v - 1)$  to turn a value  $v$ , the count of the number of occurrences of some particular stream element  $a$ , into an estimate of the second moment. Then, in Section 4.5.3 we saw why this formula works: the terms  $2v - 1$ , for  $v = 1, 2, \dots, m$  sum to  $m^2$ , where  $m$  is the number of times  $a$  appears in the stream.

Notice that  $2v - 1$  is the difference between  $v^2$  and  $(v - 1)^2$ . Suppose we wanted the third moment rather than the second. Then all we have to do is replace  $2v - 1$  by  $v^3 - (v - 1)^3 = 3v^2 - 3v + 1$ . Then  $\sum_{v=1}^m 3v^2 - 3v + 1 = m^3$ , so we can use as our estimate of the third moment the formula  $n(3v^2 - 3v + 1)$ , where  $v = X.value$  is the value associated with some variable  $X$ . More generally, we can estimate  $k$ th moments for any  $k \geq 2$  by turning value  $v = X.value$  into  $n(v^k - (v - 1)^k)$ .

#### 4.5.5 Dealing With Infinite Streams

Technically, the estimate we used for second and higher moments assumes that  $n$ , the stream length, is a constant. In practice,  $n$  grows with time. That fact, by itself, doesn't cause problems, since we store only the values of variables and multiply some function of that value by  $n$  when it is time to estimate the moment. If we count the number of stream elements seen and store this value, which only requires  $\log n$  bits, then we have  $n$  available whenever we need it.

A more serious problem is that we must be careful how we select the positions for the variables. If we do this selection once and for all, then as the stream gets longer, we are biased in favor of early positions, and the estimate of the moment will be too large. On the other hand, if we wait too long to pick positions, then

early in the stream we do not have many variables and so will get an unreliable estimate.

The proper technique is to maintain as many variables as we can store at all times, and to throw some out as the stream grows. The discarded variables are replaced by new ones, in such a way that at all times, the probability of picking any one position for a variable is the same as that of picking any other position. Suppose we have space to store  $s$  variables. Then the first  $s$  positions of the stream are each picked as the position of one of the  $s$  variables.

Inductively, suppose we have seen  $n$  stream elements, and the probability of any particular position being the position of a variable is uniform, that is  $s/n$ . When the  $(n+1)$ st element arrives, pick that position with probability  $s/(n+1)$ . If not picked, then the  $s$  variables keep their same positions. However, if the  $(n+1)$ st position is picked, then throw out one of the current  $s$  variables, with equal probability. Replace the one discarded by a new variable whose element is the one at position  $n+1$  and whose value is 1.

Surely, the probability that position  $n+1$  is selected for a variable is what it should be:  $s/(n+1)$ . However, the probability of every other position also is  $s/(n+1)$ , as we can prove by induction on  $n$ . By the inductive hypothesis, before the arrival of the  $(n+1)$ st stream element, this probability was  $s/n$ . With probability  $1 - s/(n+1)$  the  $(n+1)$ st position will not be selected, and the probability of each of the first  $n$  positions remains  $s/n$ . However, with probability  $s/(n+1)$ , the  $(n+1)$ st position is picked, and the probability for each of the first  $n$  positions is reduced by factor  $(s-1)/s$ . Considering the two cases, the probability of selecting each of the first  $n$  positions is

$$(1 - \frac{s}{n+1})\left(\frac{s}{n}\right) + \left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)\left(\frac{s}{n}\right)$$

This expression simplifies to

$$(1 - \frac{s}{n+1})\left(\frac{s}{n}\right) + \left(\frac{s-1}{n+1}\right)\left(\frac{s}{n}\right)$$

and then to

$$\left( (1 - \frac{s}{n+1}) + \left(\frac{s-1}{n+1}\right) \right) \left(\frac{s}{n}\right)$$

which in turn simplifies to

$$\left(\frac{n}{n+1}\right)\left(\frac{s}{n}\right) = \frac{s}{n+1}$$

Thus, we have shown by induction on the stream length  $n$  that all positions have equal probability  $s/n$  of being chosen as the position of a variable.

#### 4.5.6 Exercises for Section 4.5

**Exercise 4.5.1:** Compute the surprise number (second moment) for the stream 3, 1, 4, 1, 3, 4, 2, 1, 2. What is the third moment of this stream?

### A General Stream-Sampling Problem

Notice that the technique described in Section 4.5.5 actually solves a more general problem. It gives us a way to maintain a sample of  $s$  stream elements so that at all times, all stream elements are equally likely to be selected for the sample.

As an example of where this technique can be useful, recall that in Section 4.2 we arranged to select all the tuples of a stream having key value in a randomly selected subset. Suppose that, as time goes on, there are too many tuples associated with any one key. We can arrange to limit the number of tuples for any key  $K$  to a fixed constant  $s$  by using the technique of Section 4.5.5 whenever a new tuple for key  $K$  arrives.

**Exercise 4.5.2:** If a stream has  $n$  elements, of which  $m$  are distinct, what are the minimum and maximum possible surprise number, as a function of  $m$  and  $n$ ?

**Exercise 4.5.3:** Suppose we are given the stream of Exercise 4.5.1, to which we apply the Alon-Matias-Szegedy Algorithm to estimate the surprise number. For each possible value of  $i$ , if  $X_i$  is a variable starting position  $i$ , what is the value of  $X_i.value$ ?

**Exercise 4.5.4:** Repeat Exercise 4.5.3 if the intent of the variables is to compute third moments. What is the value of each variable at the end? What estimate of the third moment do you get from each variable? How does the average of these estimates compare with the true value of the third moment?

**Exercise 4.5.5:** Prove by induction on  $m$  that  $1 + 3 + 5 + \dots + (2m - 1) = m^2$ .

**Exercise 4.5.6:** If we wanted to compute fourth moments, how would we convert  $X.value$  to an estimate of the fourth moment?

## 4.6 Counting Ones in a Window

We now turn our attention to counting problems for streams. Suppose we have a window of length  $N$  on a binary stream. We want at all times to be able to answer queries of the form “how many 1’s are there in the last  $k$  bits?” for any  $k \leq N$ . As in previous sections, we focus on the situation where we cannot afford to store the entire window. After showing an approximate algorithm for the binary case, we discuss how this idea can be extended to summing numbers.

### 4.6.1 The Cost of Exact Counts

To begin, suppose we want to be able to count exactly the number of 1's in the last  $k$  bits for any  $k \leq N$ . Then we claim it is necessary to store all  $N$  bits of the window, as any representation that used fewer than  $N$  bits could not work. In proof, suppose we have a representation that uses fewer than  $N$  bits to represent the  $N$  bits in the window. Since there are  $2^N$  sequences of  $N$  bits, but fewer than  $2^N$  representations, there must be two different bit strings  $w$  and  $x$  that have the same representation. Since  $w \neq x$ , they must differ in at least one bit. Let the last  $k - 1$  bits of  $w$  and  $x$  agree, but let them differ on the  $k$ th bit from the right end.

**Example 4.10:** If  $w = 0101$  and  $x = 1010$ , then  $k = 1$ , since scanning from the right, they first disagree at position 1. If  $w = 1001$  and  $x = 0101$ , then  $k = 3$ , because they first disagree at the third position from the right.  $\square$

Suppose the data representing the contents of the window is whatever sequence of bits represents both  $w$  and  $x$ . Ask the query “how many 1's are in the last  $k$  bits?” The query-answering algorithm will produce the same answer, whether the window contains  $w$  or  $x$ , because the algorithm can only see their representation. But the correct answers are surely different for these two bit-strings. Thus, we have proved that we must use at least  $N$  bits to answer queries about the last  $k$  bits for any  $k$ .

In fact, we need  $N$  bits, even if the only query we can ask is “how many 1's are in the entire window of length  $N$ ?”. The argument is similar to that used above. Suppose we use fewer than  $N$  bits to represent the window, and therefore we can find  $w$ ,  $x$ , and  $k$  as above. It might be that  $w$  and  $x$  have the same number of 1's, as they did in both cases of Example 4.10. However, if we follow the current window by any  $N - k$  bits, we will have a situation where the true window contents resulting from  $w$  and  $x$  are identical except for the leftmost bit, and therefore, their counts of 1's are unequal. However, since the representations of  $w$  and  $x$  are the same, the representation of the window must still be the same if we feed the same bit sequence to these representations. Thus, we can force the answer to the query “how many 1's in the window?” to be incorrect for one of the two possible window contents.

### 4.6.2 The Datar-Gionis-Indyk-Motwani Algorithm

We shall present the simplest case of an algorithm called DGIM. This version of the algorithm uses  $O(\log^2 N)$  bits to represent a window of  $N$  bits, and allows us to estimate the number of 1's in the window with an error of no more than 50%. Later, we shall discuss an improvement of the method that limits the error to any fraction  $\epsilon > 0$ , and still uses only  $O(\log^2 N)$  bits (although with a constant factor that grows as  $\epsilon$  shrinks).

To begin, each bit of the stream has a *timestamp*, the position in which it arrives. The first bit has timestamp 1, the second has timestamp 2, and so on.

Since we only need to distinguish positions within the window of length  $N$ , we shall represent timestamps modulo  $N$ , so they can be represented by  $\log_2 N$  bits. If we also store the total number of bits ever seen in the stream (i.e., the most recent timestamp) modulo  $N$ , then we can determine from a timestamp modulo  $N$  where in the current window the bit with that timestamp is.

We divide the window into *buckets*,<sup>5</sup> consisting of:

1. The timestamp of its right (most recent) end.
2. The number of 1's in the bucket. This number must be a power of 2, and we refer to the number of 1's as the *size* of the bucket.

To represent a bucket, we need  $\log_2 N$  bits to represent the timestamp (modulo  $N$ ) of its right end. To represent the number of 1's we only need  $\log_2 \log_2 N$  bits. The reason is that we know this number  $i$  is a power of 2, say  $2^j$ , so we can represent  $i$  by coding  $j$  in binary. Since  $j$  is at most  $\log_2 N$ , it requires  $\log_2 \log_2 N$  bits. Thus,  $O(\log N)$  bits suffice to represent a bucket.

There are six rules that must be followed when representing a stream by buckets.

- The right end of a bucket is always a position with a 1.
- Every position with a 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.
- Buckets cannot decrease in size as we move to the left (back in time).

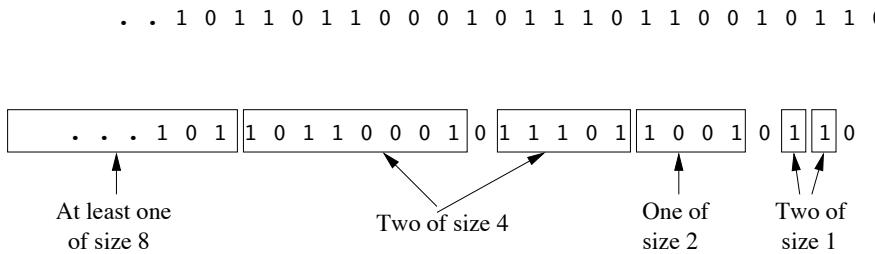


Figure 4.2: A bit-stream divided into buckets following the DGIM rules

---

<sup>5</sup>Do not confuse these “buckets” with the “buckets” discussed in connection with hashing.

**Example 4.11:** Figure 4.2 shows a bit stream divided into buckets in a way that satisfies the DGIM rules. At the right (most recent) end we see two buckets of size 1. To its left we see one bucket of size 2. Note that this bucket covers four positions, but only two of them are 1. Proceeding left, we see two buckets of size 4, and we suggest that a bucket of size 8 exists further left.

Notice that it is OK for some 0's to lie between buckets. Also, observe from Fig. 4.2 that the buckets do not overlap; there are one or two of each size up to the largest size, and sizes only increase moving left.  $\square$

In the next sections, we shall explain the following about the DGIM algorithm:

1. Why the number of buckets representing a window must be small.
2. How to estimate the number of 1's in the last  $k$  bits for any  $k$ , with an error no greater than 50%.
3. How to maintain the DGIM conditions as new bits enter the stream.

### 4.6.3 Storage Requirements for the DGIM Algorithm

We observed that each bucket can be represented by  $O(\log N)$  bits. If the window has length  $N$ , then there are no more than  $N$  1's, surely. Suppose the largest bucket is of size  $2^j$ . Then  $j$  cannot exceed  $\log_2 N$ , or else there are more 1's in this bucket than there are 1's in the entire window. Thus, there are at most two buckets of all sizes from  $\log_2 N$  down to 1, and no buckets of larger sizes.

We conclude that there are  $O(\log N)$  buckets. Since each bucket can be represented in  $O(\log N)$  bits, the total space required for all the buckets representing a window of size  $N$  is  $O(\log^2 N)$ .

### 4.6.4 Query Answering in the DGIM Algorithm

Suppose we are asked how many 1's there are in the last  $k$  bits of the window, for some  $1 \leq k \leq N$ . Find the bucket  $b$  with the earliest timestamp that includes at least some of the  $k$  most recent bits. Estimate the number of 1's to be the sum of the sizes of all the buckets to the right (more recent) than bucket  $b$ , plus half the size of  $b$  itself.

**Example 4.12:** Suppose the stream is that of Fig. 4.2, and  $k = 10$ . Then the query asks for the number of 1's in the ten rightmost bits, which happen to be 0110010110. Let the current timestamp (time of the rightmost bit) be  $t$ . Then the two buckets with one 1, having timestamps  $t - 1$  and  $t - 2$  are completely included in the answer. The bucket of size 2, with timestamp  $t - 4$ , is also completely included. However, the rightmost bucket of size 4, with timestamp  $t - 8$  is only partly included. We know it is the last bucket to contribute to the answer, because the next bucket to its left has timestamp less than  $t - 9$  and

thus is completely out of the window. On the other hand, we know the buckets to its right are completely inside the range of the query because of the existence of a bucket to their left with timestamp  $t - 9$  or greater.

Our estimate of the number of 1's in the last ten positions is thus 6. This number is the two buckets of size 1, the bucket of size 2, and half the bucket of size 4 that is partially within range. Of course the correct answer is 5.  $\square$

Suppose the above estimate of the answer to a query involves a bucket  $b$  of size  $2^j$  that is partially within the range of the query. Let us consider how far from the correct answer  $c$  our estimate could be. There are two cases: the estimate could be larger or smaller than  $c$ .

*Case 1:* The estimate is less than  $c$ . In the worst case, all the 1's of  $b$  are actually within the range of the query, so the estimate misses half bucket  $b$ , or  $2^{j-1}$  1's. But in this case,  $c$  is at least  $2^j$ ; in fact it is at least  $2^{j+1} - 1$ , since there is at least one bucket of each of the sizes  $2^{j-1}, 2^{j-2}, \dots, 1$ . We conclude that our estimate is at least 50% of  $c$ .

*Case 2:* The estimate is greater than  $c$ . In the worst case, only the rightmost bit of bucket  $b$  is within range, and there is only one bucket of each of the sizes smaller than  $b$ . Then  $c = 1 + 2^{j-1} + 2^{j-2} + \dots + 1 = 2^j$  and the estimate we give is  $2^{j-1} + 2^{j-1} + 2^{j-2} + \dots + 1 = 2^j + 2^{j-1} - 1$ . We see that the estimate is no more than 50% greater than  $c$ .

#### 4.6.5 Maintaining the DGIM Conditions

Suppose we have a window of length  $N$  properly represented by buckets that satisfy the DGIM conditions. When a new bit comes in, we may need to modify the buckets, so they continue to represent the window and continue to satisfy the DGIM conditions. First, whenever a new bit enters:

- Check the leftmost (earliest) bucket. If its timestamp has now reached the current timestamp minus  $N$ , then this bucket no longer has any of its 1's in the window. Therefore, drop it from the list of buckets.

Now, we must consider whether the new bit is 0 or 1. If it is 0, then no further change to the buckets is needed. If the new bit is a 1, however, we may need to make several changes. First:

- Create a new bucket with the current timestamp and size 1.

If there was only one bucket of size 1, then nothing more needs to be done. However, if there are now three buckets of size 1, that is one too many. We fix this problem by combining the leftmost (earliest) two buckets of size 1.

- To combine any two adjacent buckets of the same size, replace them by one bucket of twice the size. The timestamp of the new bucket is the timestamp of the rightmost (later in time) of the two buckets.

Combining two buckets of size 1 may create a third bucket of size 2. If so, we combine the leftmost two buckets of size 2 into a bucket of size 4. That, in turn, may create a third bucket of size 4, and if so we combine the leftmost two into a bucket of size 8. This process may ripple through the bucket sizes, but there are at most  $\log_2 N$  different sizes, and the combination of two adjacent buckets of the same size only requires constant time. As a result, any new bit can be processed in  $O(\log N)$  time.

**Example 4.13:** Suppose we start with the buckets of Fig. 4.2 and a 1 enters. First, the leftmost bucket evidently has not fallen out of the window, so we do not drop any buckets. We create a new bucket of size 1 with the current timestamp, say  $t$ . There are now three buckets of size 1, so we combine the leftmost two. They are replaced with a single bucket of size 2. Its timestamp is  $t - 2$ , the timestamp of the bucket on the right (i.e., the rightmost bucket that actually appears in Fig. 4.2).

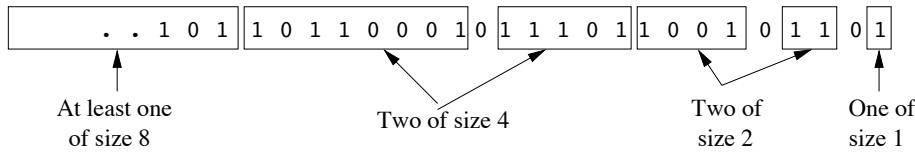


Figure 4.3: Modified buckets after a new 1 arrives in the stream

There are now two buckets of size 2, but that is allowed by the DGIM rules. Thus, the final sequence of buckets after the addition of the 1 is as shown in Fig. 4.3.  $\square$

#### 4.6.6 Reducing the Error

Instead of allowing either one or two of each size bucket, suppose we allow either  $r - 1$  or  $r$  of each of the exponentially growing sizes  $1, 2, 4, \dots$ , for some integer  $r > 2$ . In order to represent any possible number of 1's, we must relax this condition for the buckets of size 1 and buckets of the largest size present; there may be any number, from 1 to  $r$ , of buckets of these sizes.

The rule for combining buckets is essentially the same as in Section 4.6.5. If we get  $r + 1$  buckets of size  $2^j$ , combine the leftmost two into a bucket of size  $2^{j+1}$ . That may, in turn, cause there to be  $r + 1$  buckets of size  $2^{j+1}$ , and if so we continue combining buckets of larger sizes.

The argument used in Section 4.6.4 can also be used here. However, because there are more buckets of smaller sizes, we can get a stronger bound on the error. We saw there that the largest relative error occurs when only one 1 from the leftmost bucket  $b$  is within the query range, and we therefore overestimate the true count. Suppose bucket  $b$  is of size  $2^j$ . Then the true count is at least

### Bucket Sizes and Ripple-Carry Adders

There is a pattern to the distribution of bucket sizes as we execute the basic algorithm of Section 4.6.5. Think of two buckets of size  $2^j$  as a "1" in position  $j$  and one bucket of size  $2^j$  as a "0" in that position. Then as 1's arrive in the stream, the bucket sizes after each 1 form consecutive binary integers. The occasional long sequences of bucket combinations are analogous to the occasional long rippling of carries as we go from an integer like 101111 to 110000.

$1 + (r - 1)(2^{j-1} + 2^{j-2} + \dots + 1) = 1 + (r - 1)(2^j - 1)$ . The overestimate is  $2^{j-1} - 1$ . Thus, the fractional error is

$$\frac{2^{j-1} - 1}{1 + (r - 1)(2^j - 1)}$$

No matter what  $j$  is, this fraction is upper bounded by  $1/(r - 1)$ . Thus, by picking  $r$  sufficiently large, we can limit the error to any desired  $\epsilon > 0$ .

#### 4.6.7 Extensions to the Counting of Ones

It is natural to ask whether we can extend the technique of this section to handle aggregations more general than counting 1's in a binary stream. An obvious direction to look is to consider streams of integers and ask if we can estimate the sum of the last  $k$  integers for any  $1 \leq k \leq N$ , where  $N$ , as usual, is the window size.

It is unlikely that we can use the DGIM approach to streams containing both positive and negative integers. We could have a stream containing both very large positive integers and very large negative integers, but with a sum in the window that is very close to 0. Any imprecision in estimating the values of these large integers would have a huge effect on the estimate of the sum, and so the fractional error could be unbounded.

For example, suppose we broke the stream into buckets as we have done, but represented the bucket by the sum of the integers therein, rather than the count of 1's. If  $b$  is the bucket that is partially within the query range, it could be that  $b$  has, in its first half, very large negative integers and in its second half, equally large positive integers, with a sum of 0. If we estimate the contribution of  $b$  by half its sum, that contribution is essentially 0. But the actual contribution of that part of bucket  $b$  that is in the query range could be anything from 0 to the sum of all the positive integers. This difference could be far greater than the actual query answer, and so the estimate would be meaningless.

On the other hand, some other extensions involving integers do work. Suppose that the stream consists of only positive integers in the range 1 to  $2^m$  for

some  $m$ . We can treat each of the  $m$  bits of each integer as if it were a separate stream. We then use the DGIM method to count the 1's in each bit. Suppose the count of the  $i$ th bit (assuming bits count from the low-order end, starting at 0) is  $c_i$ . Then the sum of the integers is

$$\sum_{i=0}^{m-1} c_i 2^i$$

If we use the technique of Section 4.6.6 to estimate each  $c_i$  with fractional error at most  $\epsilon$ , then the estimate of the true sum has error at most  $\epsilon$ . The worst case occurs when all the  $c_i$ 's are overestimated or all are underestimated by the same fraction.

#### 4.6.8 Exercises for Section 4.6

**Exercise 4.6.1:** Suppose the window is as shown in Fig. 4.2. Estimate the number of 1's the the last  $k$  positions, for  $k =$  (a) 5 (b) 15. In each case, how far off the correct value is your estimate?

! **Exercise 4.6.2:** There are several ways that the bit-stream 1001011011101 could be partitioned into buckets. Find all of them.

**Exercise 4.6.3:** Describe what happens to the buckets if three more 1's enter the window represented by Fig. 4.3. You may assume none of the 1's shown leave the window.

### 4.7 Decaying Windows

We have assumed that a sliding window held a certain tail of the stream, either the most recent  $N$  elements for fixed  $N$ , or all the elements that arrived after some time in the past. Sometimes we do not want to make a sharp distinction between recent elements and those in the distant past, but want to weight the recent elements more heavily. In this section, we consider “exponentially decaying windows,” and an application where they are quite useful: finding the most common “recent” elements.

#### 4.7.1 The Problem of Most-Common Elements

Suppose we have a stream whose elements are the movie tickets purchased all over the world, with the name of the movie as part of the element. We want to keep a summary of the stream that is the most popular movies “currently.” While the notion of “currently” is imprecise, intuitively, we want to discount the popularity of a movie like *Star Wars–Episode 4*, which sold many tickets, but most of these were sold decades ago. On the other hand, a movie that sold

$n$  tickets in each of the last 10 weeks is probably more popular than a movie that sold  $2n$  tickets last week but nothing in previous weeks.

One solution would be to imagine a bit stream for each movie. The  $i$ th bit has value 1 if the  $i$ th ticket is for that movie, and 0 otherwise. Pick a window size  $N$ , which is the number of most recent tickets that would be considered in evaluating popularity. Then, use the method of Section 4.6 to estimate the number of tickets for each movie, and rank movies by their estimated counts. This technique might work for movies, because there are only thousands of movies, but it would fail if we were instead recording the popularity of items sold at Amazon, or the rate at which different Twitter-users tweet, because there are too many Amazon products and too many tweeters. Further, it only offers approximate answers.

#### 4.7.2 Definition of the Decaying Window

An alternative approach is to redefine the question so that we are not asking for a count of 1's in a window. Rather, let us compute a smooth aggregation of all the 1's ever seen in the stream, with decaying weights, so the further back in the stream, the less weight is given. Formally, let a stream currently consist of the elements  $a_1, a_2, \dots, a_t$ , where  $a_1$  is the first element to arrive and  $a_t$  is the current element. Let  $c$  be a small constant, such as  $10^{-6}$  or  $10^{-9}$ . Define the *exponentially decaying window* for this stream to be the sum

$$\sum_{i=0}^{t-1} a_{t-i}(1 - c)^i$$

The effect of this definition is to spread out the weights of the stream elements as far back in time as the stream goes. In contrast, a fixed window with the same sum of the weights,  $1/c$ , would put equal weight 1 on each of the most recent  $1/c$  elements to arrive and weight 0 on all previous elements. The distinction is suggested by Fig. 4.4.

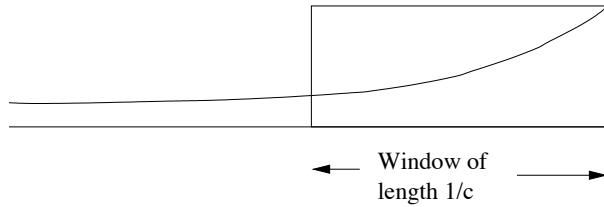


Figure 4.4: A decaying window and a fixed-length window of equal weight

It is much easier to adjust the sum in an exponentially decaying window than in a sliding window of fixed length. In the sliding window, we have to worry about the element that falls out of the window each time a new element arrives. That forces us to keep the exact elements along with the sum, or to use

an approximation scheme such as DGIM. However, when a new element  $a_{t+1}$  arrives at the stream input, all we need to do is:

1. Multiply the current sum by  $1 - c$ .
2. Add  $a_{t+1}$ .

The reason this method works is that each of the previous elements has now moved one position further from the current element, so its weight is multiplied by  $1 - c$ . Further, the weight on the current element is  $(1 - c)^0 = 1$ , so adding  $a_{t+1}$  is the correct way to include the new element's contribution.

### 4.7.3 Finding the Most Popular Elements

Let us return to the problem of finding the most popular movies in a stream of ticket sales.<sup>6</sup> We shall use an exponentially decaying window with a constant  $c$ , which you might think of as  $10^{-9}$ . That is, we approximate a sliding window holding the last one billion ticket sales. For each movie, we imagine a separate stream with a 1 each time a ticket for that movie appears in the stream, and a 0 each time a ticket for some other movie arrives. The decaying sum of the 1's measures the current popularity of the movie.

We imagine that the number of possible movies in the stream is huge, so we do not want to record values for the unpopular movies. Therefore, we establish a threshold, say  $1/2$ , so that if the popularity score for a movie goes below this number, its score is dropped from the counting. For reasons that will become obvious, the threshold must be less than 1, although it can be any number less than 1. When a new ticket arrives on the stream, do the following:

1. For each movie whose score we are currently maintaining, multiply its score by  $(1 - c)$ .
2. Suppose the new ticket is for movie  $M$ . If there is currently a score for  $M$ , add 1 to that score. If there is no score for  $M$ , create one and initialize it to 1.
3. If any score is below the threshold  $1/2$ , drop that score.

It may not be obvious that the number of movies whose scores are maintained at any time is limited. However, note that the sum of all scores is  $1/c$ . There cannot be more than  $2/c$  movies with score of  $1/2$  or more, or else the sum of the scores would exceed  $1/c$ . Thus,  $2/c$  is a limit on the number of movies being counted at any time. Of course in practice, the ticket sales would be concentrated on only a small number of movies at any time, so the number of actively counted movies would be much less than  $2/c$ .

---

<sup>6</sup>This example should be taken with a grain of salt, because, as we pointed out, there aren't enough different movies for this technique to be essential. Imagine, if you will, that the number of movies is extremely large, so counting ticket sales of each one separately is not feasible.

## 4.8 Summary of Chapter 4

- ◆ *The Stream Data Model:* This model assumes data arrives at a processing engine at a rate that makes it infeasible to store everything in active storage. One strategy to dealing with streams is to maintain summaries of the streams, sufficient to answer the expected queries about the data. A second approach is to maintain a sliding window of the most recently arrived data.
- ◆ *Sampling of Streams:* To create a sample of a stream that is usable for a class of queries, we identify a set of key attributes for the stream. By hashing the key of any arriving stream element, we can use the hash value to decide consistently whether all or none of the elements with that key will become part of the sample.
- ◆ *Bloom Filters:* This technique allows us to filter streams so elements that belong to a particular set are allowed through, while most nonmembers are deleted. We use a large bit array, and several hash functions. Members of the selected set are hashed to buckets, which are bits in the array, and those bits are set to 1. To test a stream element for membership, we hash the element to a set of bits using each of the hash functions, and only accept the element if all these bits are 1.
- ◆ *Counting Distinct Elements:* To estimate the number of different elements appearing in a stream, we can hash elements to integers, interpreted as binary numbers.  $2^k$  raised to the power that is the longest sequence of 0's seen in the hash value of any stream element is an estimate of the number of different elements. By using many hash functions and combining these estimates, first by taking averages within groups, and then taking the median of the averages, we get a reliable estimate.
- ◆ *Moments of Streams:* The  $k$ th moment of a stream is the sum of the  $k$ th powers of the counts of each element that appears at least once in the stream. The 0th moment is the number of distinct elements, and the 1st moment is the length of the stream.
- ◆ *Estimating Second Moments:* A good estimate for the second moment, or surprise number, is obtained by choosing a random position in the stream, taking twice the number of times this element appears in the stream from that position onward, subtracting 1, and multiplying by the length of the stream. Many random variables of this type can be combined like the estimates for counting the number of distinct elements, to produce a reliable estimate of the second moment.
- ◆ *Estimating Higher Moments:* The technique for second moments works for  $k$ th moments as well, as long as we replace the formula  $2x - 1$  (where  $x$  is the number of times the element appears at or after the selected position) by  $x^k - (x - 1)^k$ .

- ◆ *Estimating the Number of 1's in a Window:* We can estimate the number of 1's in a window of 0's and 1's by grouping the 1's into buckets. Each bucket has a number of 1's that is a power of 2; there are one or two buckets of each size, and sizes never decrease as we go back in time. If we record only the position and size of the buckets, we can represent the contents of a window of size  $N$  with  $O(\log^2 N)$  space.
- ◆ *Answering Queries About Numbers of 1's:* If we want to know the approximate numbers of 1's in the most recent  $k$  elements of a binary stream, we find the earliest bucket  $B$  that is at least partially within the last  $k$  positions of the window and estimate the number of 1's to be the sum of the sizes of each of the more recent buckets plus half the size of  $B$ . This estimate can never be off by more than 50% of the true count of 1's.
- ◆ *Closer Approximations to the Number of 1's:* By changing the rule for how many buckets of a given size can exist in the representation of a binary window, so that either  $r$  or  $r - 1$  of a given size may exist, we can assure that the approximation to the true number of 1's is never off by more than  $1/r$ .
- ◆ *Exponentially Decaying Windows:* Rather than fixing a window size, we can imagine that the window consists of all the elements that ever arrived in the stream, but with the element that arrived  $t$  time units ago weighted by  $e^{-ct}$  for some time-constant  $c$ . Doing so allows us to maintain certain summaries of an exponentially decaying window easily. For instance, the weighted sum of elements can be recomputed, when a new element arrives, by multiplying the old sum by  $1 - c$  and then adding the new element.
- ◆ *Maintaining Frequent Elements in an Exponentially Decaying Window:* We can imagine that each item is represented by a binary stream, where 0 means the item was not the element arriving at a given time, and 1 means that it was. We can find the elements whose sum of their binary stream is at least  $1/2$ . When a new element arrives, multiply all recorded sums by  $1 - c$ , add 1 to the count of the item that just arrived, and delete from the record any item whose sum has fallen below  $1/2$ .

## 4.9 References for Chapter 4

Many ideas associated with stream management appear in the “chronicle data model” of [8]. An early survey of research in stream-management systems is [2]. Also, [6] is a recent book on the subject of stream management.

The sampling technique of Section 4.2 is from [7]. The Bloom Filter is generally attributed to [3], although essentially the same technique appeared as “superimposed codes” in [9].

The algorithm for counting distinct elements is essentially that of [5], although the particular method we described appears in [1]. The latter is also the source for the algorithm for calculating the surprise number and higher moments. However, the technique for maintaining a uniformly chosen sample of positions in the stream is called “reservoir sampling” and comes from [10].

The technique for approximately counting 1’s in a window is from [4].

1. N. Alon, Y. Matias, and M. Szegedy, “The space complexity of approximating frequency moments,” *28th ACM Symposium on Theory of Computing*, pp. 20–29, 1996.
2. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and issues in data stream systems,” *Symposium on Principles of Database Systems*, pp. 1–16, 2002.
3. B.H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Comm. ACM* **13**:7, pp. 422–426, 1970.
4. M. Datar, A. Gionis, P. Indyk, and R. Motwani, “Maintaining stream statistics over sliding windows,” *SIAM J. Computing* **31**, pp. 1794–1813, 2002.
5. P. Flajolet and G.N. Martin, “Probabilistic counting for database applications,” *24th Symposium on Foundations of Computer Science*, pp. 76–82, 1983.
6. M. Garofalakis, J. Gehrke, and R. Rastogi (editors), *Data Stream Management*, Springer, 2009.
7. P.B. Gibbons, “Distinct sampling for highly-accurate answers to distinct values queries and event reports,” *Intl. Conf. on Very Large Databases*, pp. 541–550, 2001.
8. H.V. Jagadish, I.S. Mumick, and A. Silberschatz, “View maintenance issues for the chronicle data model,” *Proc. ACM Symp. on Principles of Database Systems*, pp. 113–124, 1995.
9. W.H. Kautz and R.C. Singleton, “Nonadaptive binary superimposed codes,” *IEEE Transactions on Information Theory* **10**, pp. 363–377, 1964.
10. J. Vitter, “Random sampling with a reservoir,” *ACM Transactions on Mathematical Software* **11**:1, pp. 37–57, 1985.