Big Data

(Hadoop)

Instructor: Trong-Hop Do

March 8th, 2021

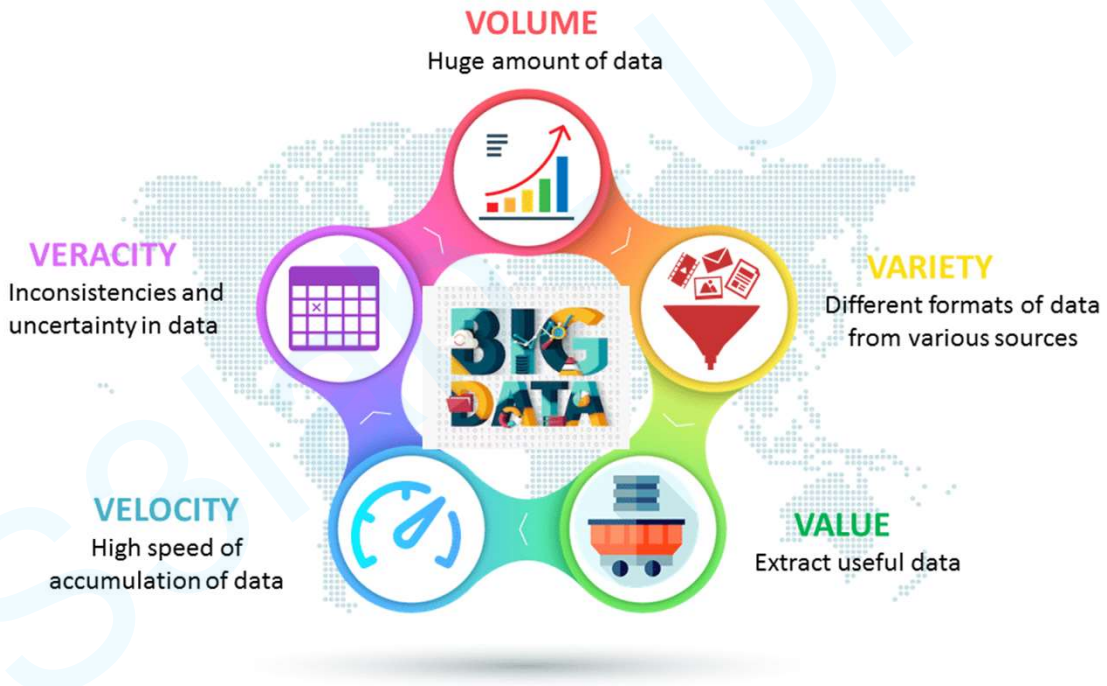
S3Lab
Smart Software System Laboratory


“Big data is at the foundation of all the megatrends that are happening today, from social to mobile to cloud to gaming.”
– Chris Lynch, Vertica Systems

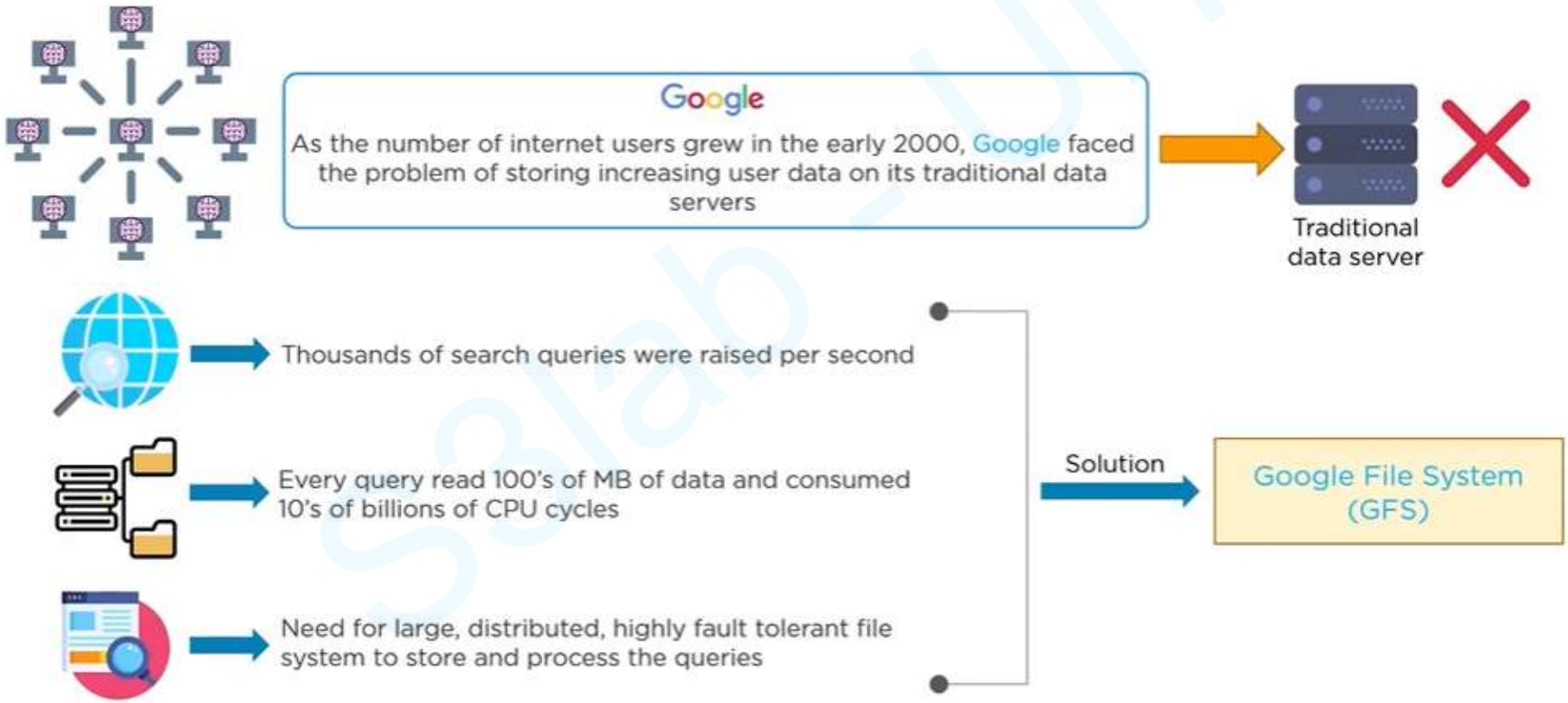
Distributed File System

Google File System

Recap – Big data 5V's



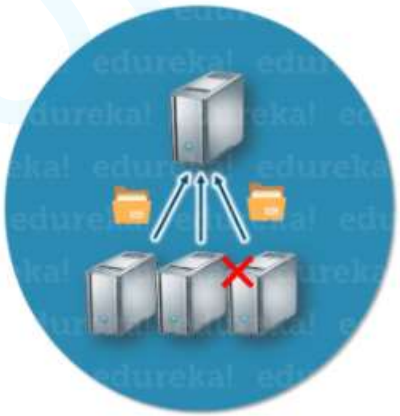
Big data case study – Google File System



Big data case study – Google File System

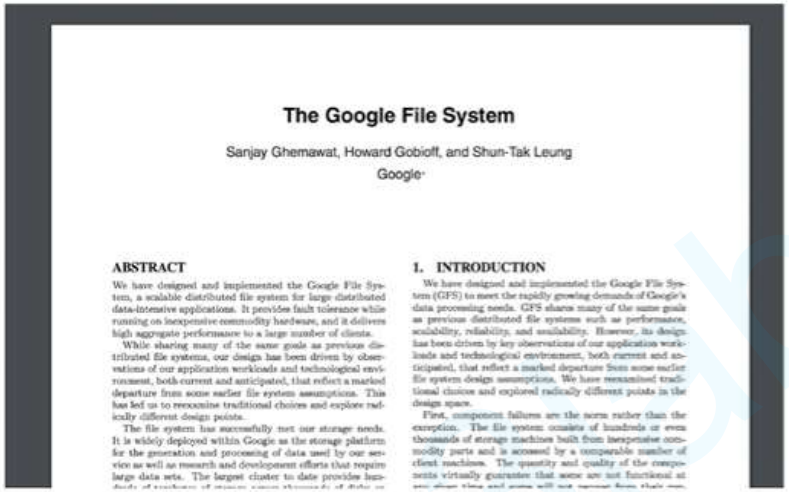
Storing huge and exponentially growing datasets

Processing data having complex structure (structured, un-structured, semi-structured)



Bringing huge amount of data to computation unit becomes a bottleneck

The 2003 Google File System paper



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hun-

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at

Paper describing
Google File System

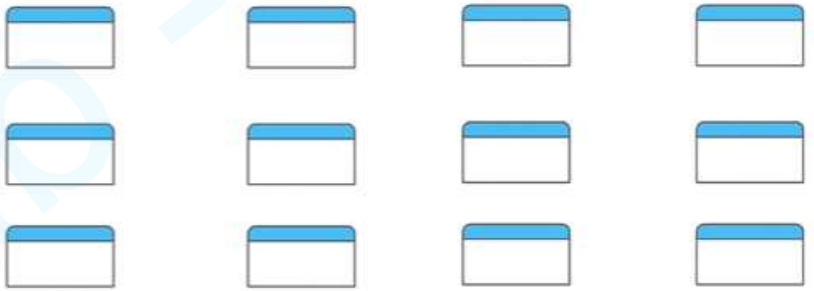
Used as a basis to
design Hadoop

What is Google File System?

GFS design consideration: Commodity hardware

Failures are common

- disk / network / server
- OS bugs
- human errors

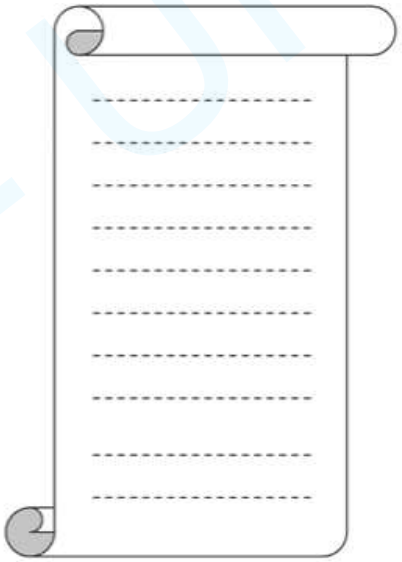


Commodity servers are cheap & can be made to scale horizontally with right software

GFS design consideration: Large files

100MB to multi-GB files

- Crawled web documents
- Batch processing

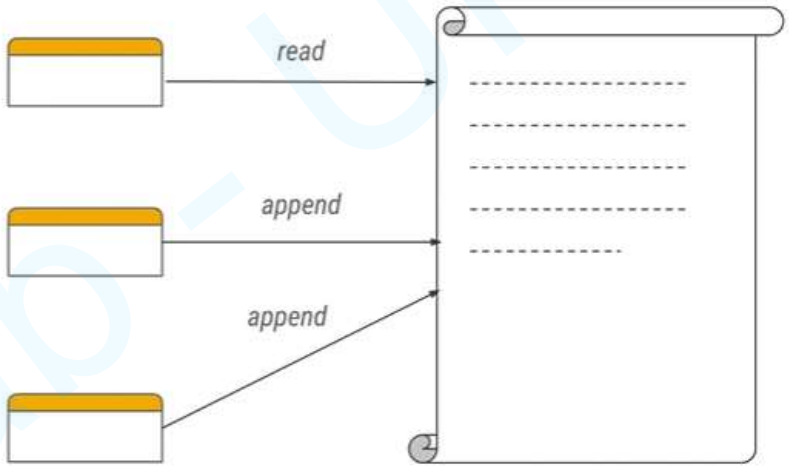


Commodity servers are cheap & can be made to scale horizontally with right software

GFS design consideration: File operations

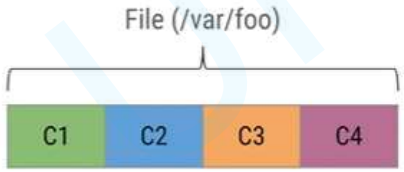
Read + Append only

- No random writes
- Mostly sequential reads



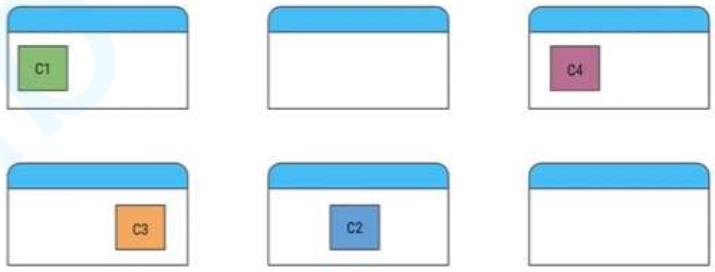
Think of web crawling - Keep appending crawled content & Use batch processing (reads) to create index

GFS design consideration: Chunks



Files split into chunks

- Each chunk of 64MB
- Identified by 64 bit ID
- Stored in Chunkservers

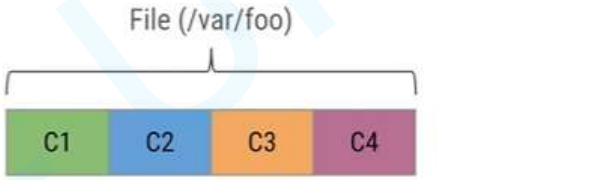


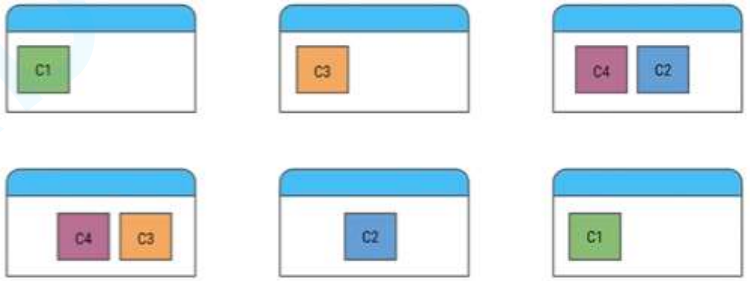
Chunkservers - Chunks of single file are distributed on multiple machines

GFS design consideration: Replicas

Files split into chunks

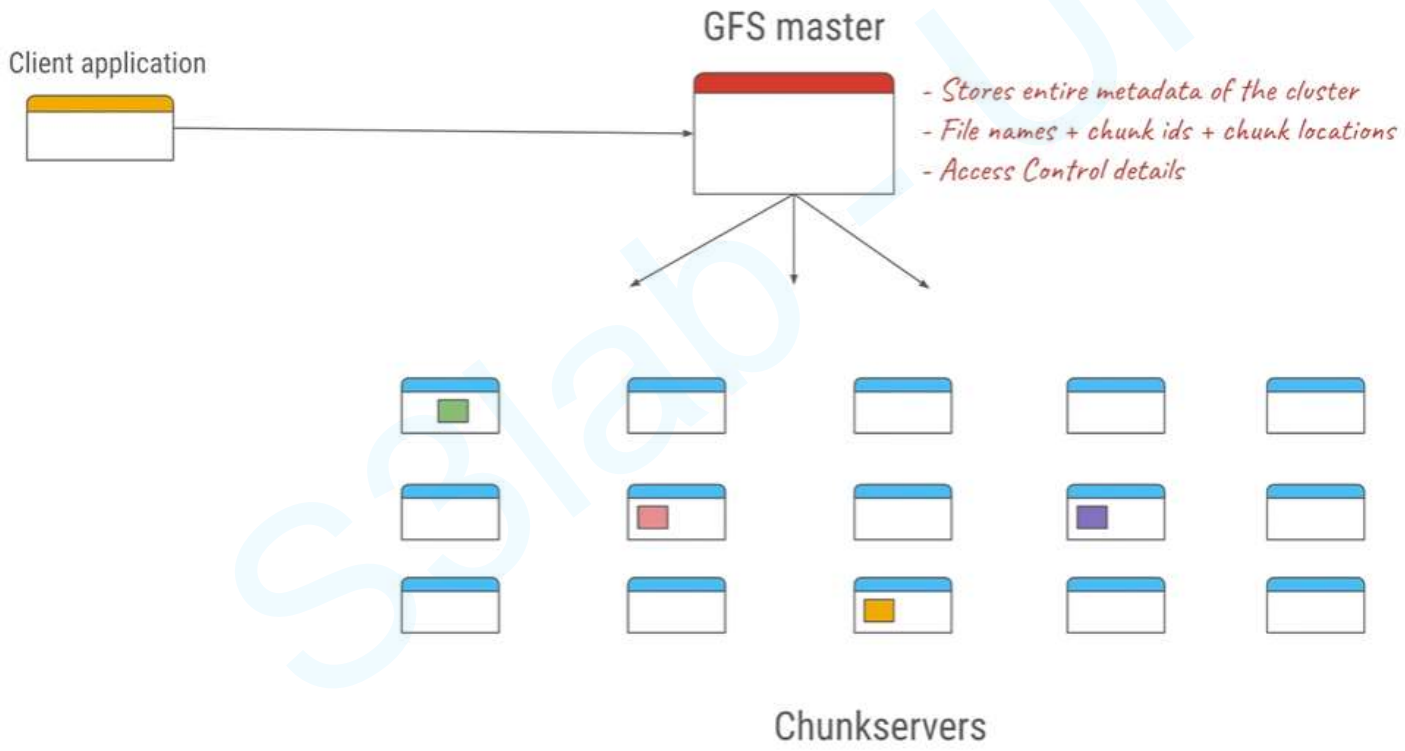
- Replica count by client
- commodity server failures



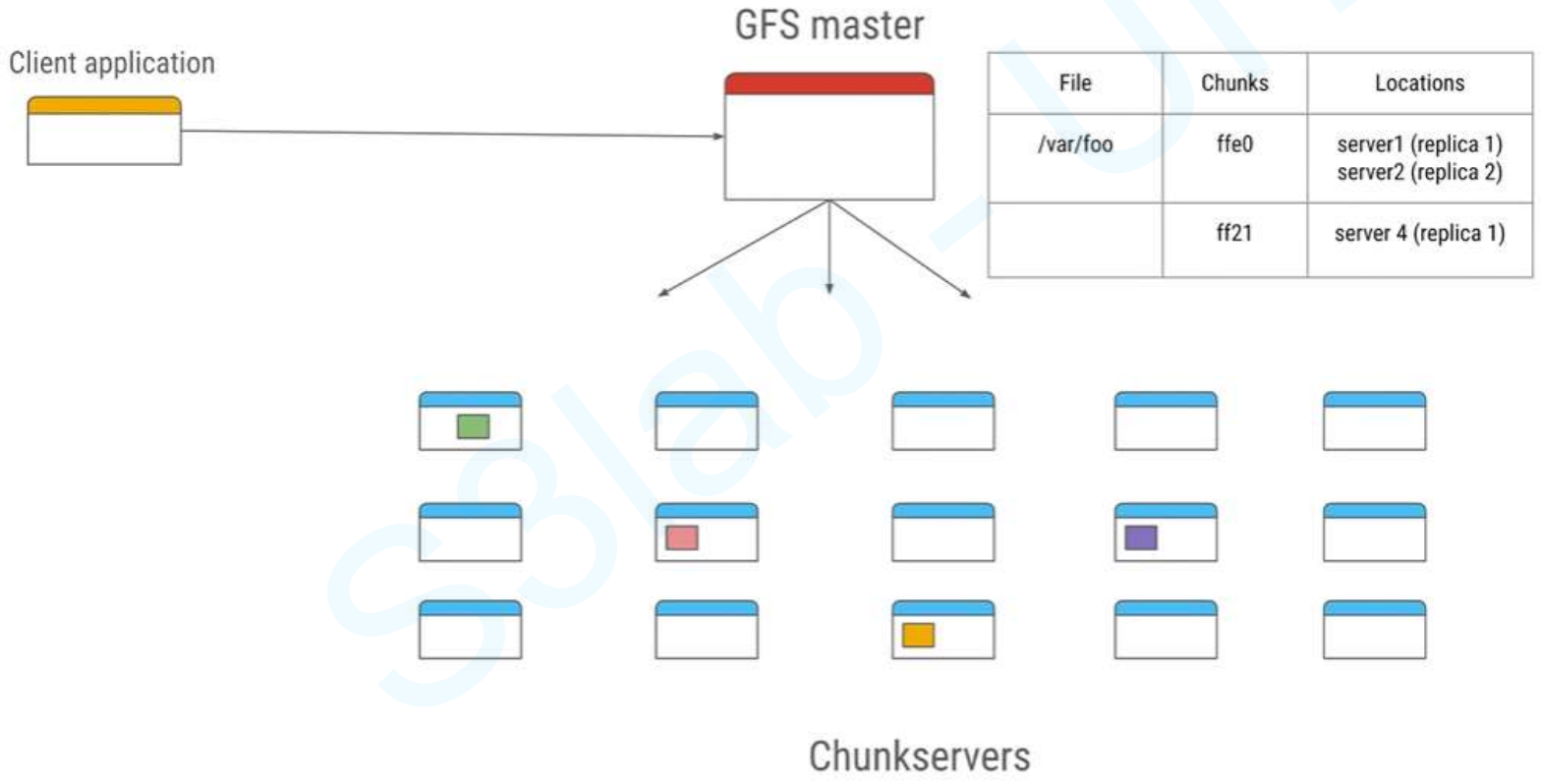


Replicas ensure durability of data if chunkserver goes down

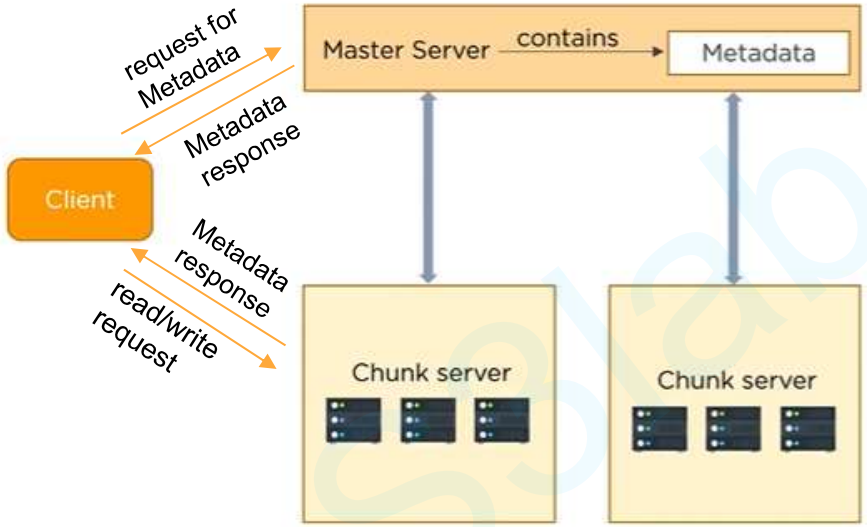
GFS design consideration: Large files

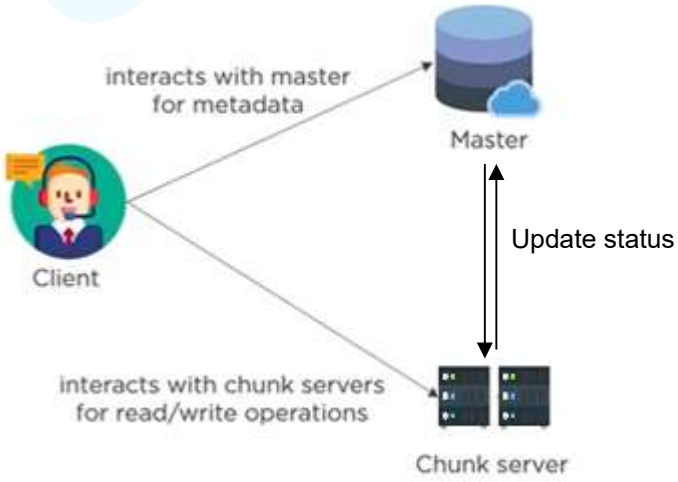


GFS architecture

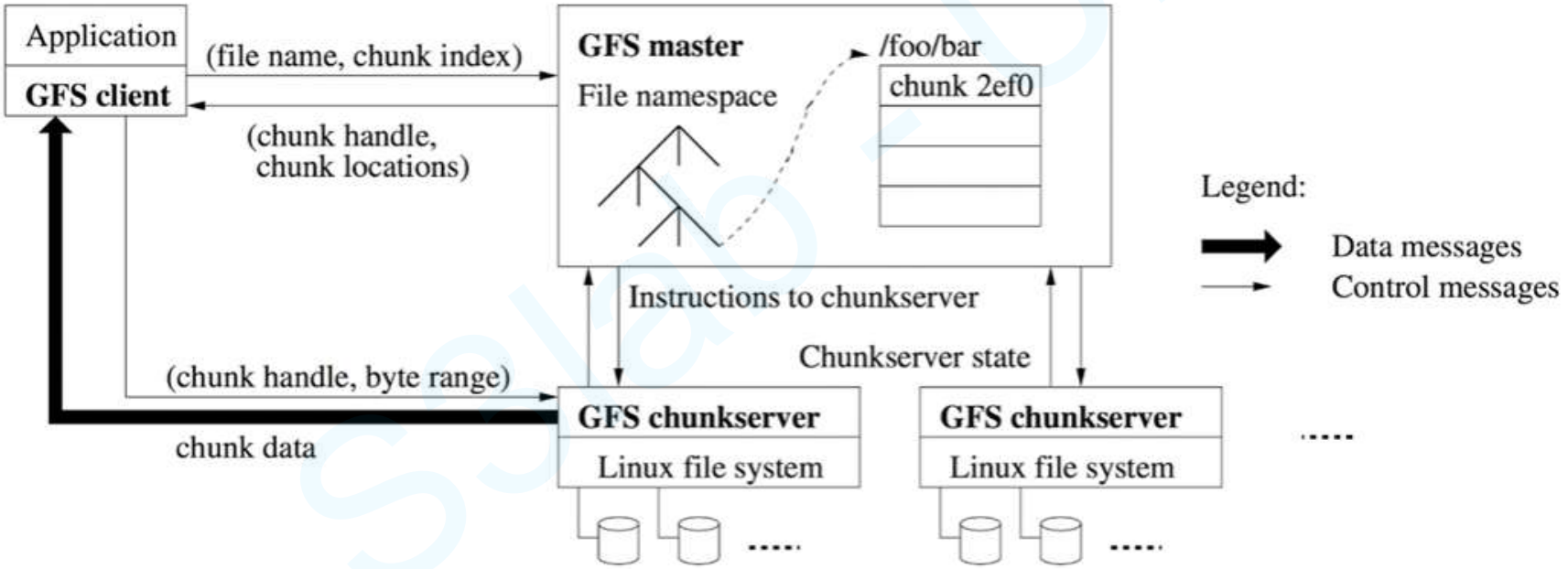


GFS architecture



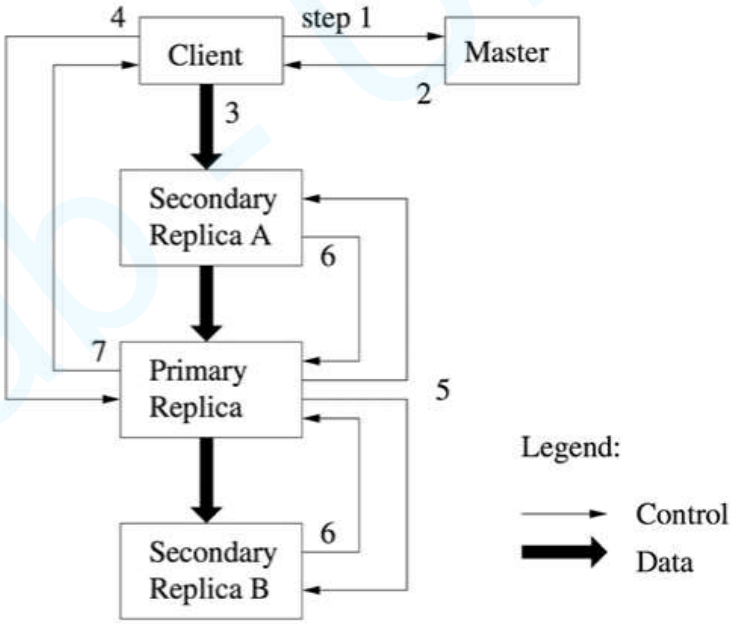


GFS: Read



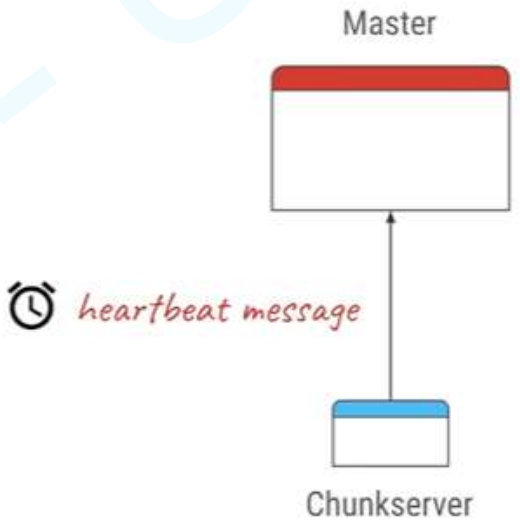
GFS: Write

1. Ask for locations to write
2. Get replicate locations
3. Write data to closest replica.
4. Request commit to primary
5. Primary instructs order of writes to secondaries
6. Secondaries acknowledge
7. Primary ack to client



GFS: Heartbeat

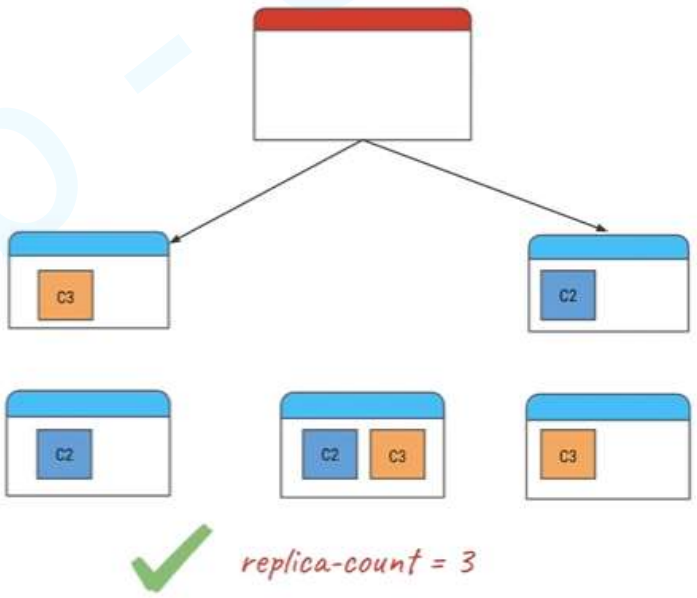
Regular heartbeats to ensure chunkservers are alive



GFS: Ensure chunk replica count

If chunkserver is down, master ensures all chunks that were on it are copied on other servers.

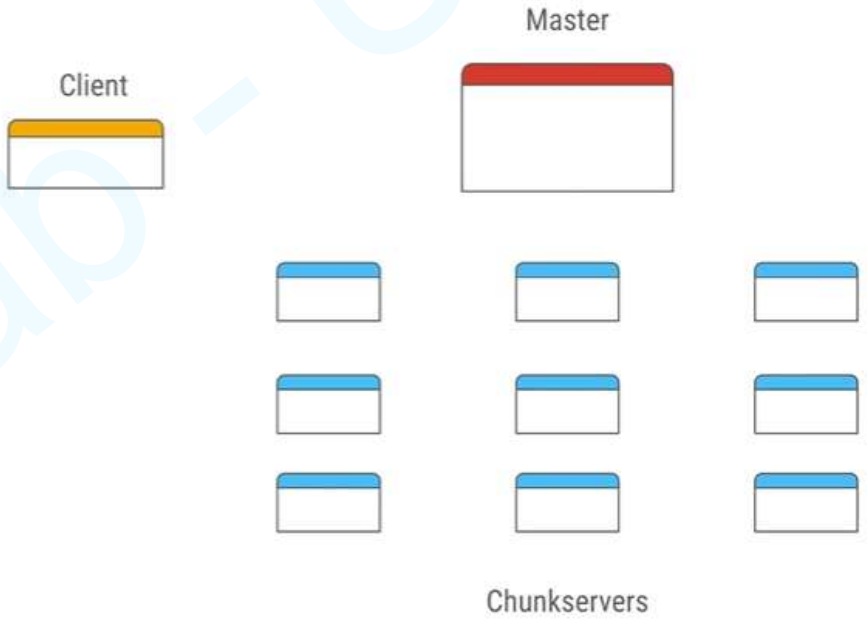
Ensures replica counts remains same.



GFS: Single Master for multi-TB cluster

Large chunk size

- 64MB chunk
- Reduced meta-data
- Reduces client interactions
- Client caches location data



GFS: Operation logs

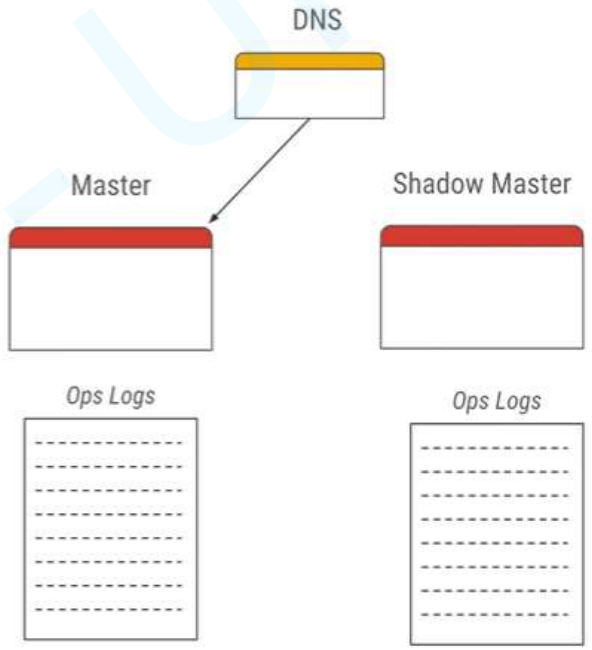
Record of all ops

- Checkpointed regularly
- Happens in background thread
- Used if master crashes
- Rebooted master replays log

GFS: Operation logs

Single Point of Failure

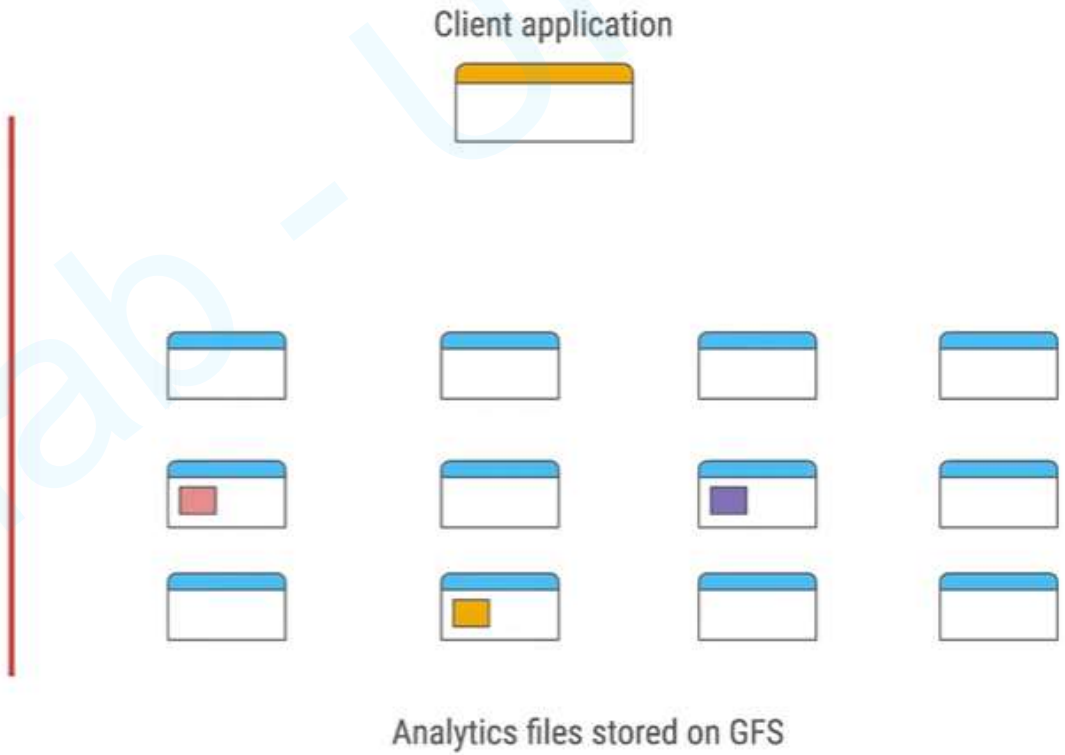
- Ops log is replicated remotely
- Shadow master uses the logs
- DNS change can change master
- Shadow master may lag slightly



Use case of data processing in GFS

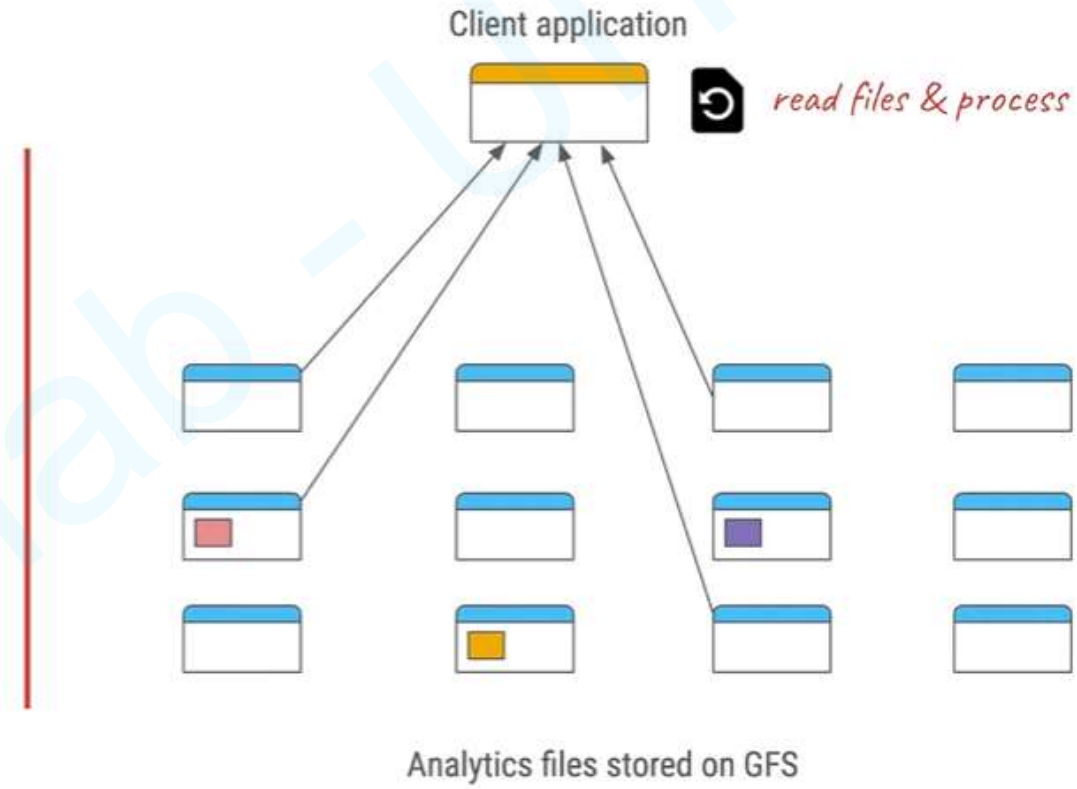
Google Music analytics stored in GFS.

Write program to find while song was played how many times.



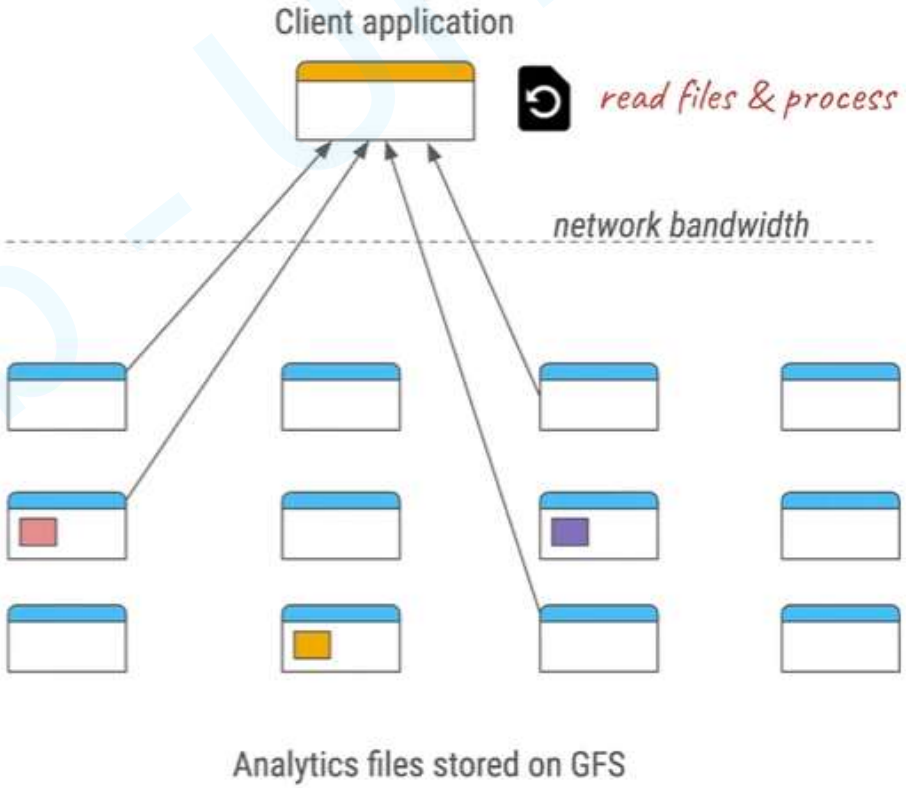
Use case of data processing in GFS

Read files from GFS. Process the files.



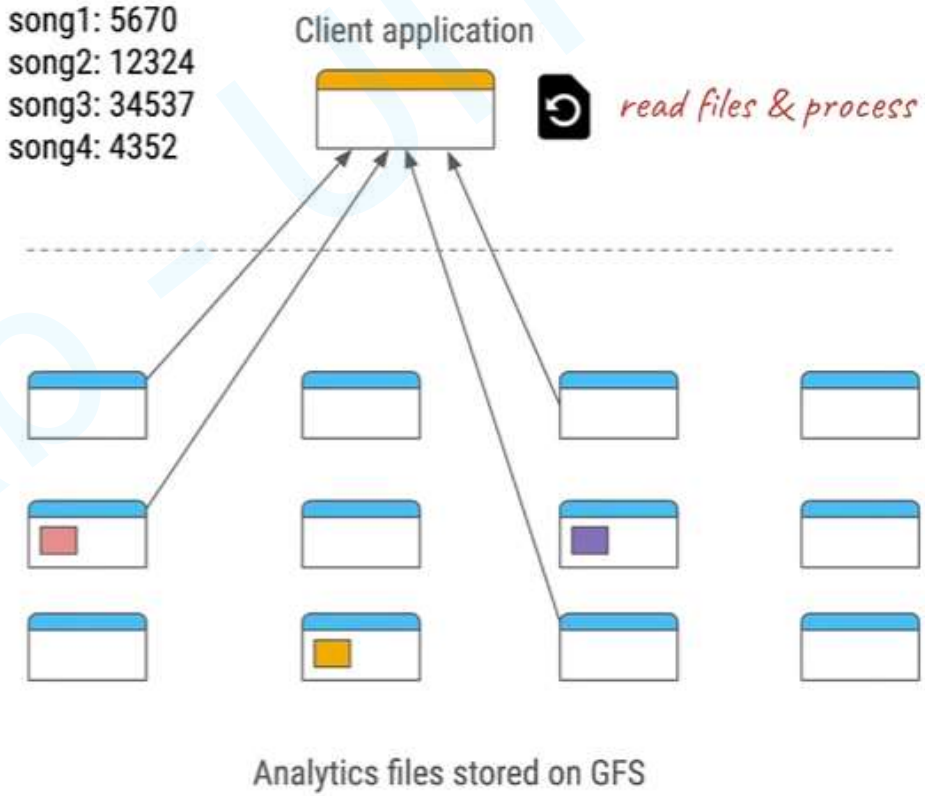
Use case of data processing in GFS

- High network bandwidth
- Multi-TB file(s)
- Slow to process



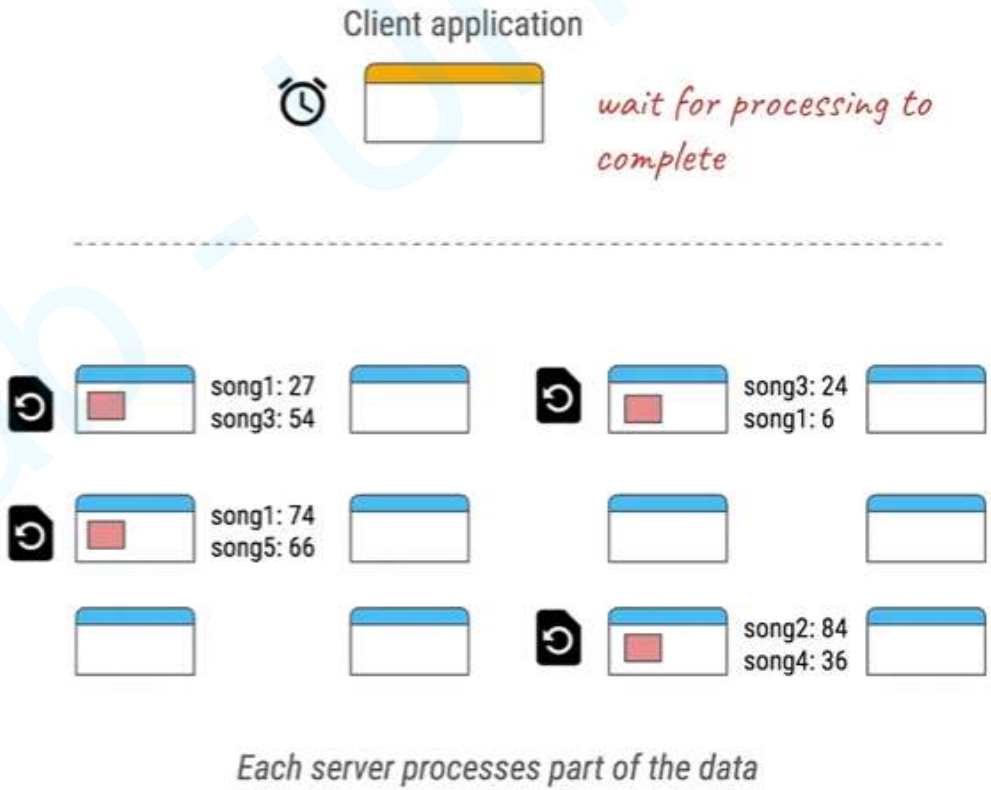
Use case of data processing in GFS

Instead of bringing data to client, bring the processing function to the data.



Use case of data processing in GFS

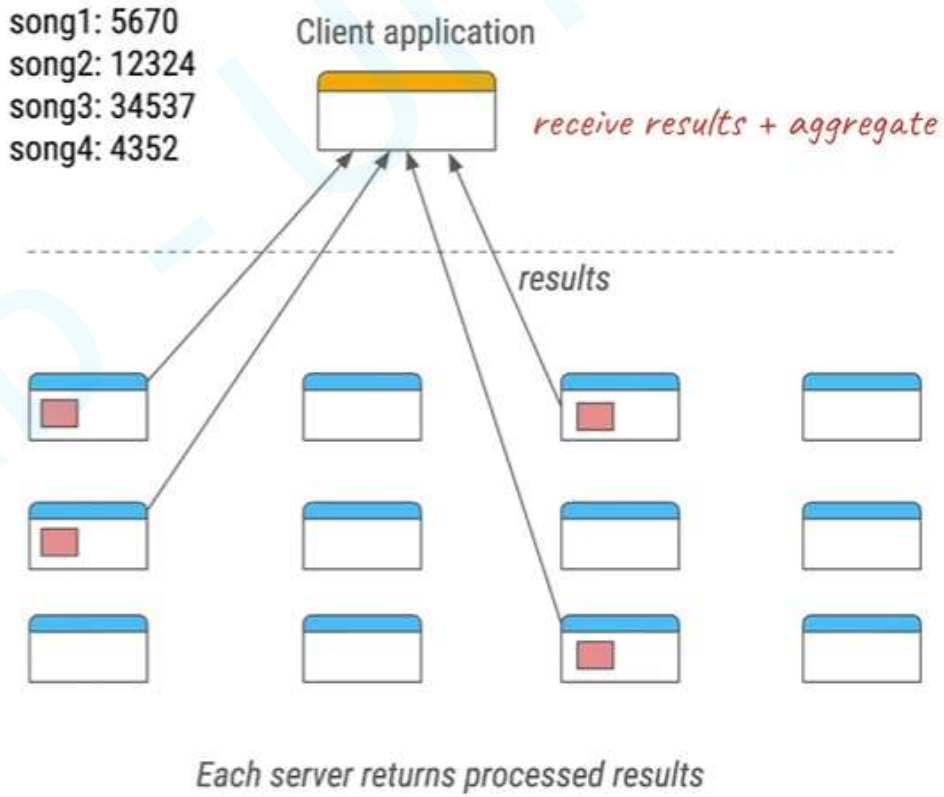
Each server applies function to the chunk it has.



Each server processes part of the data

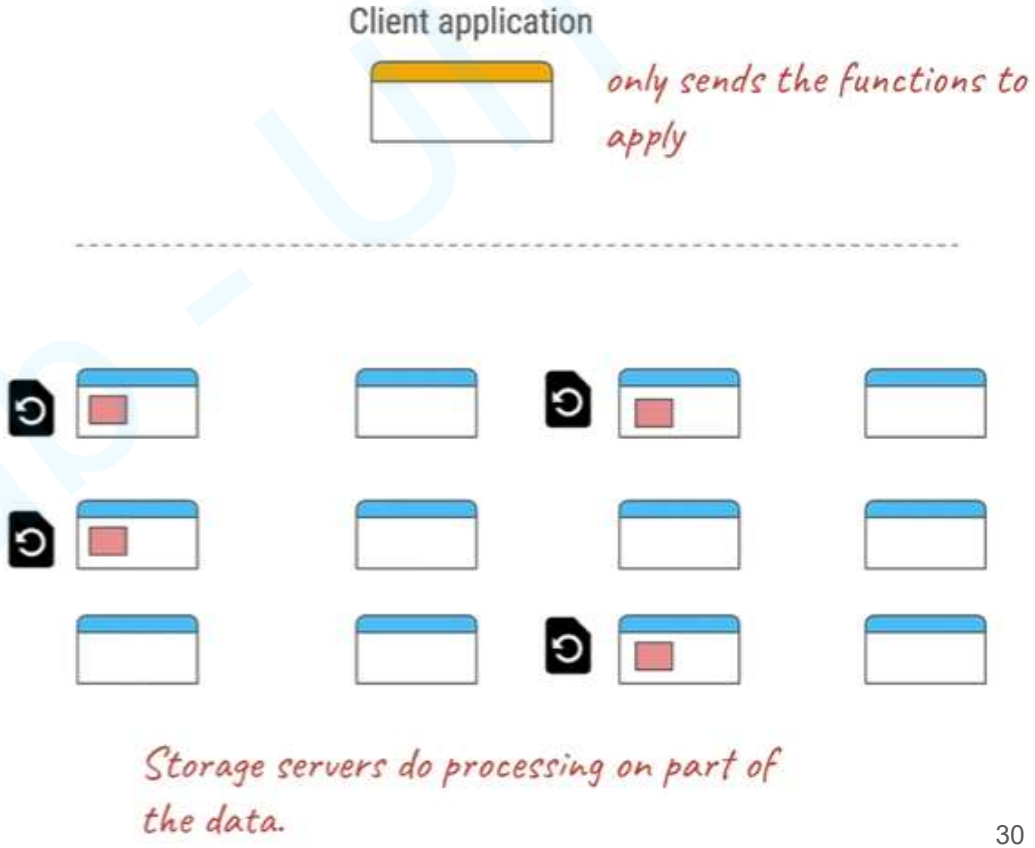
Use case of data processing in GFS

Once each server completes processing it can send result back to the application.



Map Reduce in GFS

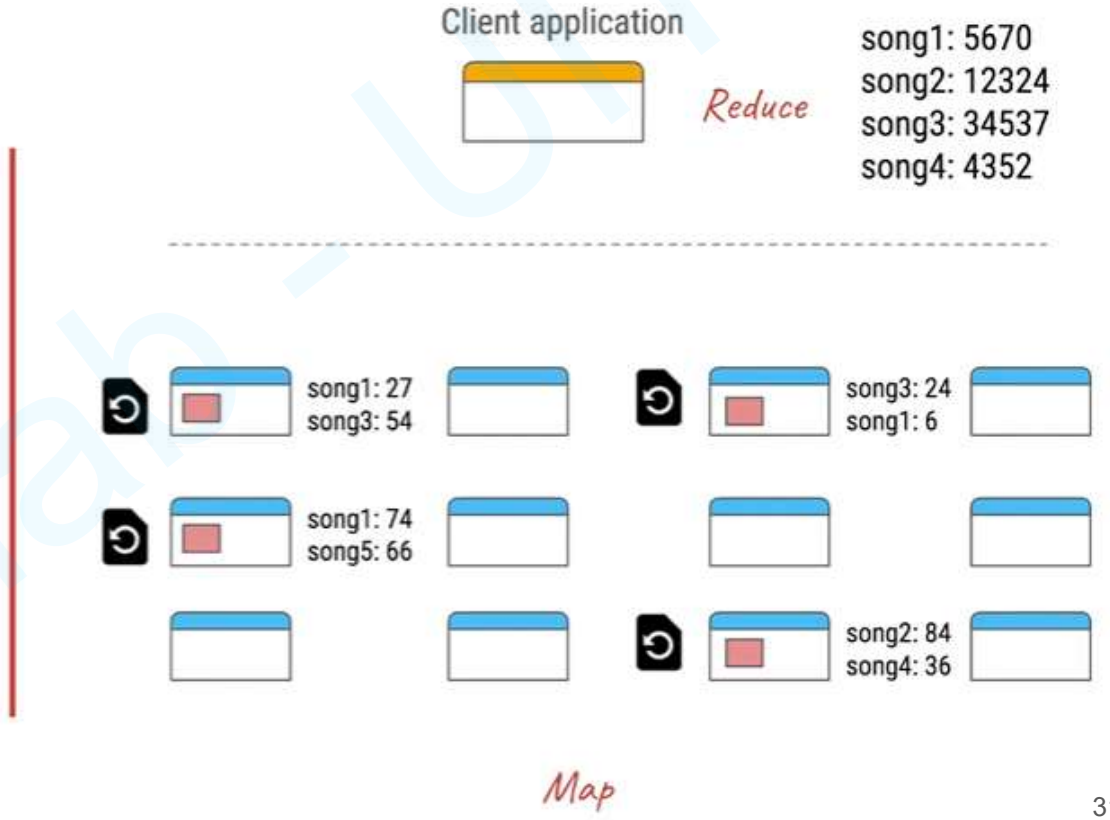
Programming model for distributed big data processing.



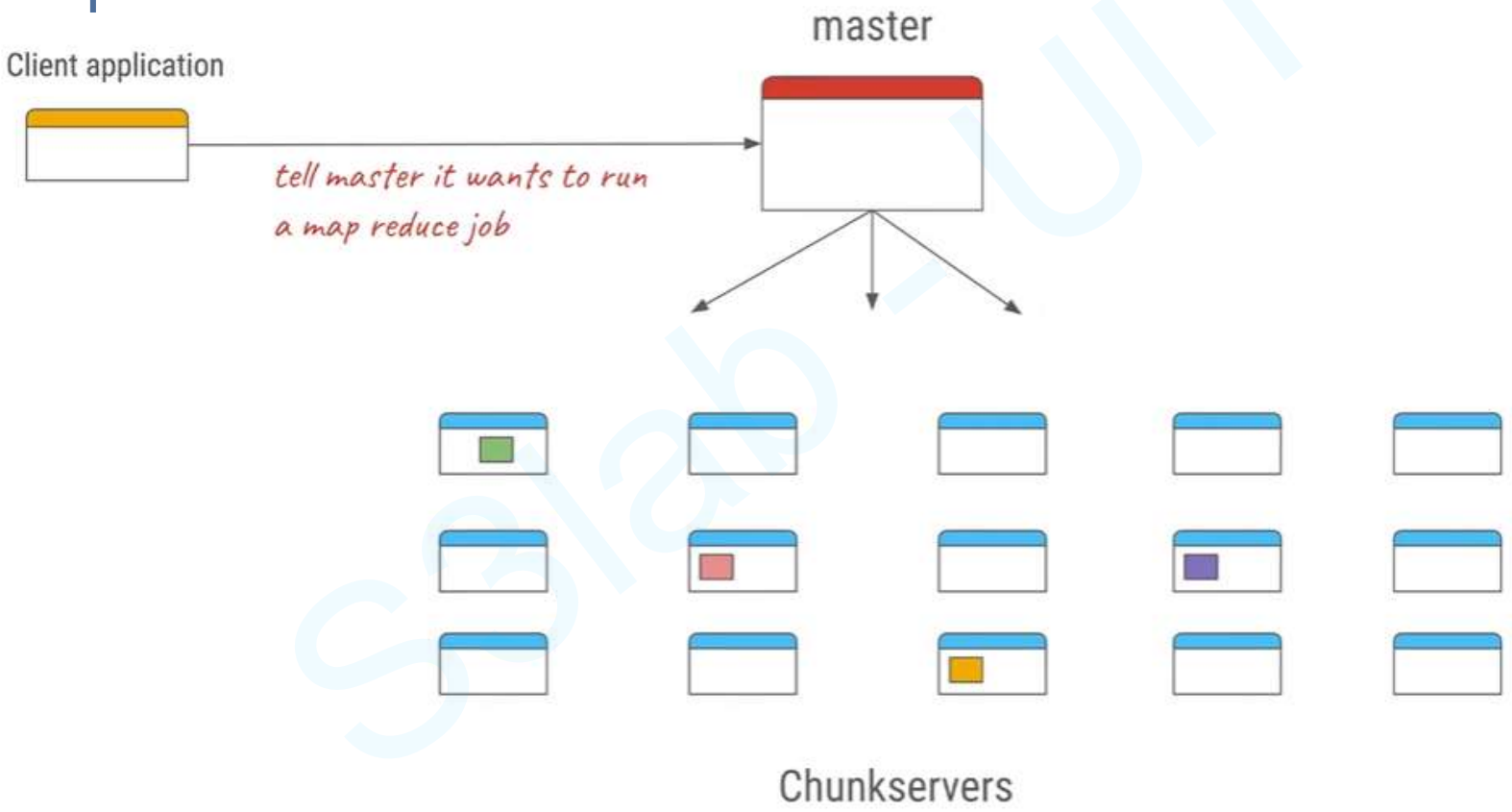
Map + Reduce

Map = Processing part of data

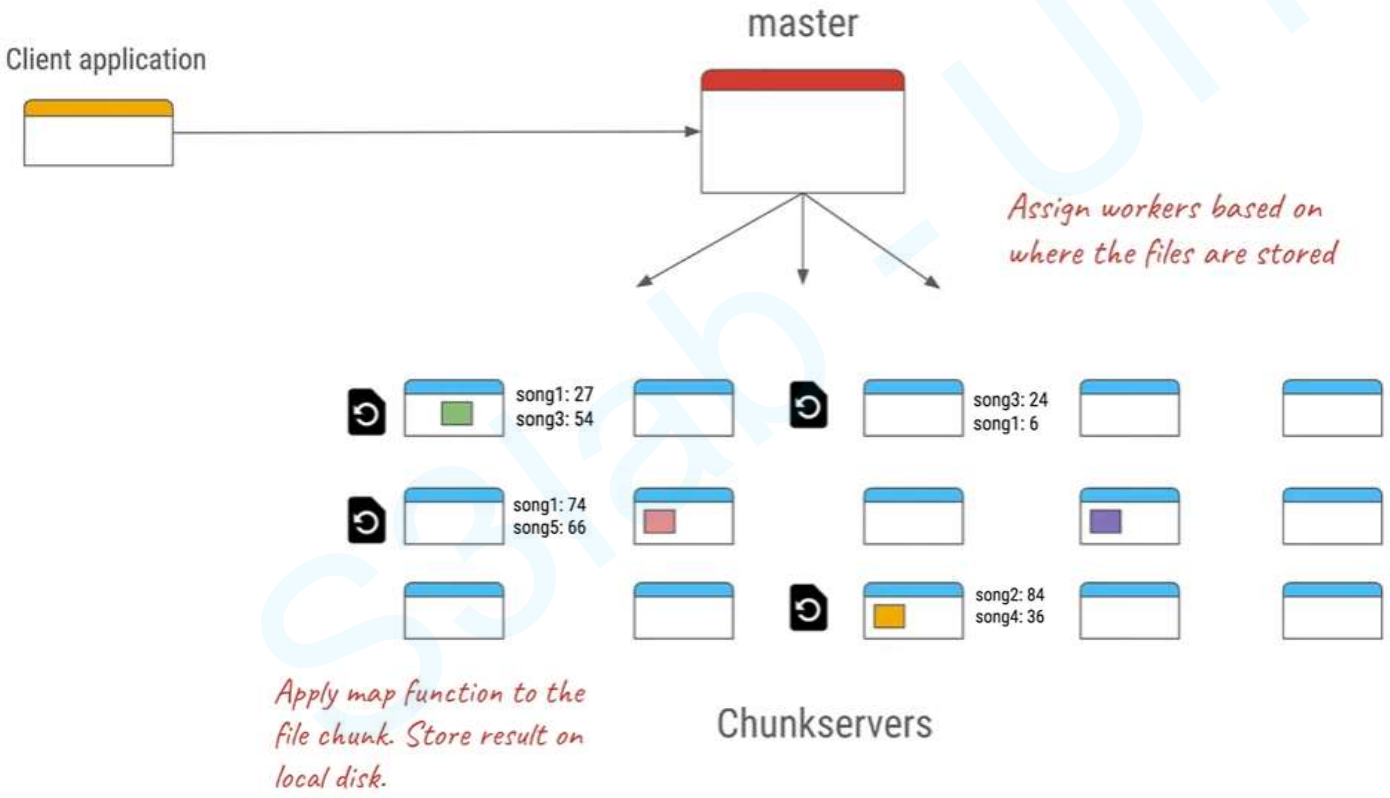
Reduce = Aggregation



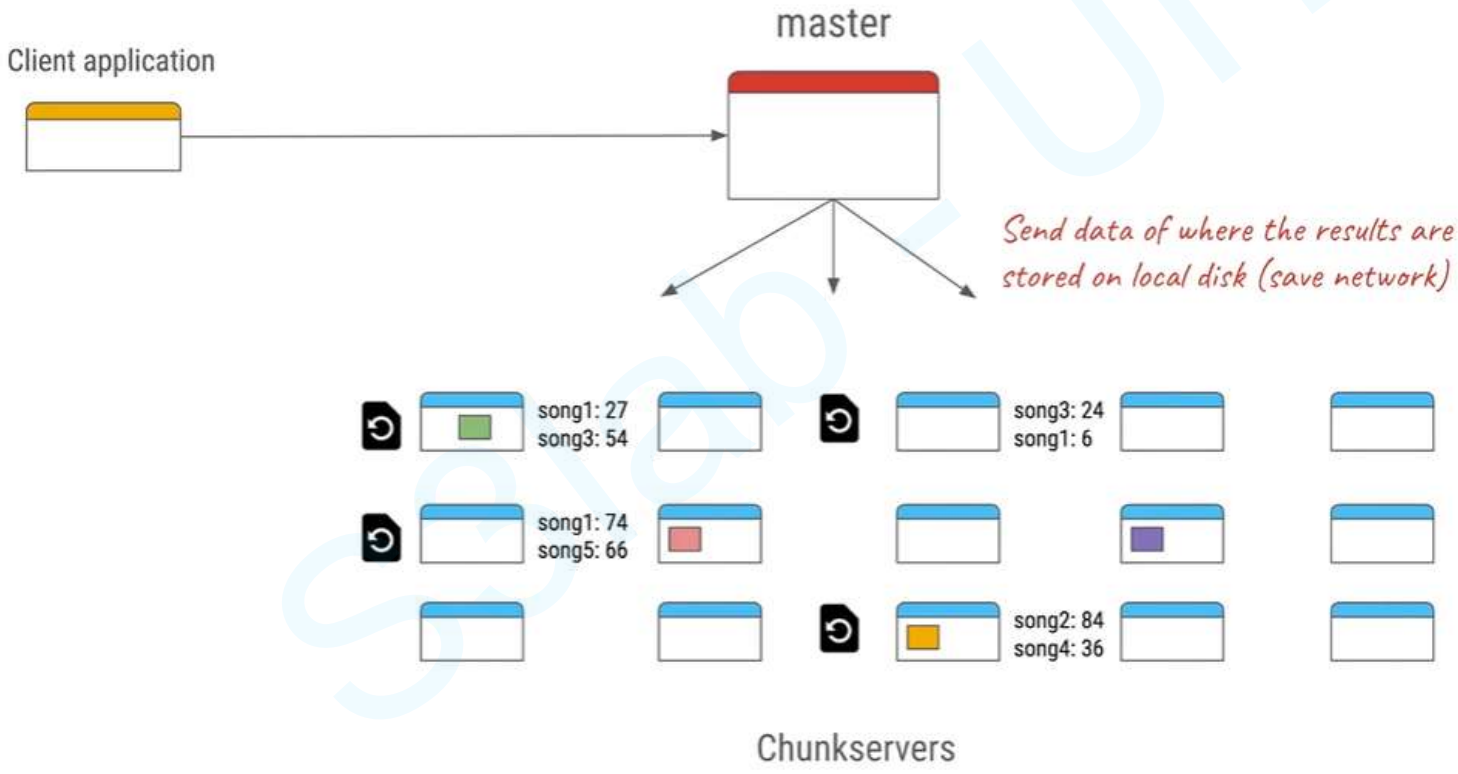
Map + Reduce



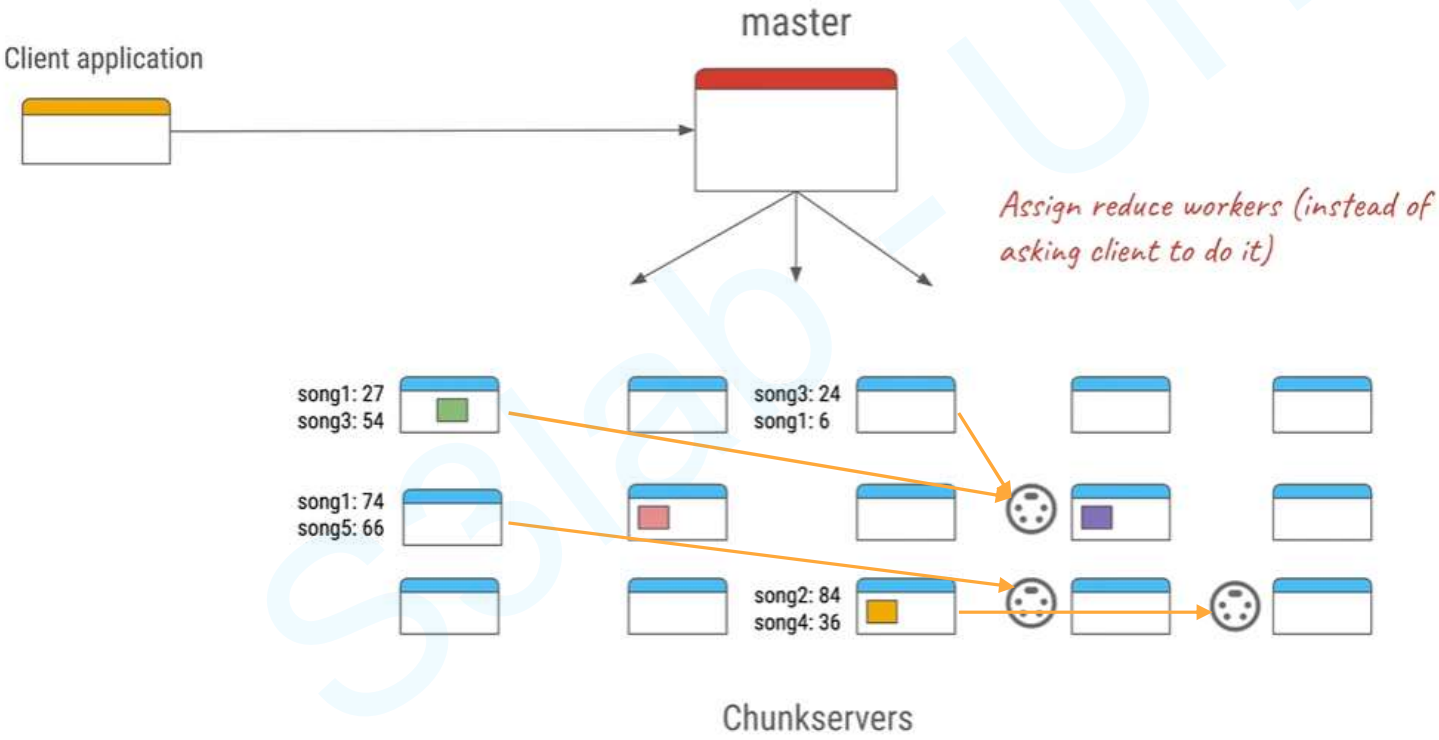
Map + Reduce


master
Client application
Assign workers based on where the files are stored
song1: 27
song3: 54
song3: 24
song1: 6
song1: 74
song5: 66
song2: 84
song4: 36
Apply map function to the file chunk. Store result on local disk.
Chunkservers

Map + Reduce


master
Client application
Send data of where the results are stored on local disk (save network)
song1: 27
song3: 54
song3: 24
song1: 6
song1: 74
song5: 66
song2: 84
song4: 36
Chunkservers

Map + Reduce


master
Client application
Assign reduce workers (instead of asking client to do it)
song1: 27
song3: 54
song3: 24
song1: 6
song1: 74
song5: 66
song2: 84
song4: 36
Chunkservers

Hadoop

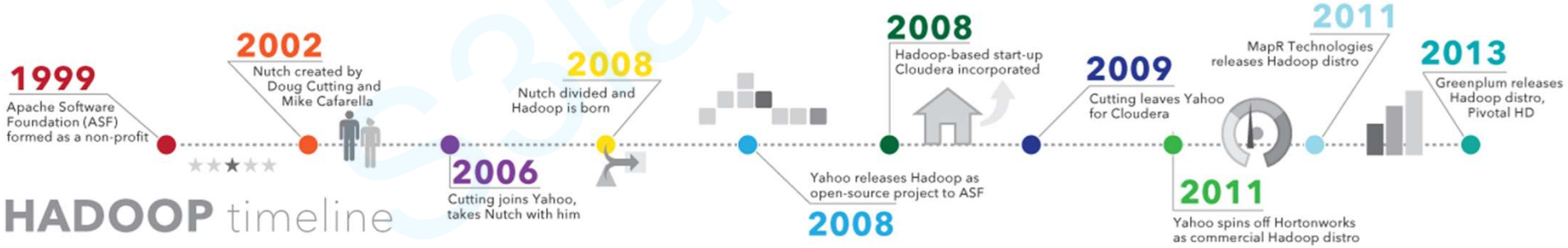
Hadoop architecture

Introduction

- Hadoop is a software framework written in **Java** for distributed processing of large datasets (terabytes or petabytes of data) across large clusters (thousands of nodes) of computers. Included some key components as below:
  - **Hadoop Common**: common utilities
  - **Hadoop Distributed File System** (**HDFS**) (Storage Component): A distributed file system that provides high-throughput access
  - **Hadoop YARN** (Scheduling): a framework for job scheduling & cluster resource management (available from Hadoop 2.x)
  - **Hadoop MapReduce** (Processing): A yarn-based system for parallel processing of large data sets

Introduction

- Hadoop is a large and active ecosystem.
- Hadoop emerged as a solution for big data problems.
- Open source under the friendly Apache License
- Originally built as a Infrastructure for the “Nutch” project.
- Based on Google’s mapreduce and google File System.

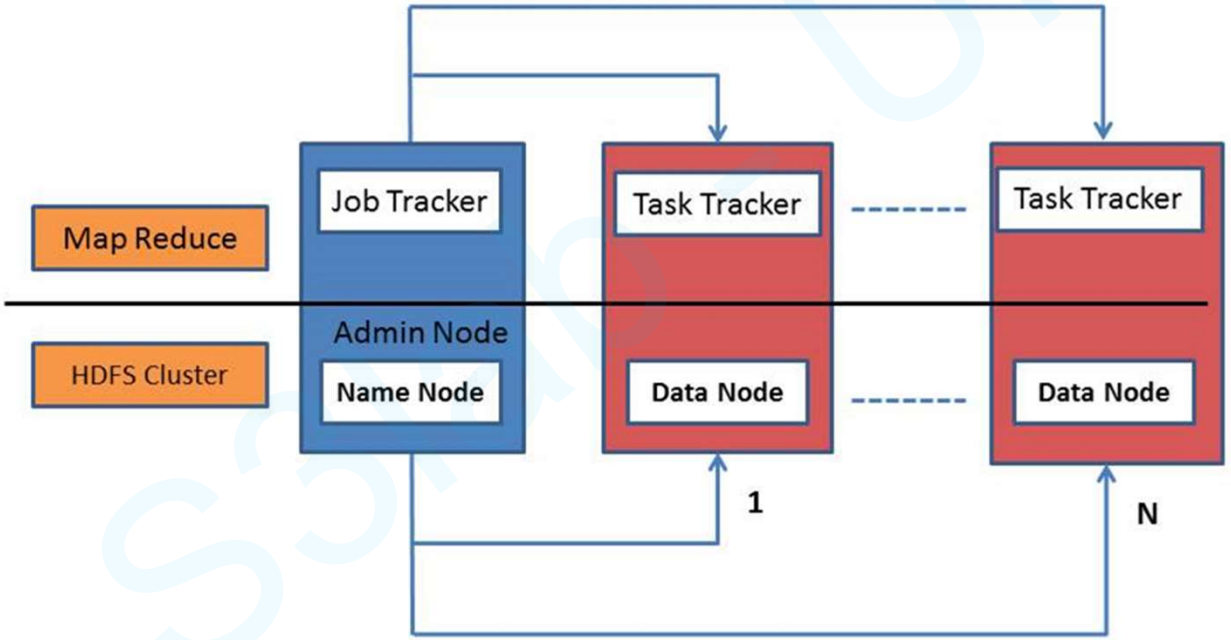


Features


Hadoop Features
High Availability
Economic
Fault Tolerance
Reliability
Replication
Flexibility
Data locality
Easy to use
Scalability
Distributed Storage
Data Flair

Hadoop 1.x architecture

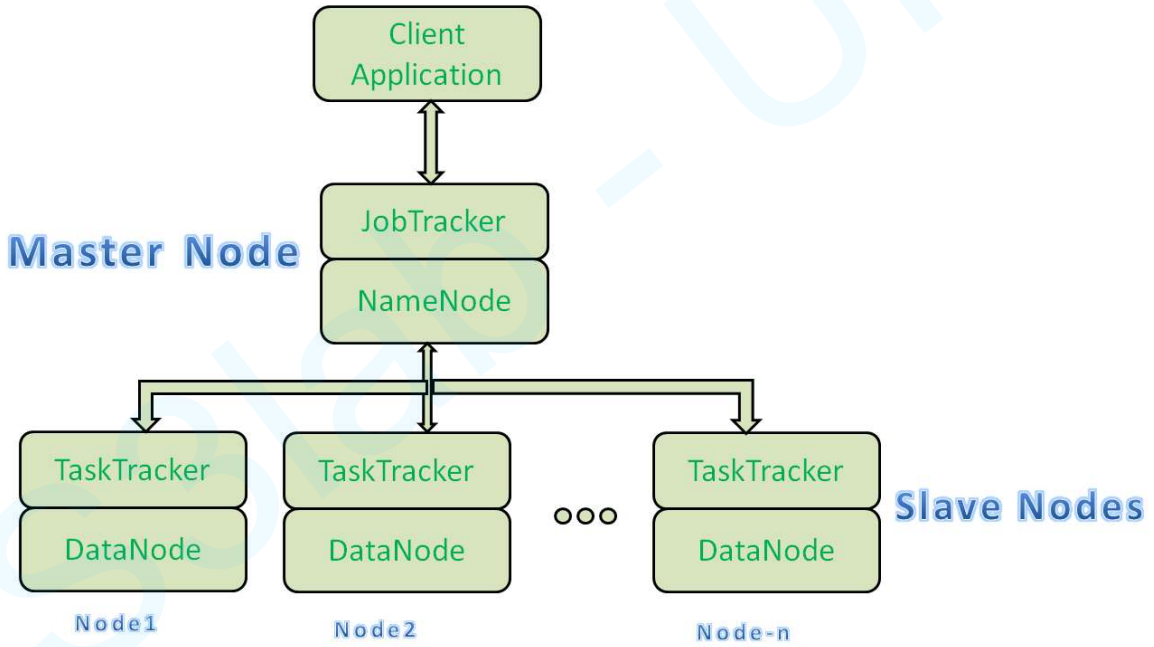
Core components



HDFS and MapReduce are known as “Two Pillars” of Hadoop 1.x

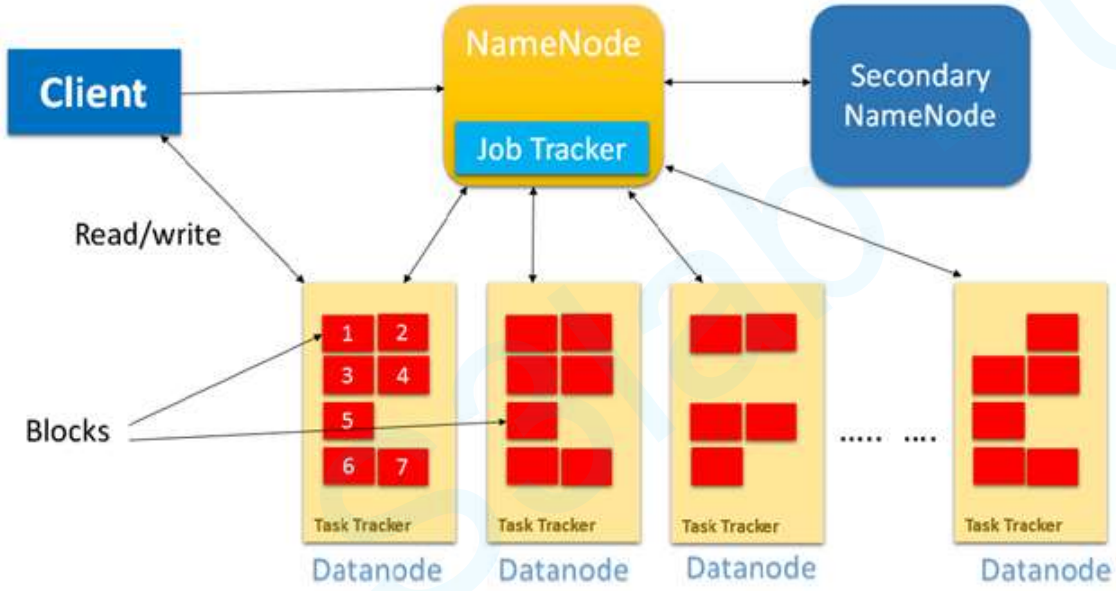
Hadoop 1.x architecture

Core components



Hadoop 1.x architecture

Core components

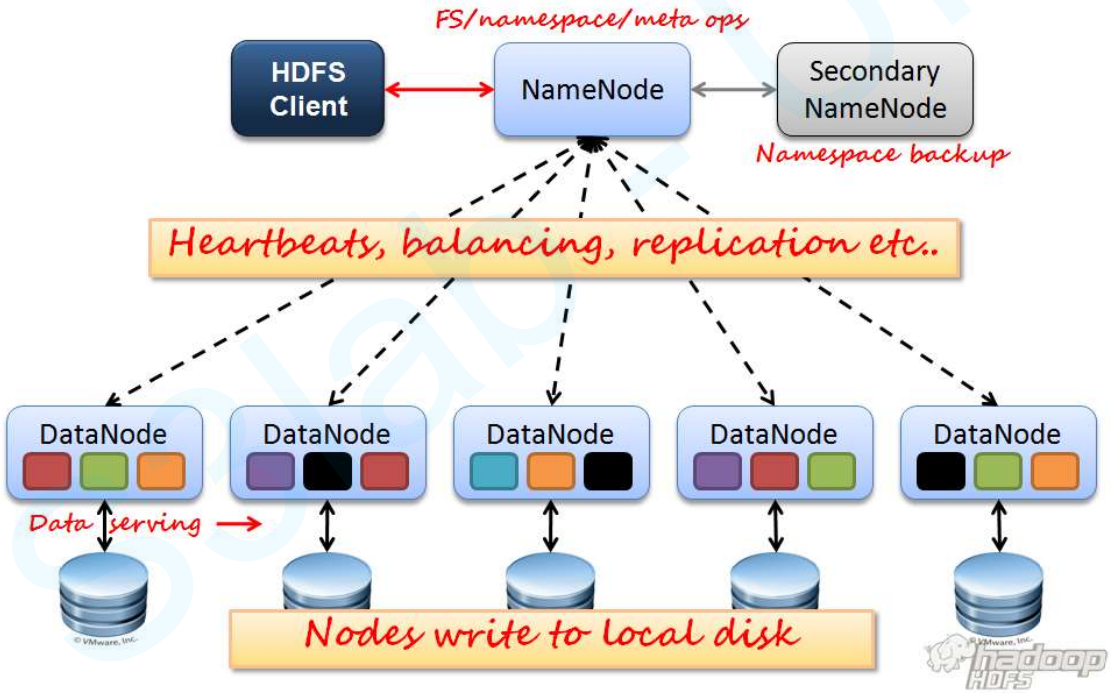


5 Hadoop daemons:

- Namenode
- Secondary namenode
- Jobtracker
- Datanode
- Tasktracker

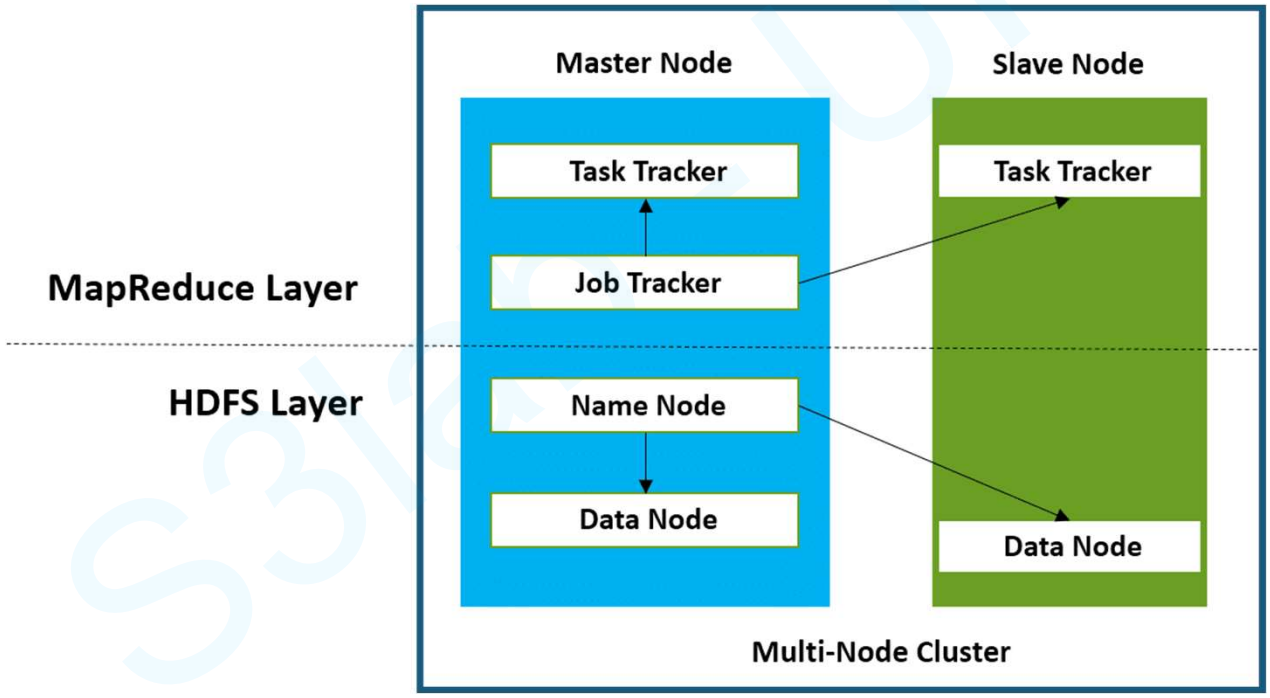
Hadoop 1.x architecture

Core components



Architecture

Multi-Node Cluster

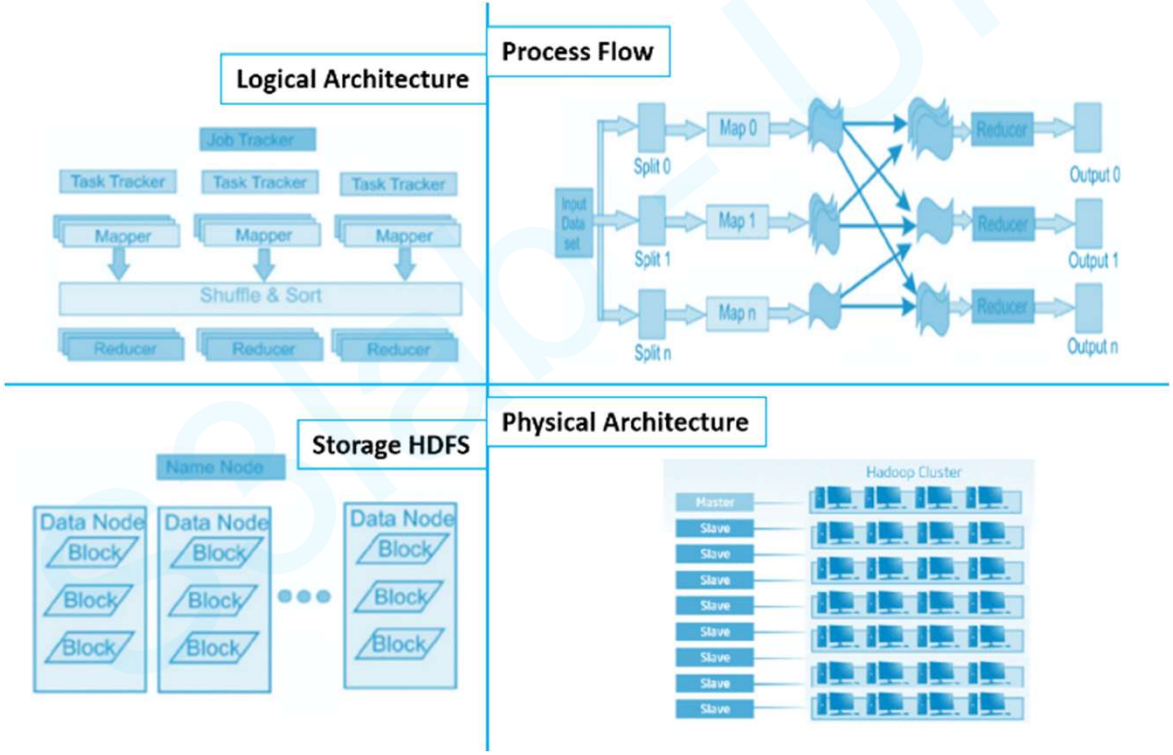


What makes Hadoop unique

- **Data locality and Shared Nothing**: Moving computation to data, instead of moving data to computation. Each node can independently process a much smaller subset of the entire dataset without needing to communicate with one another.
- **Simplified programming model**: allows user to quickly write and test
- **Schema-on-read system** (same as NoSQL platforms) # **Schema-on-write system**
- Automatic distribution of data and work across machines

Architecture

Architecture in different perspective



HDFS

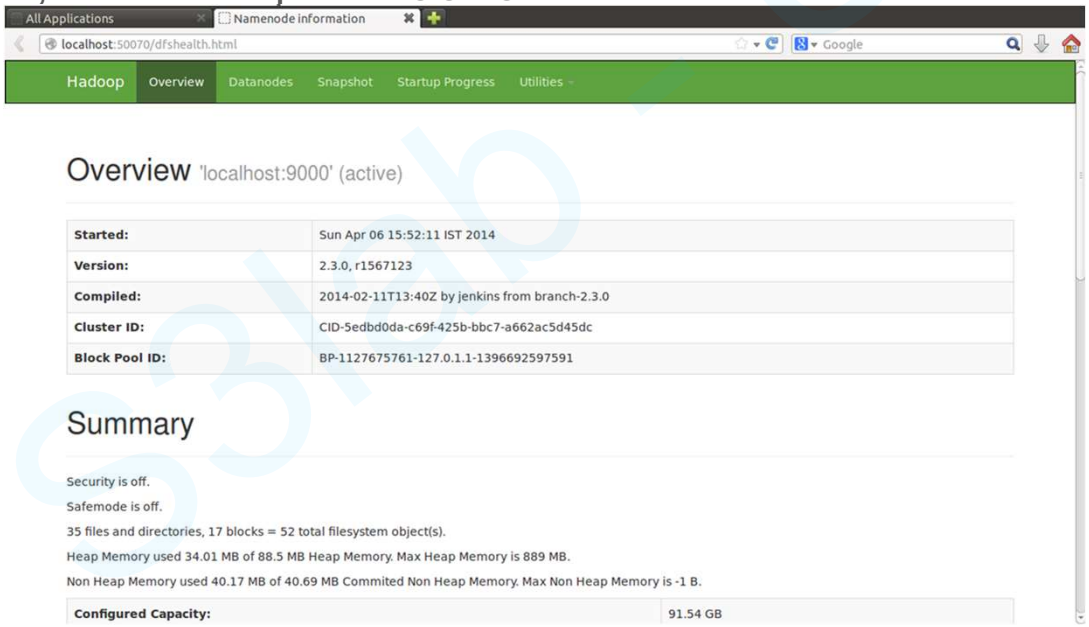
- **Hadoop Distributed File System** (**HDFS**) is designed to reliably store very large files across machines in a large cluster. It is inspired by the Google File System.
- Designed to reliably store data on **commodity hardware** (crash all the time)
- Intended for Large files and Batch inserts
- Distribute large data file into **blocks**
- Each block is replicated on multiple (slave) nodes
- HDFS component is divided into two sub-components: Name node and Data node

HDFS

- **NameNode**:
  - Master of the system, daemon runs on the **master machine**
  - Maintains, monitoring and manages the **blocks** which are present on the **DataNodes**
  - records the metadata of the files like the location of blocks, file size, permission, hierarchy etc.
  - captures all the changes to the metadata like deletion, creation and renaming of the file in edit logs.
  - It regularly receives **heartbeat** and block reports from the DataNodes.

HDFS

- All of the Hadoop server processes (daemons) serve a web UI. For NameNode, it was on port **50070**.



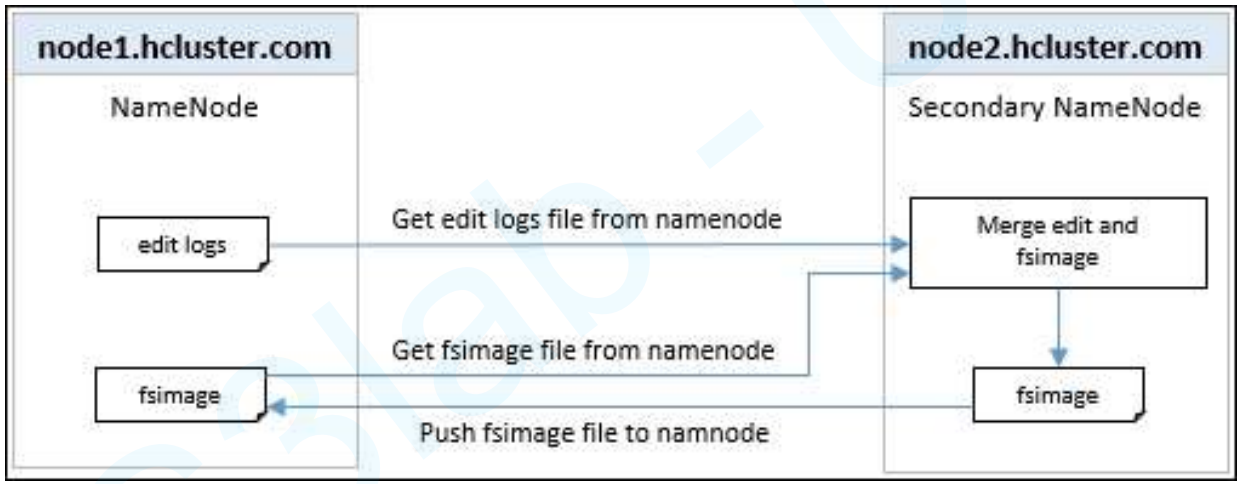
HDFS

Secondary namenode

- The secondary namenode daemon is responsible for performing periodic housekeeping functions for namenode.
- It creates checkpoints of the filesystem metadata (fsimage) present in namenode by merging the edits logfile and the fsimage file from the namenode daemon.
- In case the namenode daemon fails, this checkpoint could be used to rebuild the filesystem metadata.
- Checkpoints are done in intervals, thus checkpoint **data could be slightly outdated**. Rebuilding the fsimage file using such a checkpoint **could lead to data loss.**
- It is recommended that the secondary namenode daemon be hosted on a separate machine for large clusters.
- The checkpoints are created by merging the edits logfiles and the fsimage file from the namenode daemon.

HDFS

Secondary namenode

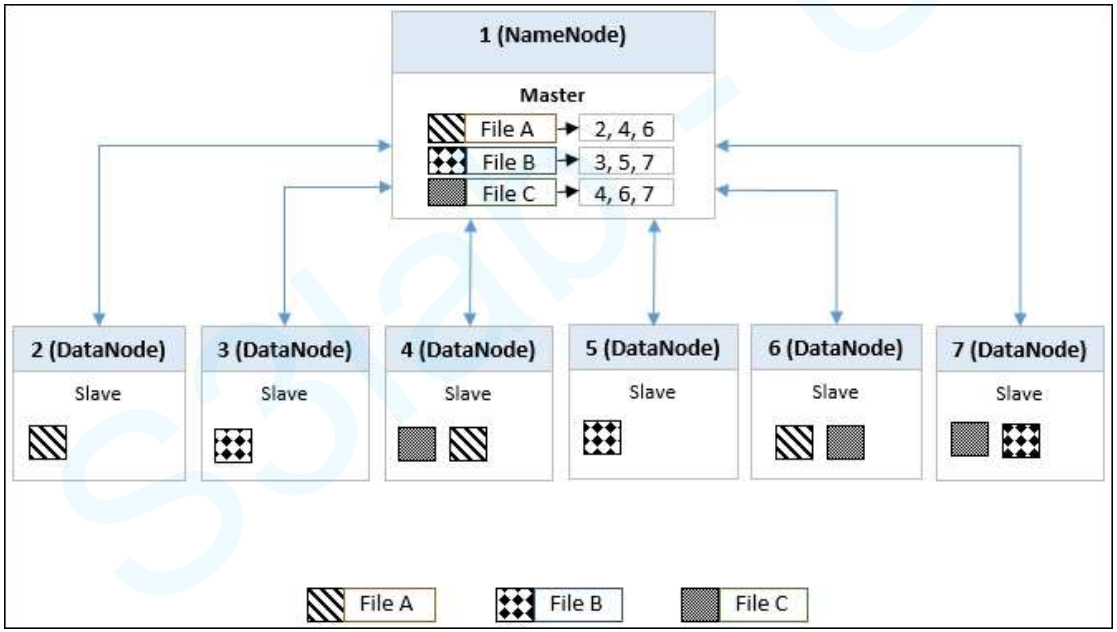


HDFS

- **DataNode**:
  - DataNode runs on the **slave machine**.
  - It stores the actual business data.
  - It serves the read-write request from the user.
  - DataNode does the ground work of creating, replicating and deleting the blocks on the command of NameNode.
  - After every **3 seconds**, by default, it sends **heartbeat** to NameNode reporting the health of HDFS.

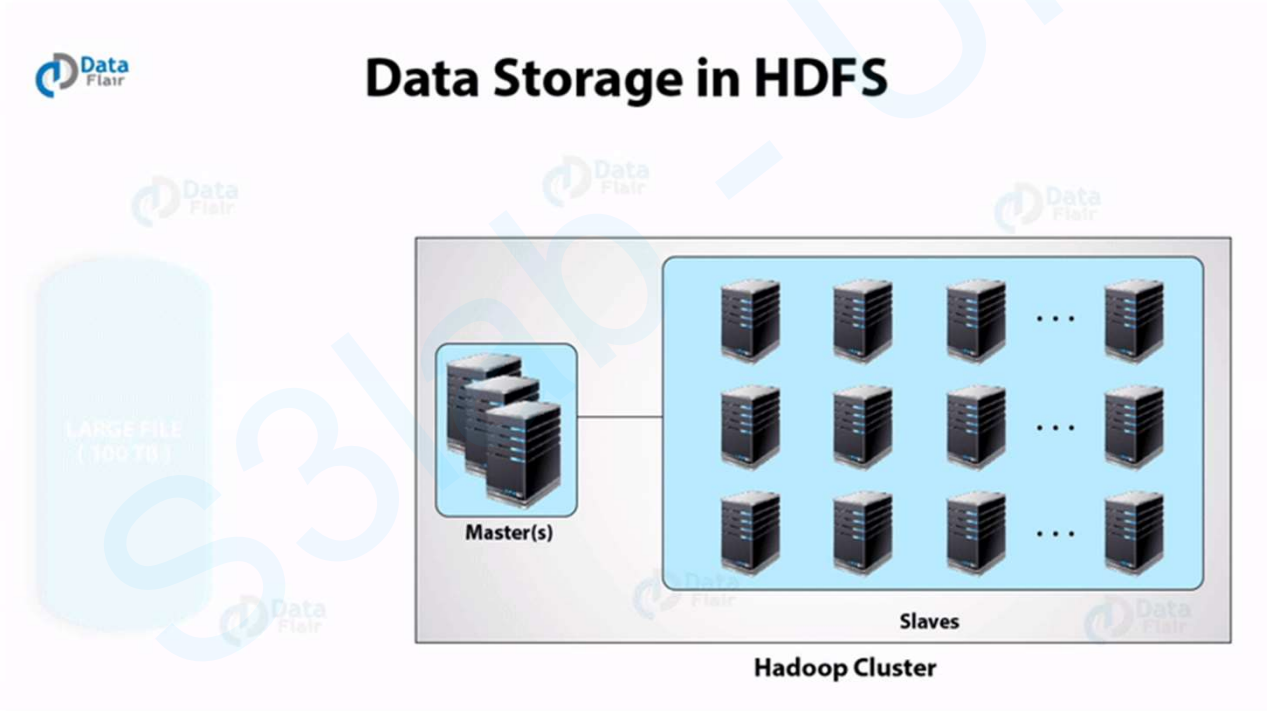
HDFS

Data node



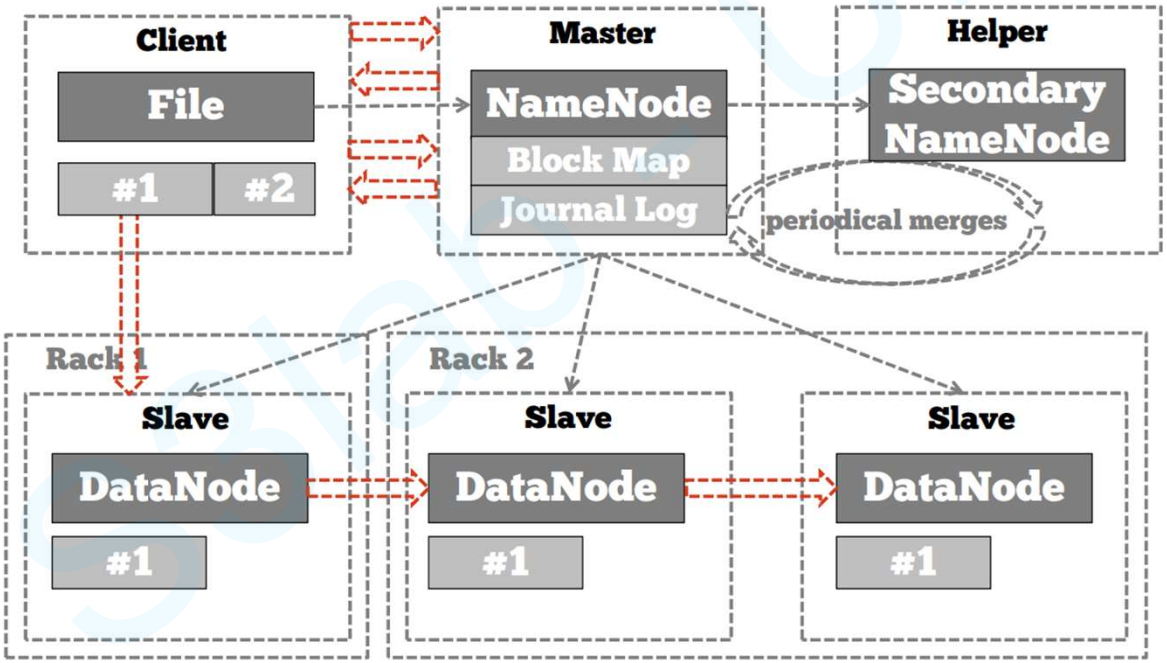
HDFS

Architecture



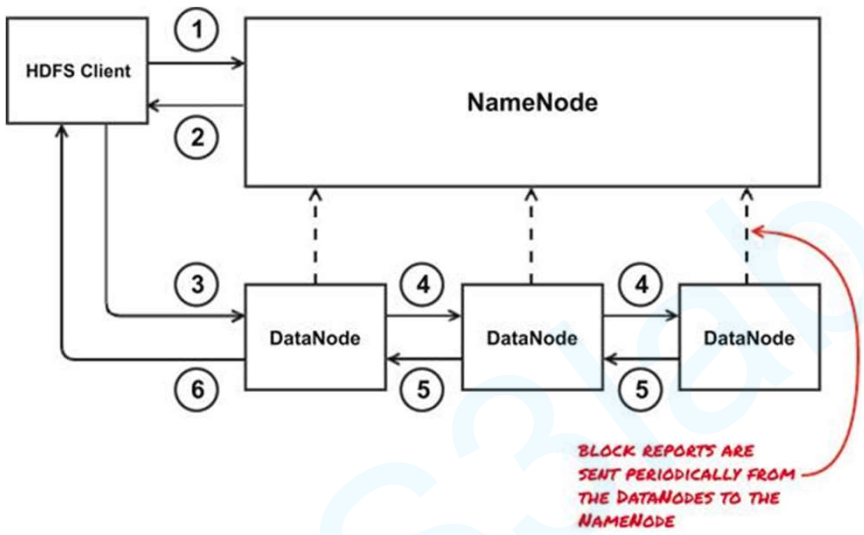
HDFS

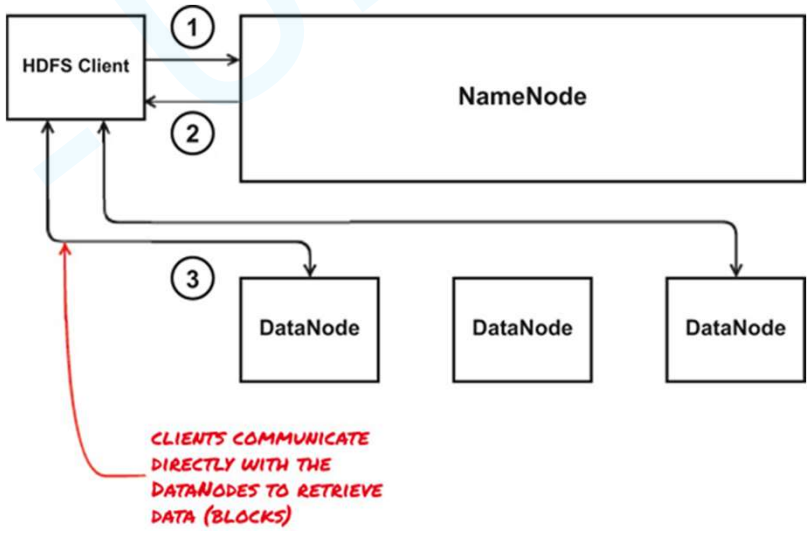
Architecture



HDFS

Write & Read files





Map Reduce

- **Programming model** for **distributed computations** at a massive scale
- Execution framework for organizing and performing such computations
- **Data locality** is king
- MapReduce component is again divided into two sub-components: JobTracker and TaskTracker
- JobTracker: takes care of all the job scheduling and assign tasks to Task Trackers.
- TaskTracker: a node in the cluster that accepts tasks - **Map**, **Reduce** & **Shuffle** operations - from jobtracker

Map Reduce

JobTracker

- The **jobtracker** daemon is responsible for accepting job requests from a client and scheduling/assigning tasktrackers with tasks to be performed.
- The jobtracker daemon tries to assign tasks to the tasktracker daemon on the datanode daemon where the data to be processed is stored. This feature is called **data locality**.
- If that is not possible, it will at least try to assign tasks to tasktrackers within the same physical server rack.
- If for some reason the node hosting the datanode and tasktracker daemons fails, the jobtracker daemon assigns the task to another tasktracker daemon where the replica of the data exists. This is possible because of the replication factor configuration for HDFS where the data blocks are replicated across multiple datanodes. This ensures that the job does not fail even if a node fails within the cluster.

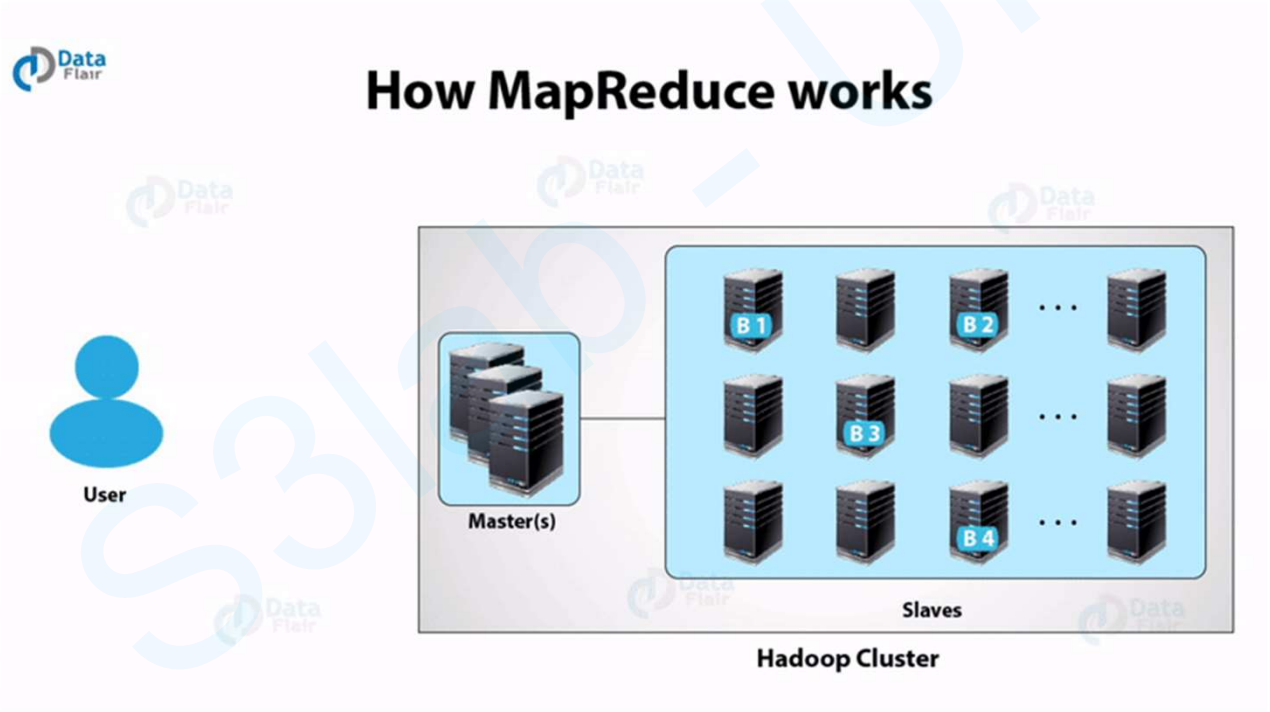
Map Reduce

TaskTracker

- The tasktracker daemon is a daemon that accepts tasks (map, reduce, and shuffle) from the jobtracker daemon.
- The tasktracker daemon is the daemon that performs the actual tasks during a MapReduce operation.
- The tasktracker daemon sends a heartbeat message to jobtracker, periodically, to notify the jobtracker daemon that it is alive.
- Along with the heartbeat, it also sends the free slots available within it, to process tasks.
- The tasktracker daemon starts and monitors the map, and reduces tasks and sends progress/status information back to the jobtracker daemon.

Map Reduce

flow

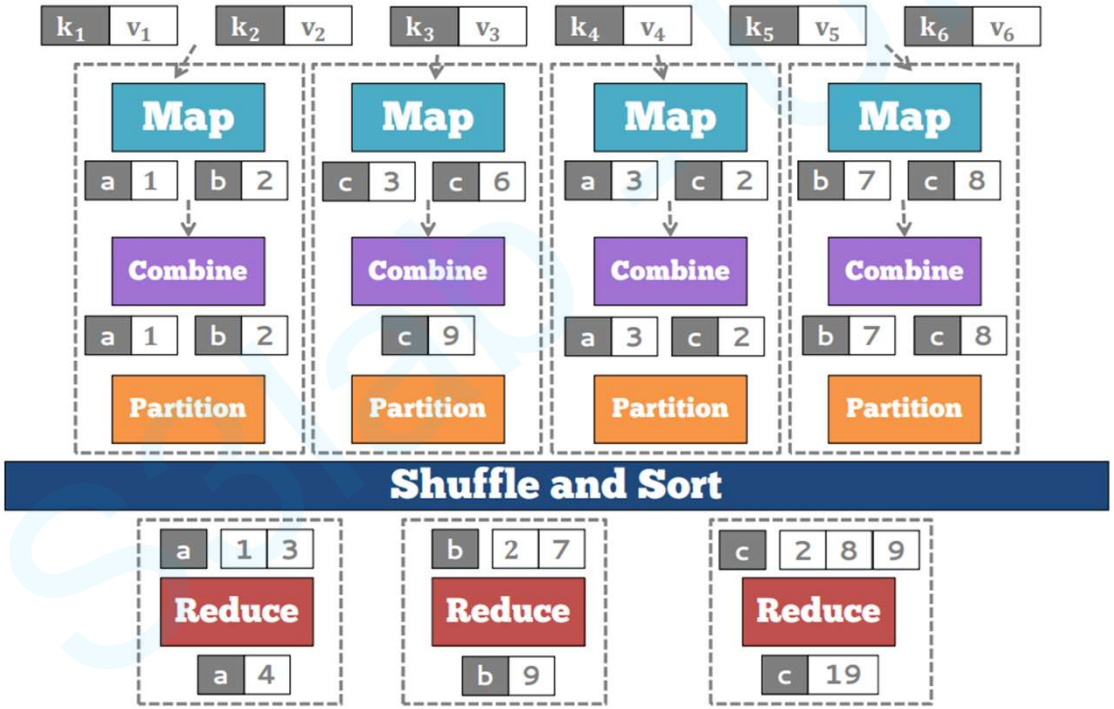


Map Reduce

- The **Mapper**:
  - Each block is processed in isolation by a map task called mapper
  - Map task runs on the node where the block is stored
  - Iterate over a large number of records
  - Extract something of interest from each
- **Shuffle** and **sort** intermediate results
- The **Reducer**:
  - Consolidate result from different mappers
  - Aggregate intermediate results
  - Produce final output

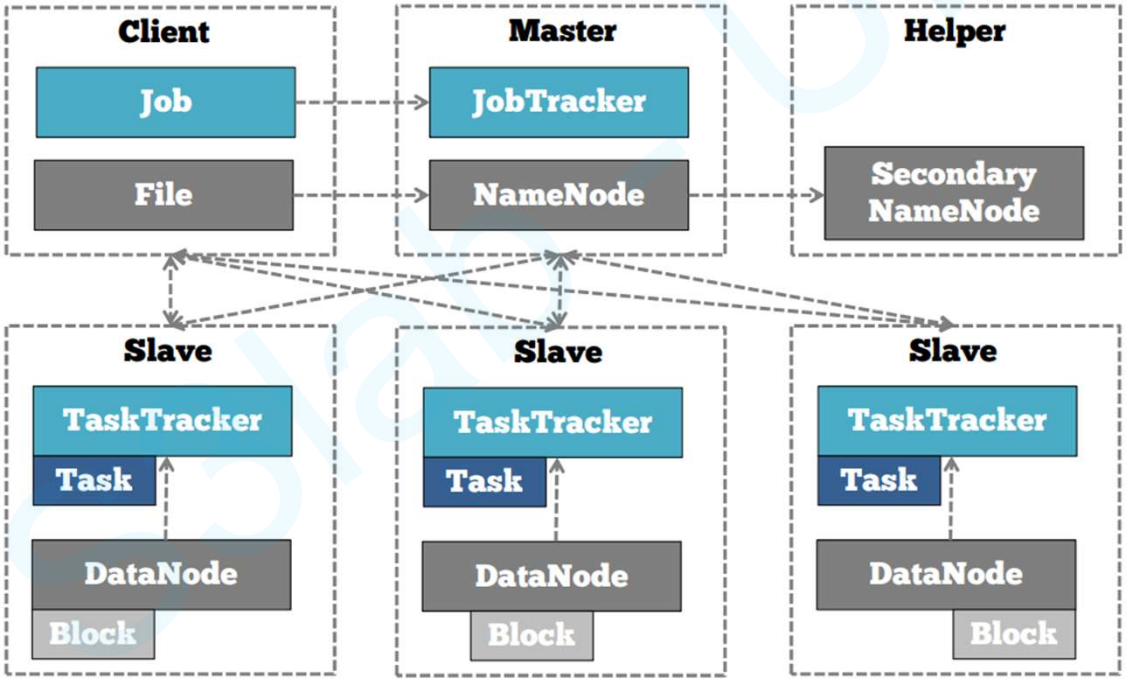
Map Reduce

flow



Map Reduce

Combined Hadoop architecture



Map Reduce

Good for ...

- Embarrassingly parallel algorithms
- Summing, grouping, filtering, joining
- Off-line batch jobs on massive data sets
- Analyzing an entire large dataset

Map Reduce

Not good for …

- Jobs that need shared state/coordination
  - Tasks are shared-nothing
  - Shared-state requires scalable state store
- Iterative jobs (i.e., graph algorithms)
  - Each iteration must read/write data to disk
  - I/O and latency cost of an iteration is high
- Low-latency jobs
- Jobs on small datasets
- Finding individual records

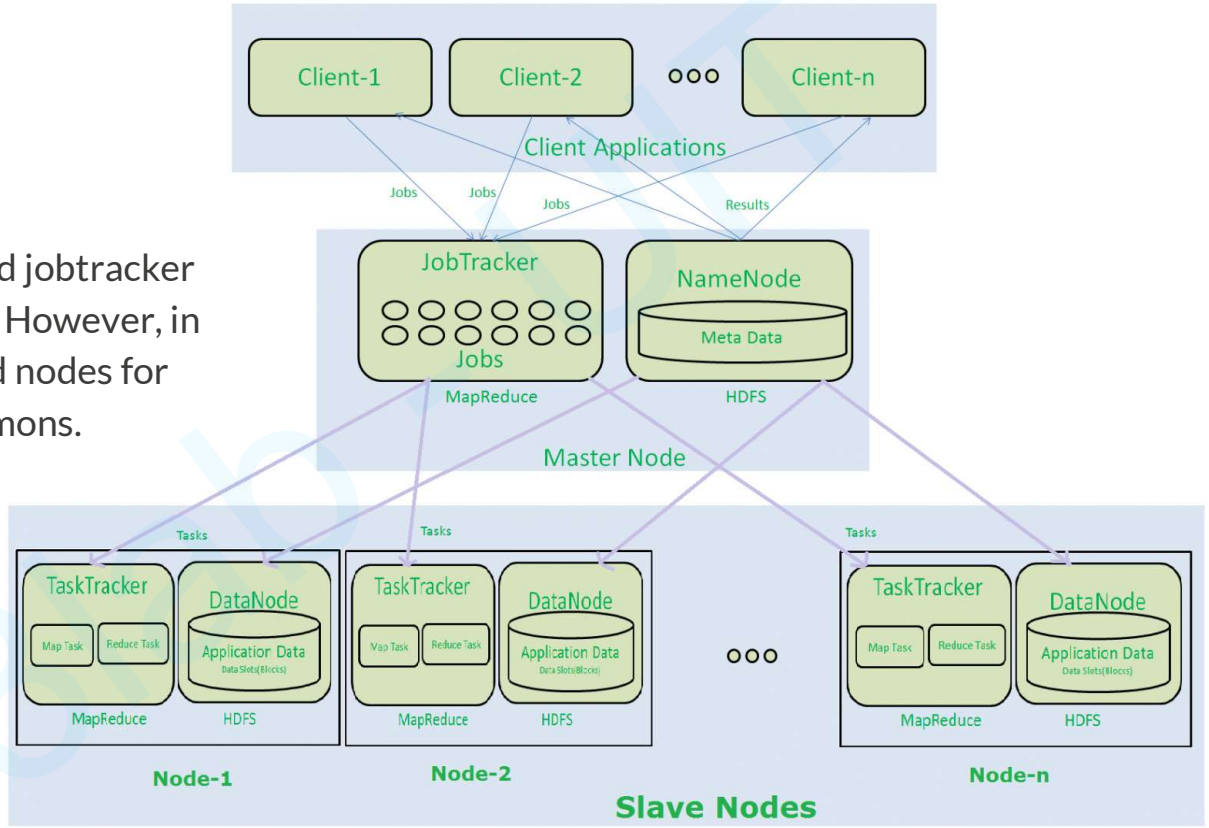
Map Reduce

limitations

- Scalability
  - Maximum cluster size ~ 4,500 nodes, concurrent tasks – 40,000
  - Coarse synchronization in JobTracker
- Availability
  - Failure kills all queued and running jobs
- Hard partition of resources into map & reduce slots
  - Low resource utilization
- Lacks support for alternate paradigms and services
  - Iterative applications implemented using MapReduce are 10x slower

Hadoop 1.x

In small clusters, the namenode and jobtracker daemons reside on the same node. However, in larger clusters, there are dedicated nodes for the namenode and jobtracker daemons.

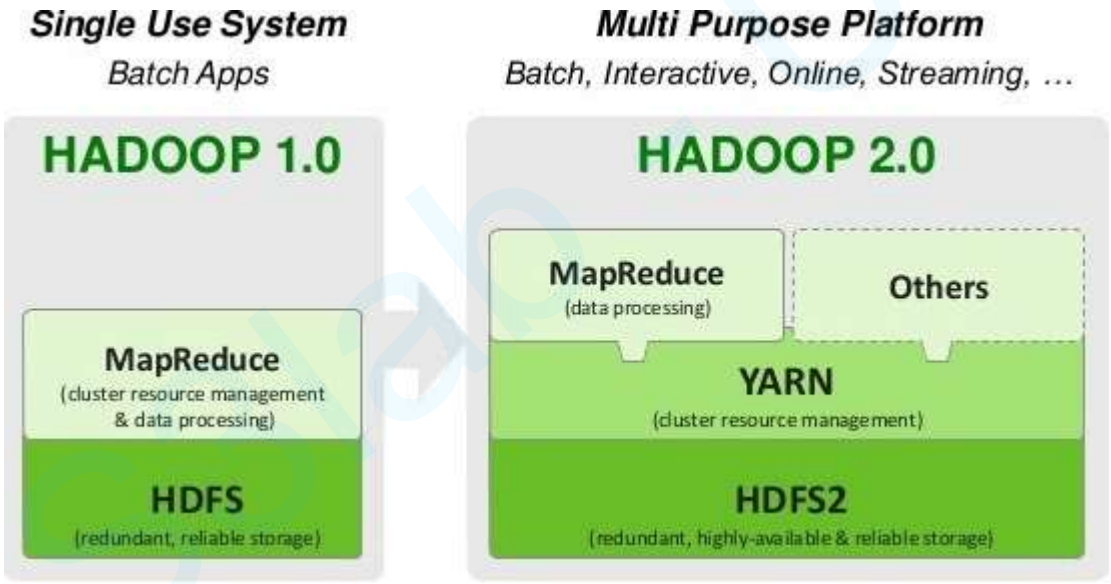


Hadoop 1.x

By default, the ports for the Hadoop daemons are:

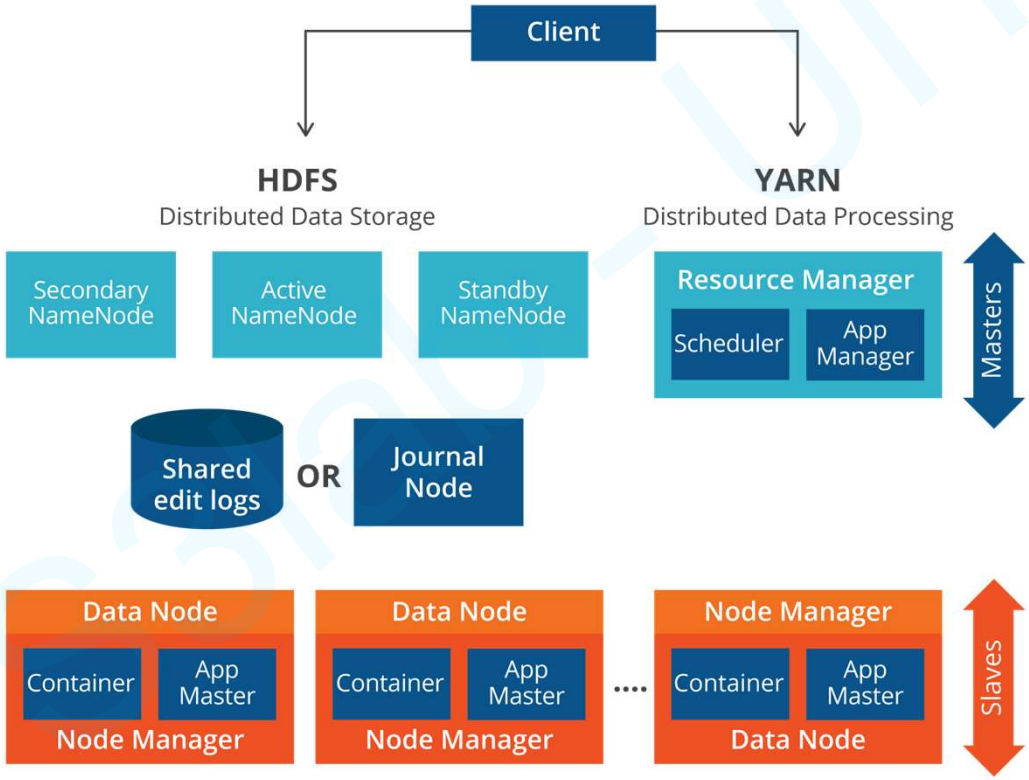
The Hadoop daemon	Port
Namenode	50070
Secondary namenode	50090
Jobtracker	50030
Datanode	50075
Tasktracker	50060

Hadoop 2.x



HDFS2, YARN, MapReduce: Three Pillars of Hadoop 2

Apache Hadoop 2.0 and YARN



Hadoop 2.x

Following are the four main improvements in Hadoop 2.0 over Hadoop 1.x:

- **HDFS Federation** – horizontal scalability of NameNode
- **NameNode High Availability** – NameNode is no longer a Single Point of Failure
- **YARN** – ability to process Terabytes and Petabytes of data available in HDFS using Non-MapReduce applications such as MPI, GIRAPH
- **Resource Manager** – splits up the two major functionalities of overburdened JobTracker (resource management and job scheduling/monitoring) into two separate daemons: a global Resource Manager and per-application ApplicationMaster

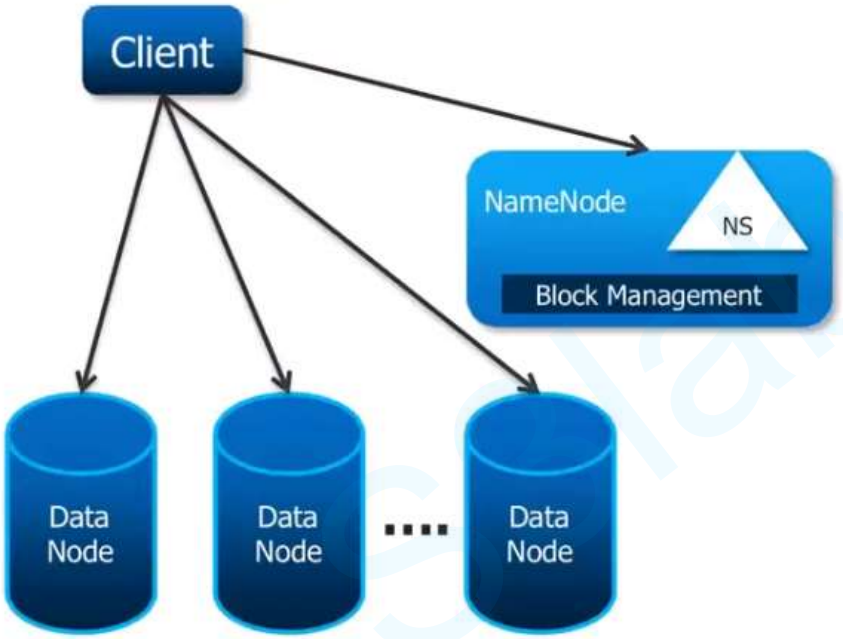
There are additional features such as **Capacity Scheduler** (Enable Multi-tenancy support in Hadoop), **Data Snapshot**, **Support for Windows**, **NFS access**, enabling increased Hadoop adoption in the Industry to solve Big Data problems.

Limitation of Hadoop 1.x

- No horizontal scalability of NameNode
- Does not support NameNode High Availability
- Overburdened JobTracker
- Not possible to run Non-MapReduce Big Data Applications on HDFS
- Does not support Multi-tenancy

Limitation of Hadoop 1.x

No Horizontal Scalability of NameNode

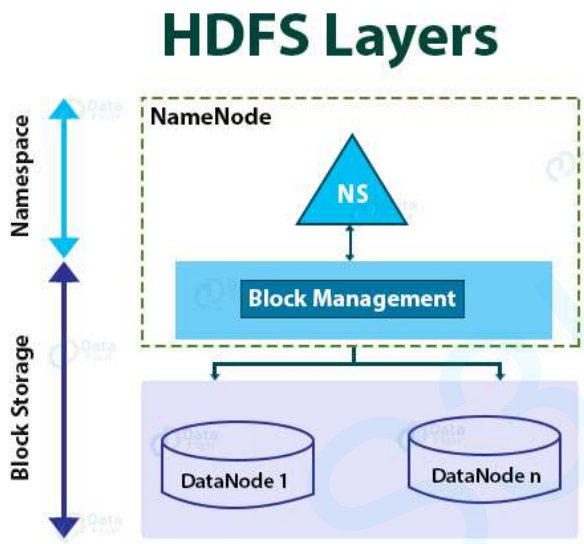


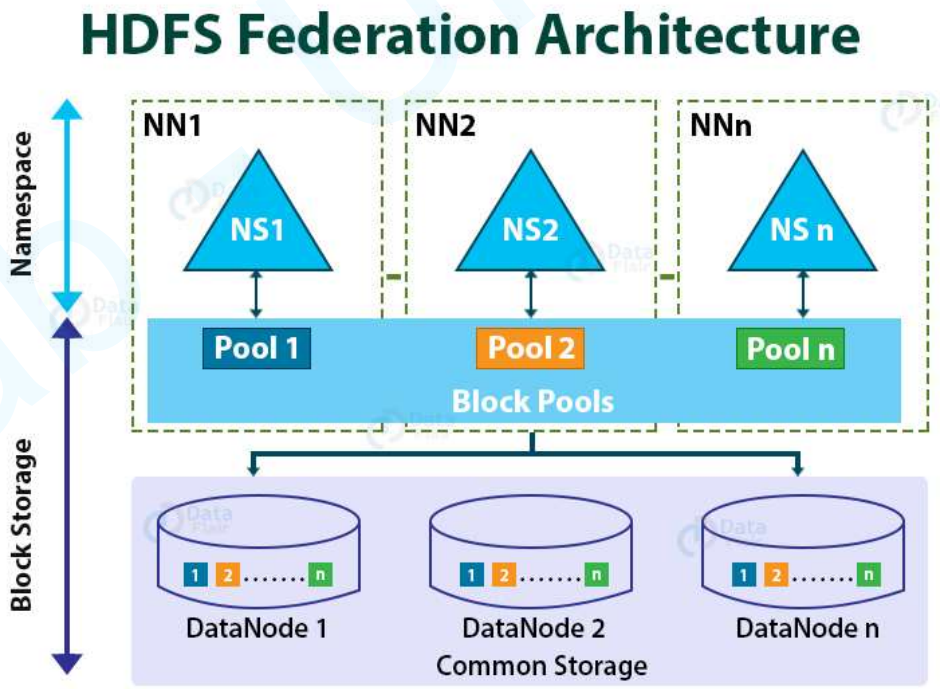
Challenges:

- Meta is stored in NameNode memory
- Bottleneck after ~4000 nodes
- Results in cascading failures

Hadoop 2.x

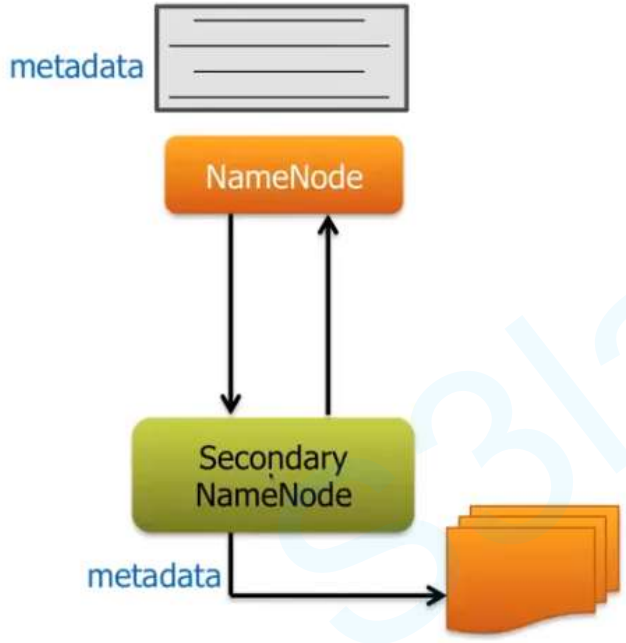
HDFS Federation





Limitation of Hadoop 1.x

NameNode – Single point of failure

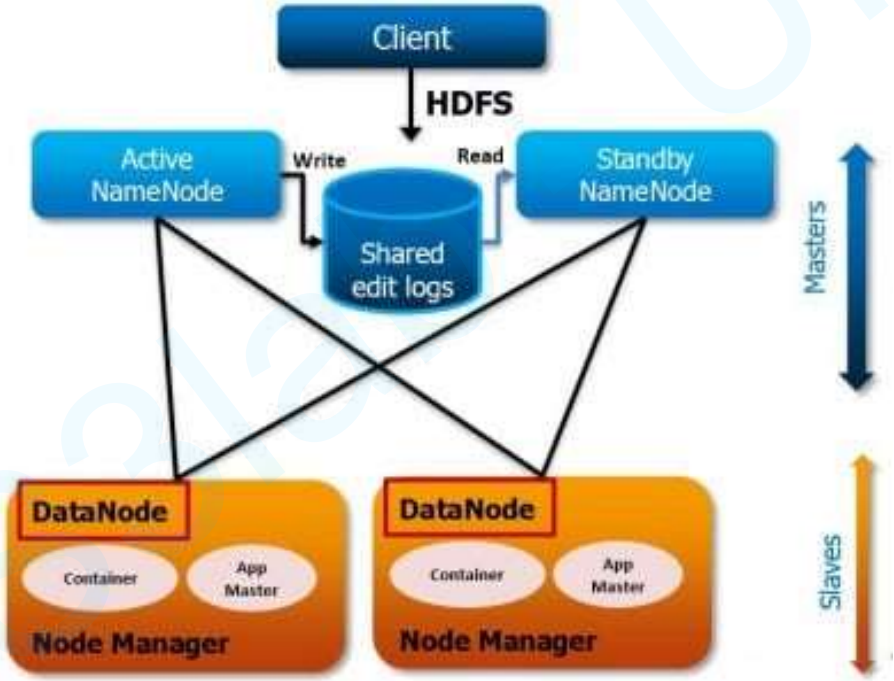


Secondary NameNode:

- "Not a hot standby" for the NameNode
- Connects to NameNode regularly
- Housekeeping, backup of NameNode metadata
- Saved metadata can build a failed NameNode

Hadoop 2.x

NameNode High Availability



Limitation of Hadoop 1.x

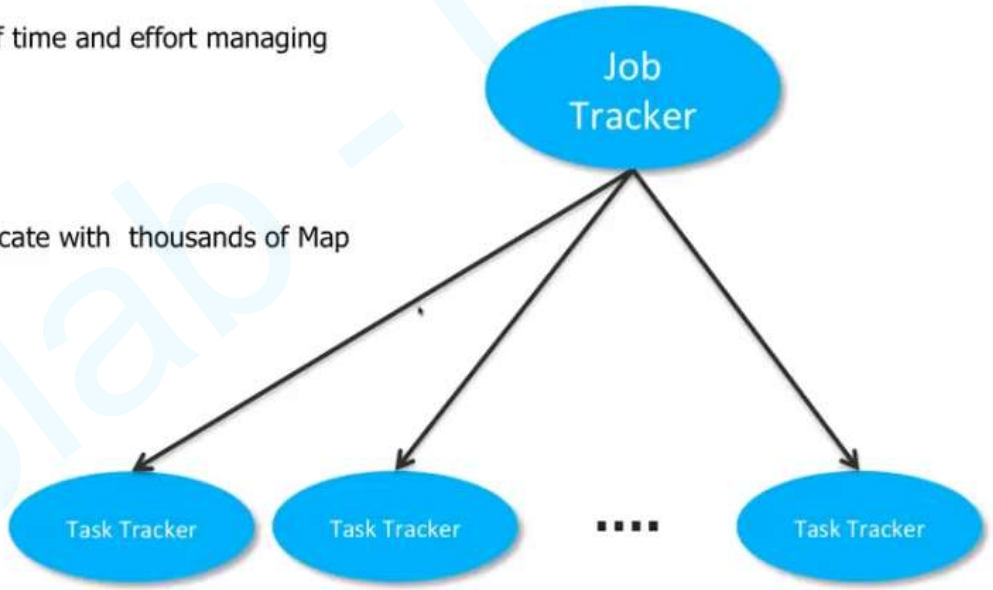
Overburdened JobTracker

CPU

- ✓ Spends a very significant portion of time and effort managing the life cycle of applications

Network

- ✓ Single Listener Thread to communicate with thousands of Map and Reduce Jobs

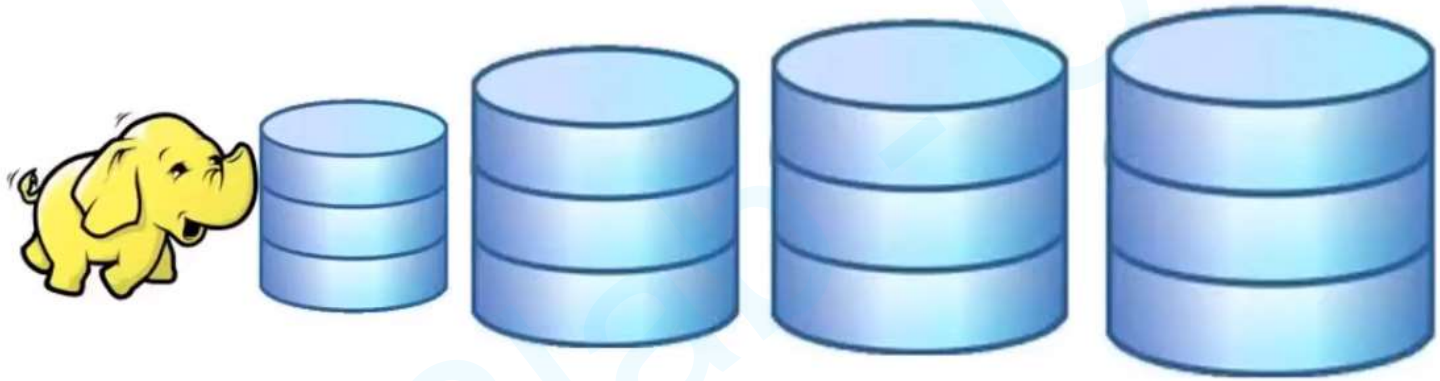


Hadoop 2.x

YARN (Yet Another Resource Nigotiator)

Limitation of Hadoop 1.x

Unutilized data in HDFS



Challenges:

- ✓ Only MapReduce processing can be achieved
- ✓ Alternate Data Storage is needed for other processing such as Real-time or Graph analysis
- ✓ Doesn't support Multi-Tenacy

Hadoop 2.x

Non-MapReduce Big Data Application

Limitation of Hadoop 1.x

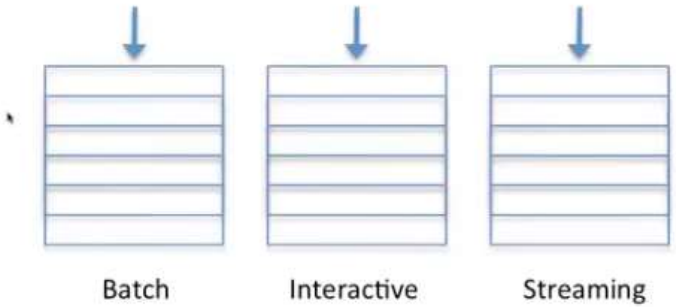
No Multi-tenancy Support

Hadoop 2.x

Capacity Scheduler – Multi-tenancy Support

Features:

- ✓ Different types of jobs are organized in different queues
- ✓ Queue shares as %'s of cluster
- ✓ Each queue has an associated priority
- ✓ FIFO scheduling within each queue
- ✓ Security ensured between applications



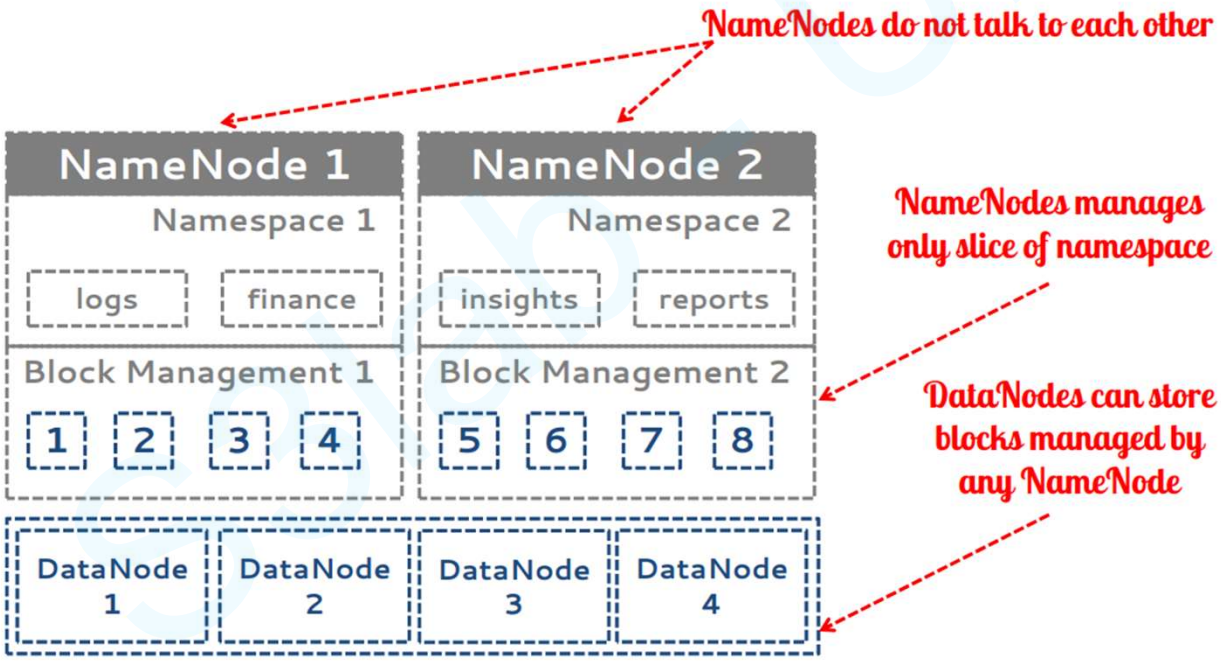
Hadoop 2.x

HDFS 2 - Federation

- Removes tight coupling of Block
- Storage and Namespace (multiple namespaces)
- Scalability & Isolation (multiple namenodes)
- High Availability
- Increased performance

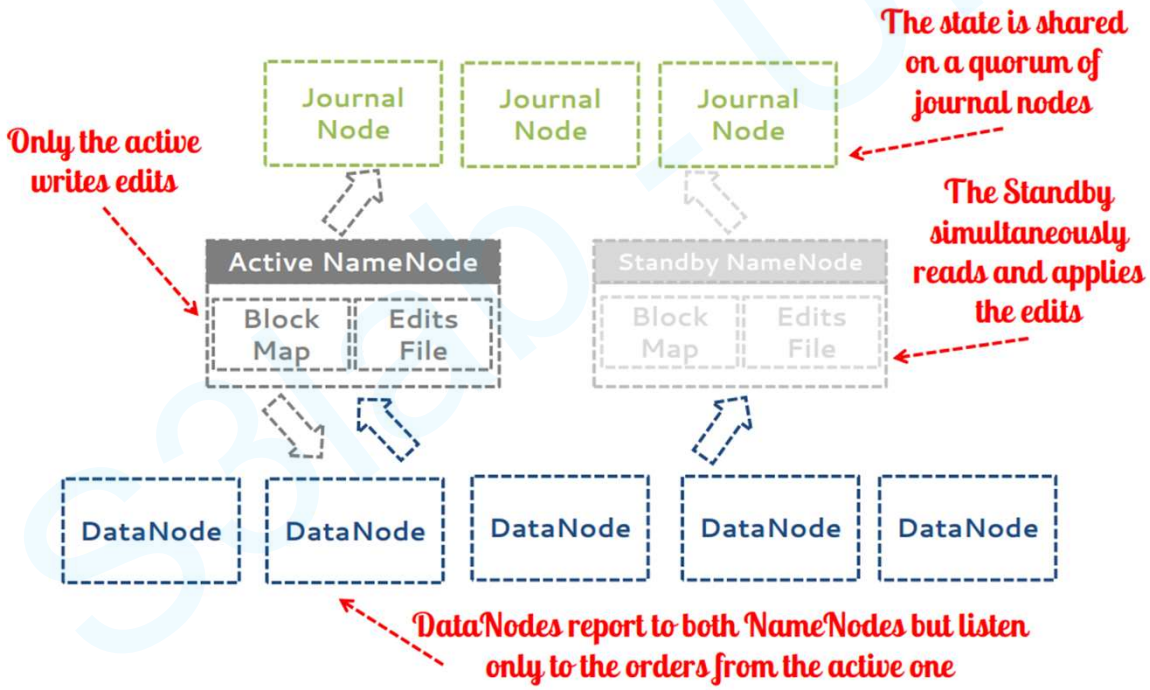
HDFS

HDFS 2 - Federation



HDFS

HDFS 2 (Federation): Quorum based storage



Map Reduce

MRv2

- Basically a porting to the YARN architecture
- MapReduce becomes a user-land library
- No need to rewrite MapReduce jobs
- Increased scalability & availability
- Better cluster utilization

Map Reduce

Example: Word Count

Input	Set of data	Bus, Car, bus, car, train, car, bus, car, train, bus, TRAIN,BUS, buS, caR, CAR, car, BUS, TRAIN
Output	Convert into another set of data (Key,Value)	(Bus,1), (Car,1), (bus,1), (car,1), (train,1), (car,1), (bus,1), (car,1), (train,1), (bus,1), (TRAIN,1),(BUS,1), (buS,1), (caR,1), (CAR,1), (car,1), (BUS,1), (TRAIN,1)

MAP

Reduce

Input **(output of Map function)**	Set of Tuples	(Bus,1), (Car,1), (bus,1), (car,1), (train,1), (car,1), (bus,1), (car,1), (train,1), (bus,1), (TRAIN,1),(BUS,1), (buS,1), (caR,1), (CAR,1), (car,1), (BUS,1), (TRAIN,1)
Output	Converts into smaller set of tuples	(BUS,7), (CAR,7), (TRAIN,4)

Map Reduce

Example: Word Count

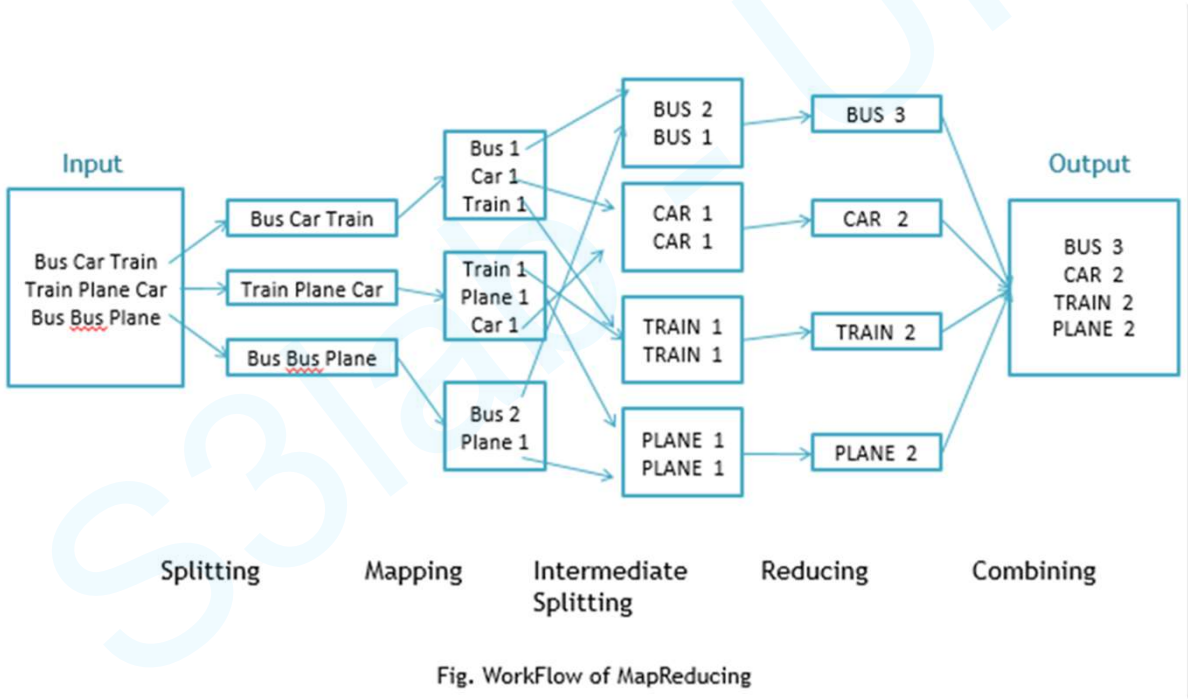


Fig. WorkFlow of MapReducing

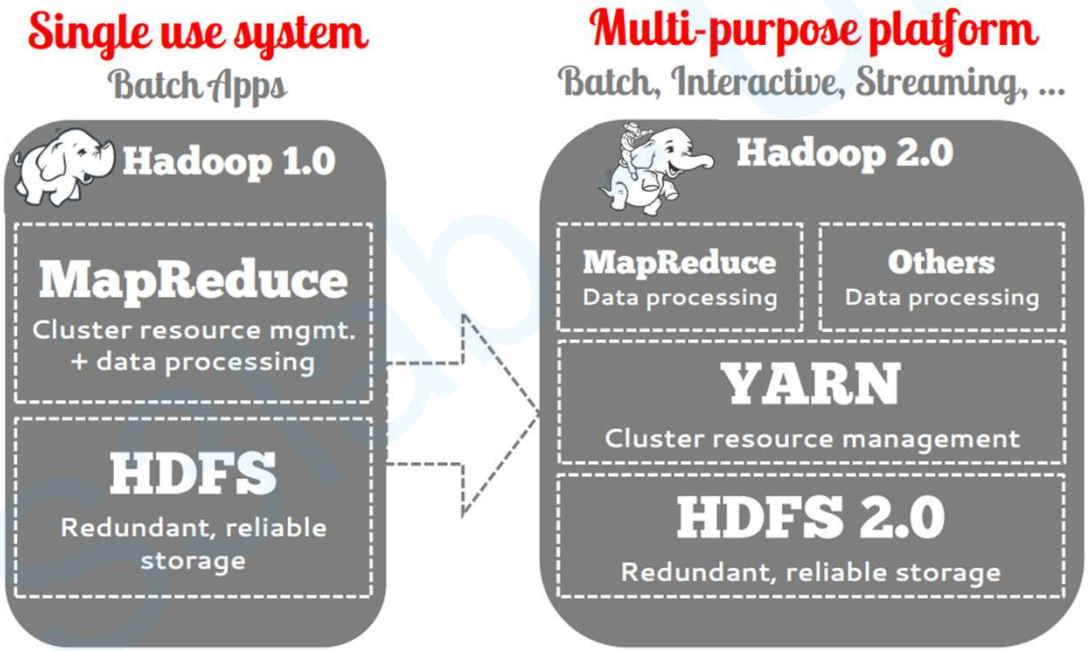
Hadoop 2.0

history

- Originally conceived & architected by the team at Yahoo!
  - Arun Murthy created the original JIRA in 2008 and now is the YARN release manager
- The team at Hortonworks has been working on YARN for 4 years:
  - 90% of code from Hortonworks & Yahoo!
- Hadoop 2.0 based architecture running at scale at Yahoo!
  - Deployed on 35,000 nodes for 6+ months

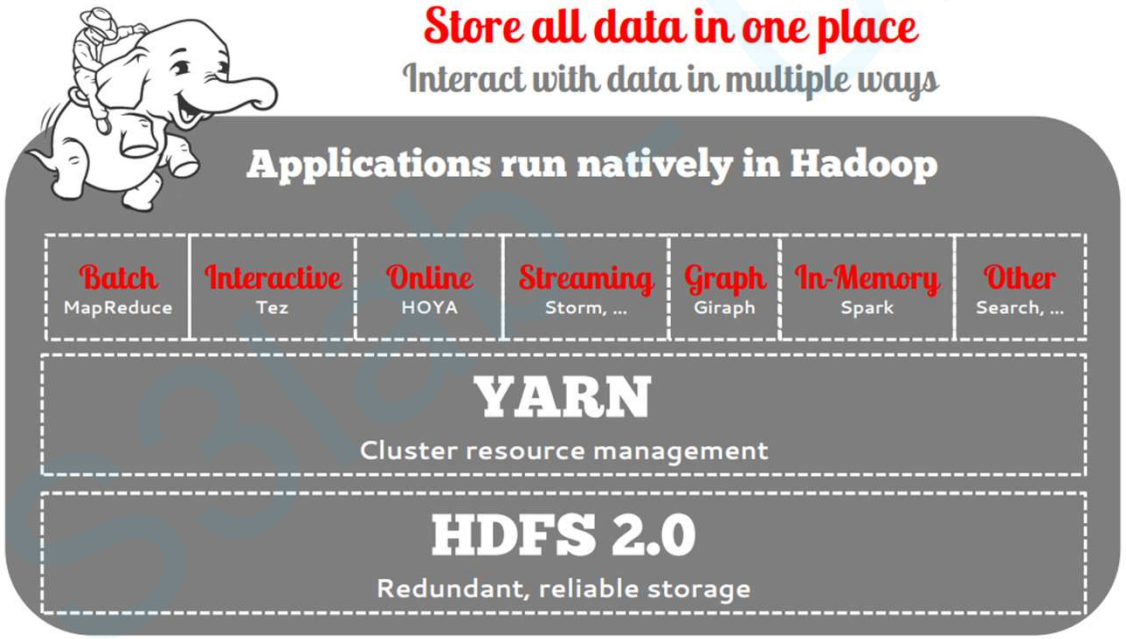
Hadoop 2.0

Next-gen platform



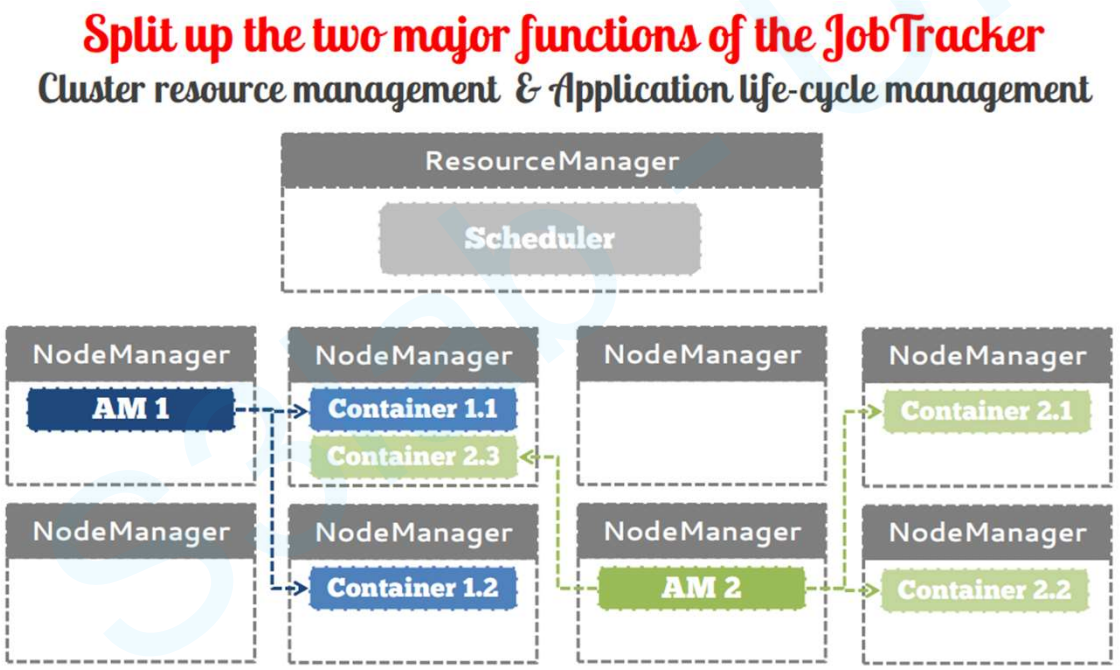
Hadoop 2.0

Taking Hadoop beyond batch

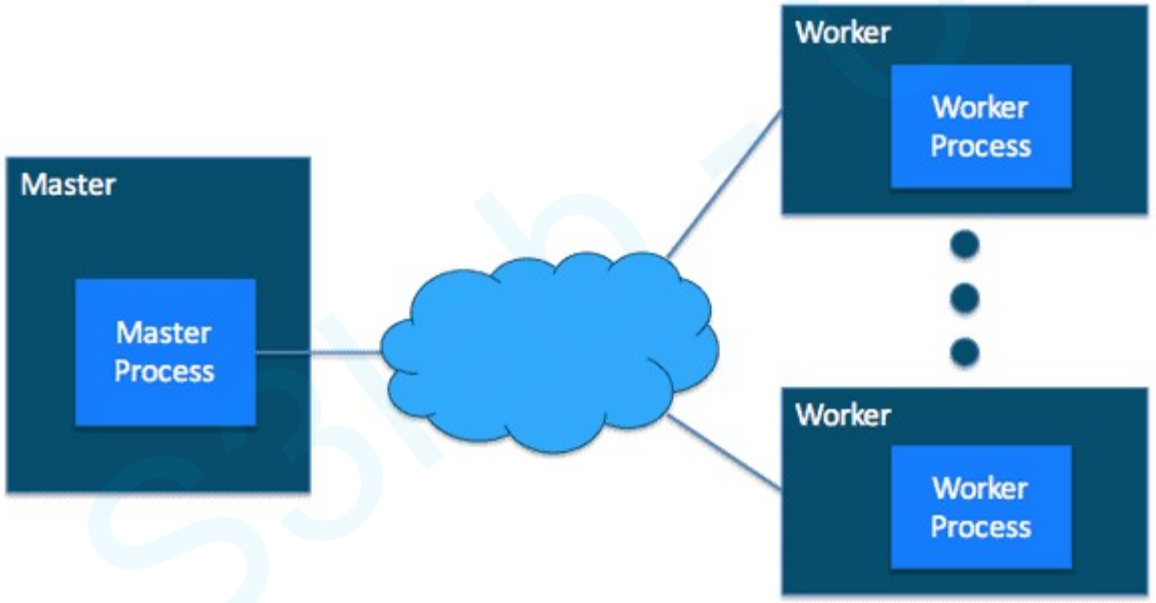


Yarn

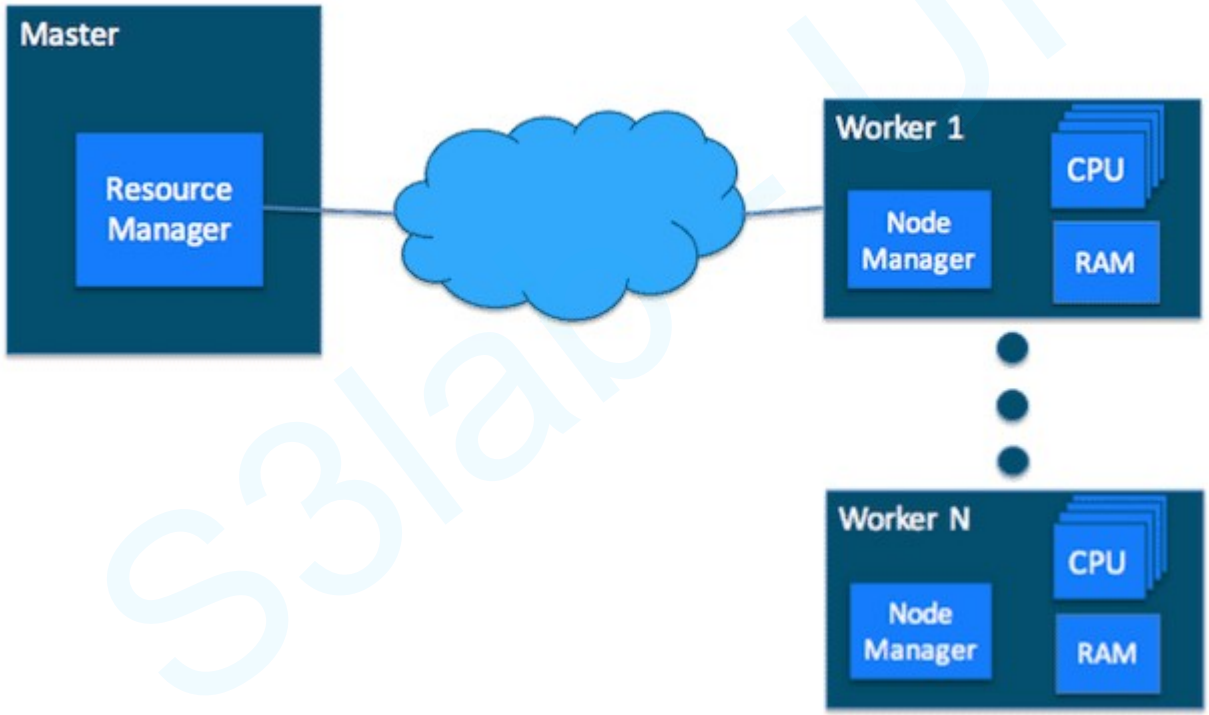
Architecture



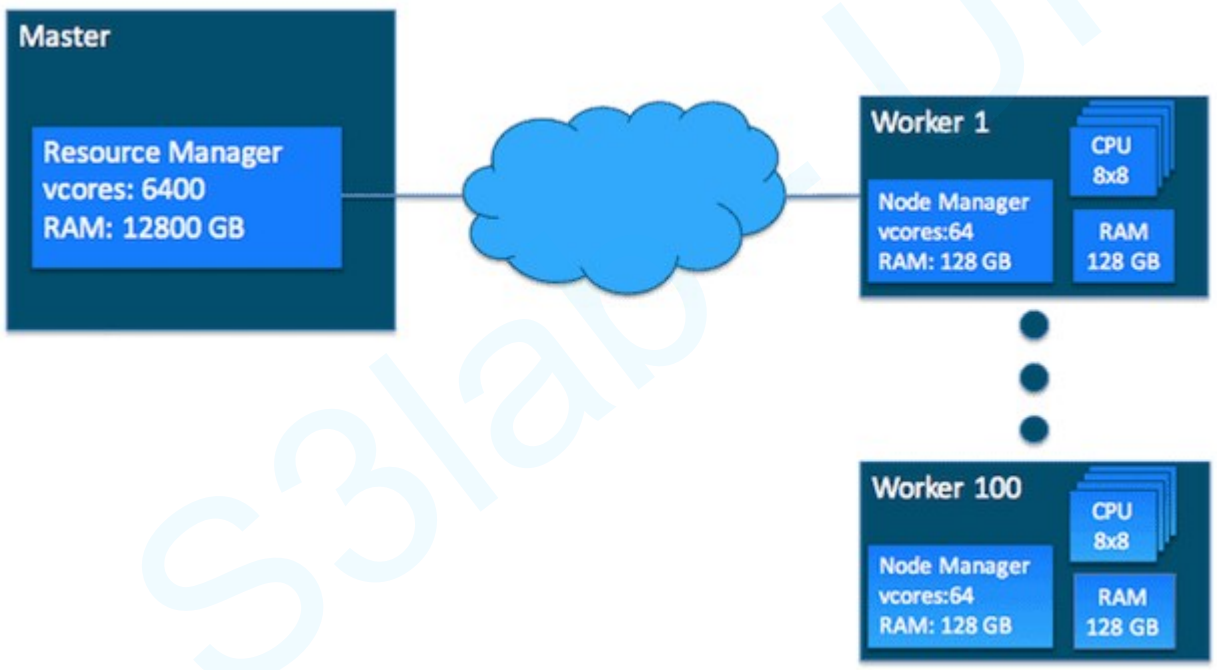
Yarn



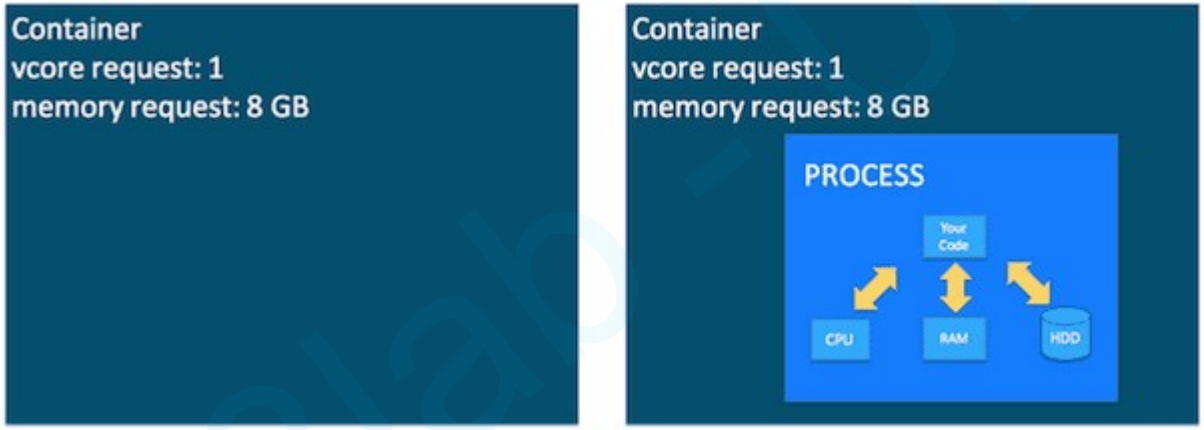
Yarn



Yarn



Yarn



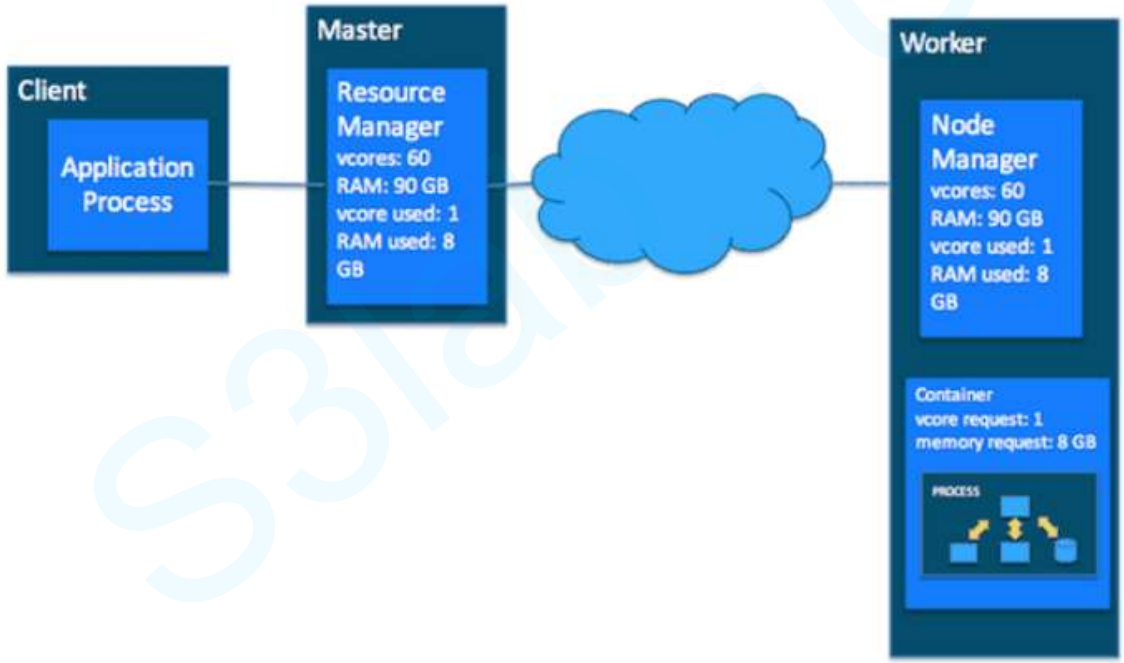
Yarn

An application running tasks on a YARN cluster consists of the following steps:

1. The application starts and talks to the ResourceManager for the cluster:

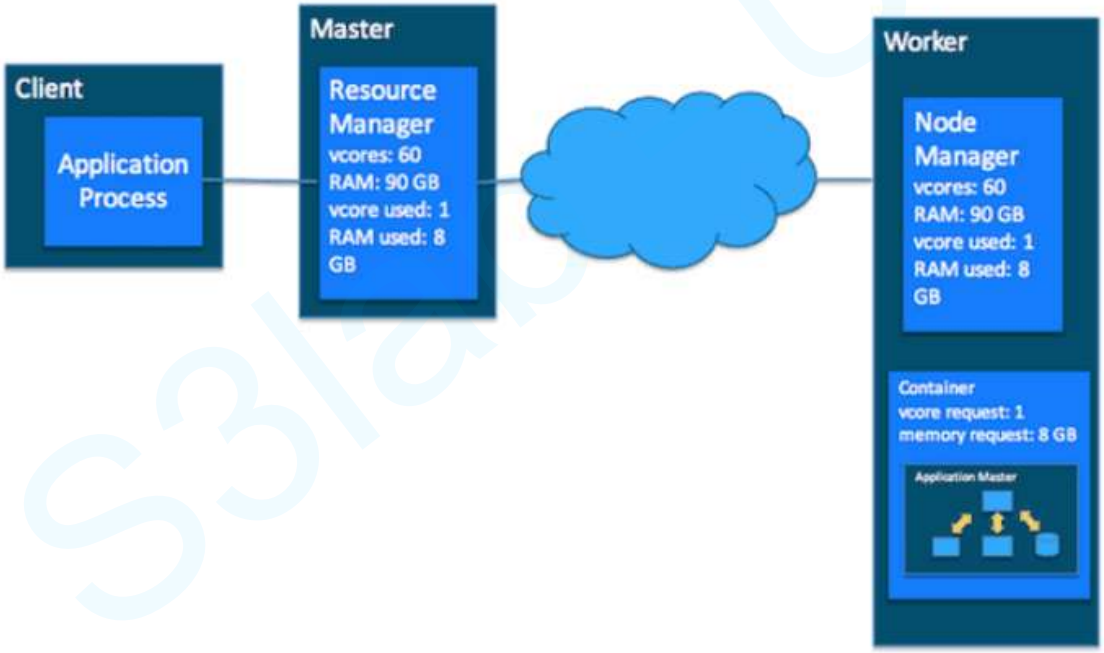
Yarn

2. The ResourceManager makes a single container request on behalf of the application:



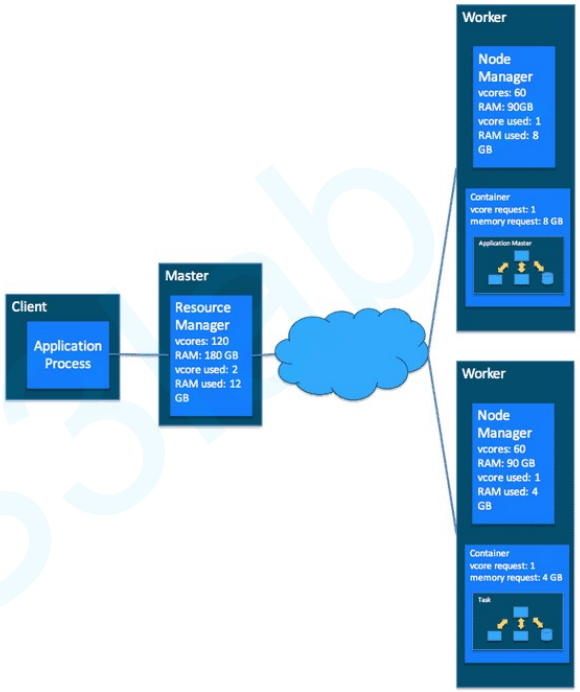
Yarn

3. The ApplicationMaster starts running within that container:



Yarn

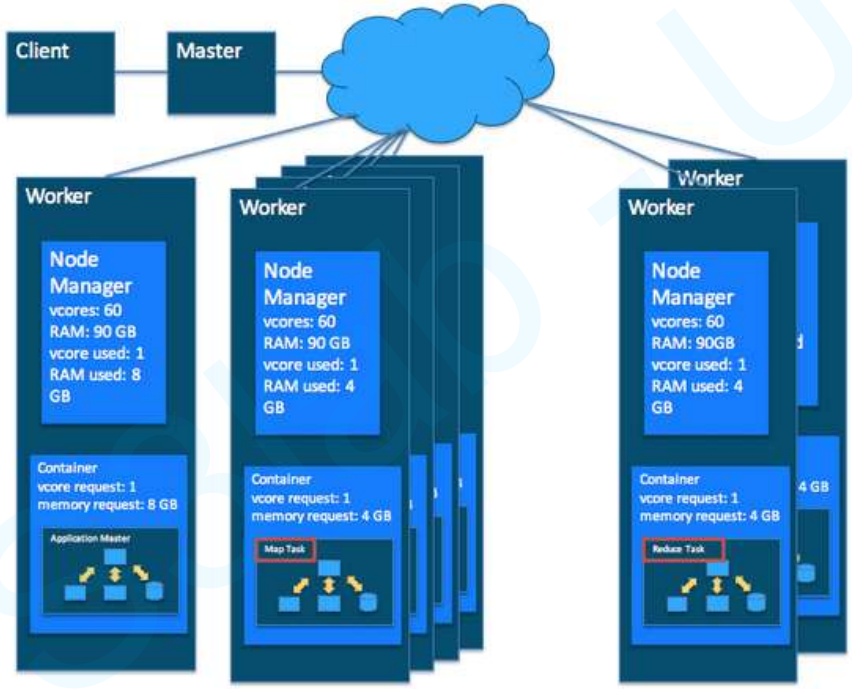
4. The ApplicationMaster requests subsequent containers from the ResourceManager that are allocated to run tasks for the application. Those tasks do most of the status communication with the ApplicationMaster allocated in Step 3):



Yarn

5. Once all tasks are finished, the ApplicationMaster exits. The last container is de-allocated from the cluster.
6. The application client exits. (The ApplicationMaster launched in a container is more specifically called a managed AM. Unmanaged ApplicationMasters run outside of YARN's control. Llama is an example of an unmanaged AM.)

MapReduce Yarn

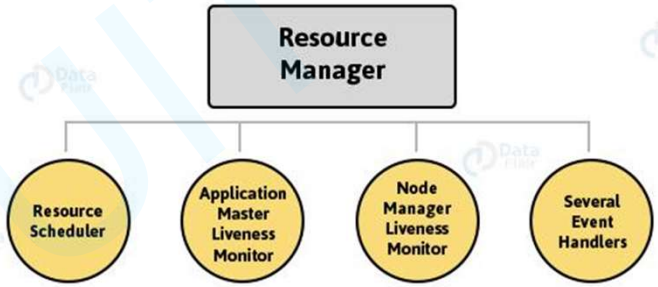


Yarn

- A cluster is made up of two or more hosts connected by an internal high-speed network. Master hosts are a small number of hosts reserved to control the rest of the cluster. Worker hosts are the non-master hosts in the cluster.
- In a cluster with YARN running, the master process is called the ResourceManager and the worker processes are called NodeManagers.
- The configuration file for YARN is named yarn-site.xml. There is a copy on each host in the cluster. It is required by the ResourceManager and NodeManager to run properly. YARN keeps track of two resources on the cluster, vcores and memory. The NodeManager on each host keeps track of the local host's resources, and the ResourceManager keeps track of the cluster's total.
- A container in YARN holds resources on the cluster. YARN determines where there is room on a host in the cluster for the size of the hold for the container. Once the container is allocated, those resources are usable by the container.
- An application in YARN comprises three parts:
- The application client, which is how a program is run on the cluster.
- An ApplicationMaster which provides YARN with the ability to perform allocation on behalf of the application.
- One or more tasks that do the actual work (runs in a process) in the container allocated by YARN.
- A MapReduce application consists of map tasks and reduce tasks.
- A MapReduce application running in a YARN cluster looks very much like the MapReduce application paradigm, but with the addition of an ApplicationMaster as a YARN requirement.

Yarn

Architecture

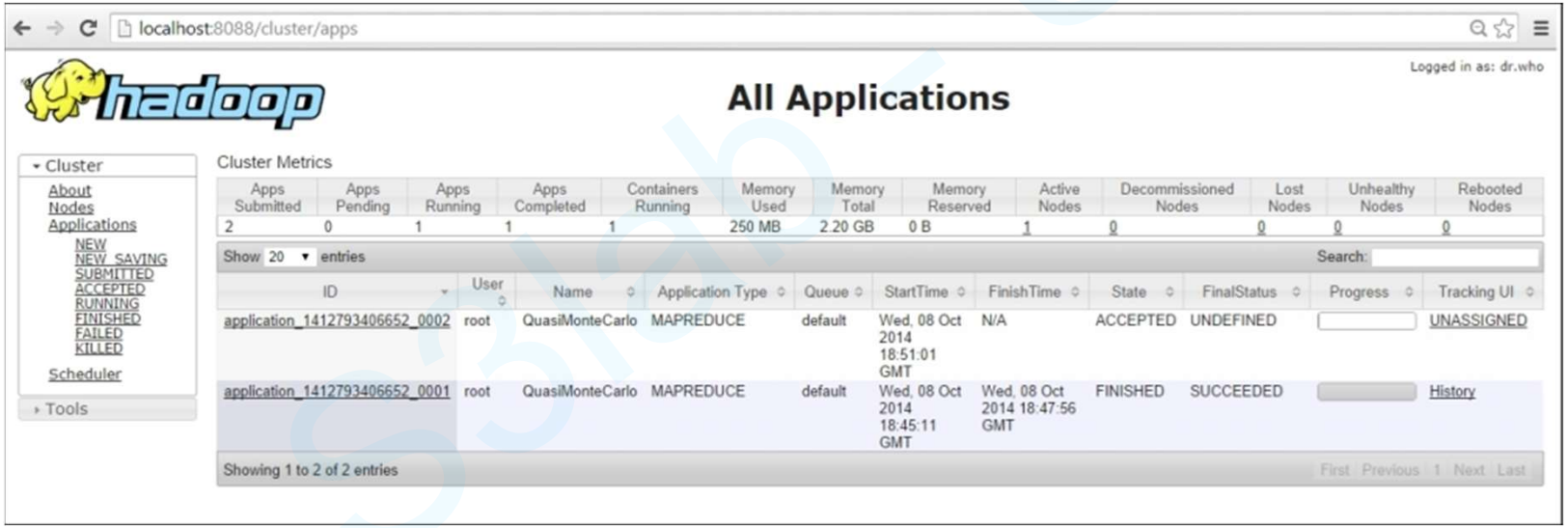


- Resource Manager
  - Resource Manager runs on the **master node**.
  - It knows where the location of slaves (Rack Awareness).
  - It is aware about how much resources each slave have.
  - Resource Scheduler is one of the important service run by the Resource Manager.
  - Resource Scheduler decides how the resources get assigned to various tasks.
  - Application Manager is one more service run by Resource Manager.
  - Application Manager negotiates the first container for an application.
  - Resource Manager keeps track of the **heart beats** from the Node Manager.

Yarn

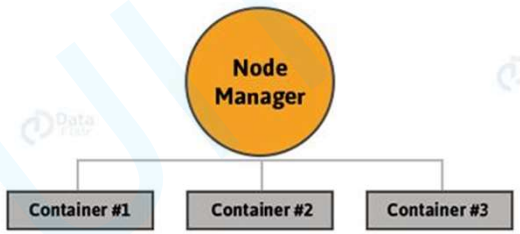
Architecture

- Resource Manager serves an embedded Web UI on port 8088



Yarn

Architecture

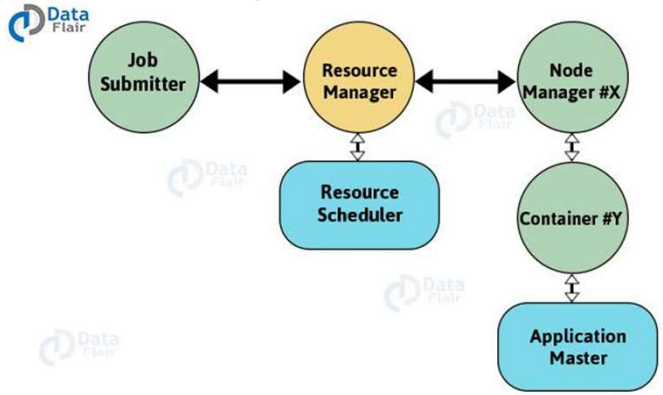


- Node Manager
  - It runs on **slave machines**.
  - It manages **containers**. Containers are nothing but a fraction of Node Manager's resource capacity
  - Node manager monitors resource utilization of each container.
  - It sends **heartbeat** to Resource Manager.

Yarn

Architecture

- Job submitter
  - The client submits the job to Resource Manager.
  - Resource Manager contacts Resource Scheduler and allocates container.
  - Now Resource Manager contacts the relevant Node Manager to launch the container.
  - Container runs Application Master



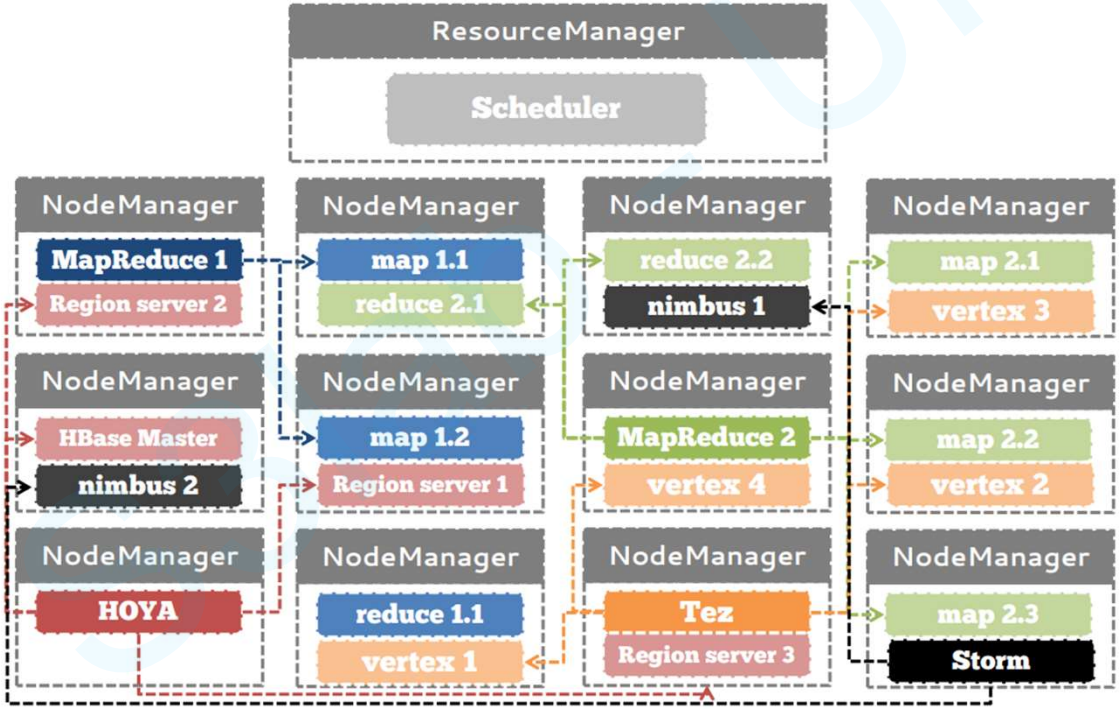
Yarn

Architecture

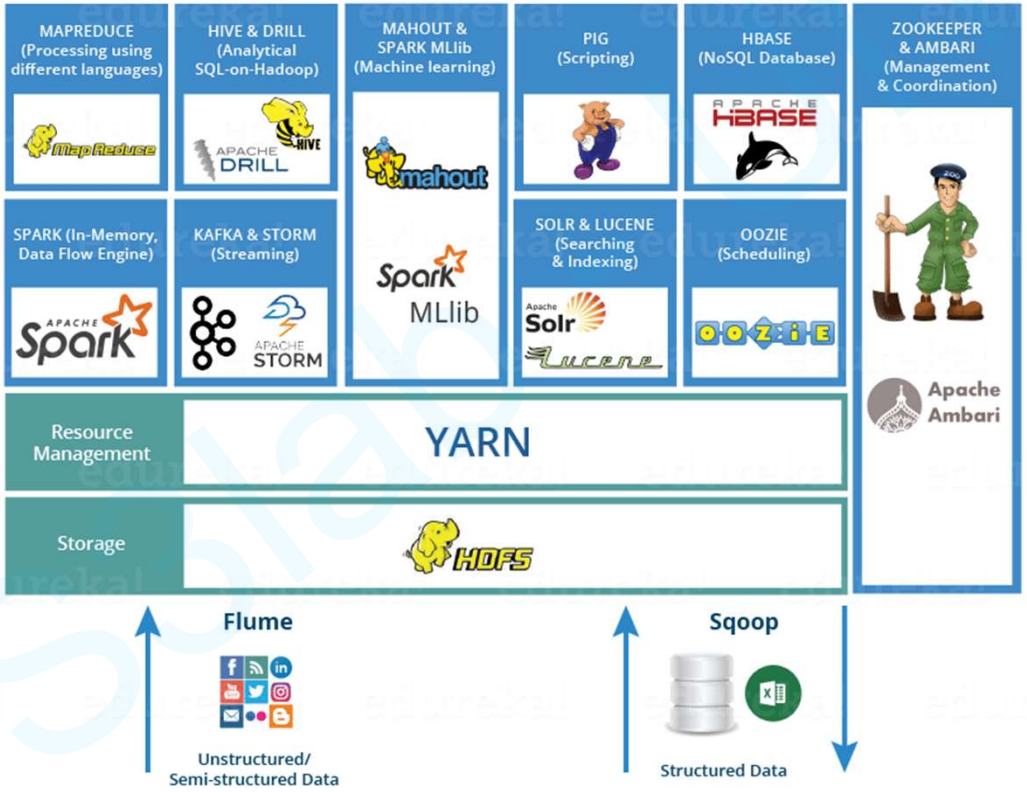
- Application Master
  - Per-application
  - Manages application scheduling and task execution
  - e.g. MapReduce Application Master

Yarn

Architecture



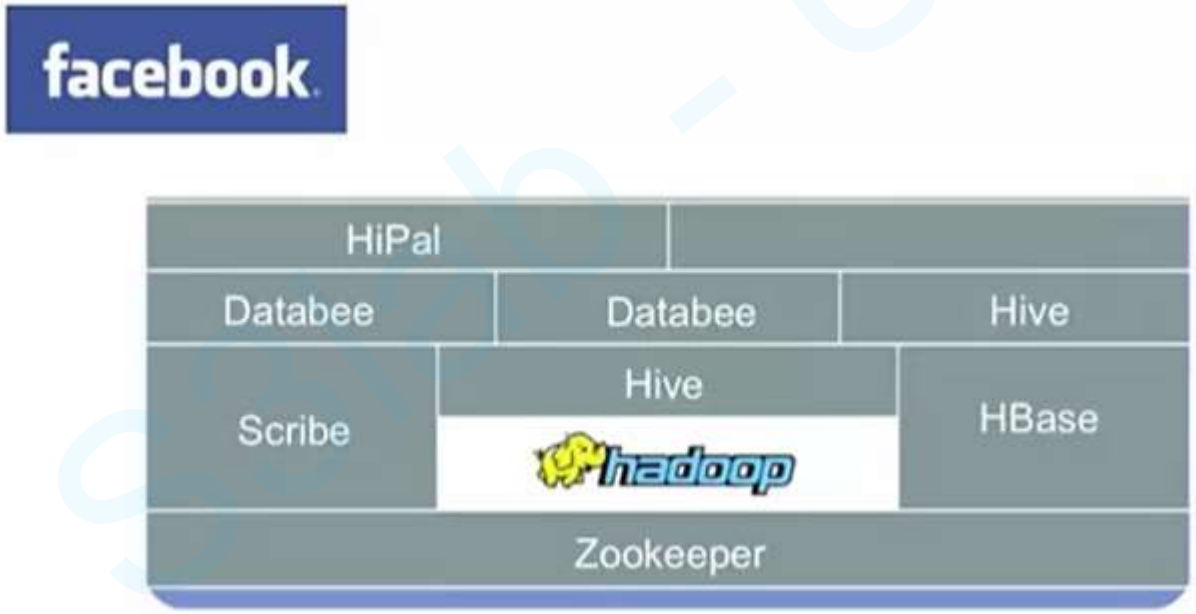
Ecosystem


MAPREDUCE (Processing using different languages)
HIVE & DRILL (Analytical SQL-on-Hadoop)
MAHOUT & SPARK MLlib (Machine learning)
PIG (Scripting)
HBASE (NoSQL Database)
ZOOKEEPER & AMBARI (Management & Coordination)
SPARK (In-Memory, Data Flow Engine)
KAFKA & STORM (Streaming)
SOLR & LUCENE (Searching & Indexing)
OOZIE (Scheduling)
Resource Management
YARN
Storage
HDFS
Flume
Unstructured/ Semi-structured Data
Sqoop
Structured Data

Hadoop Ecosystem History

The Google Stack

Hadoop Ecosystem History

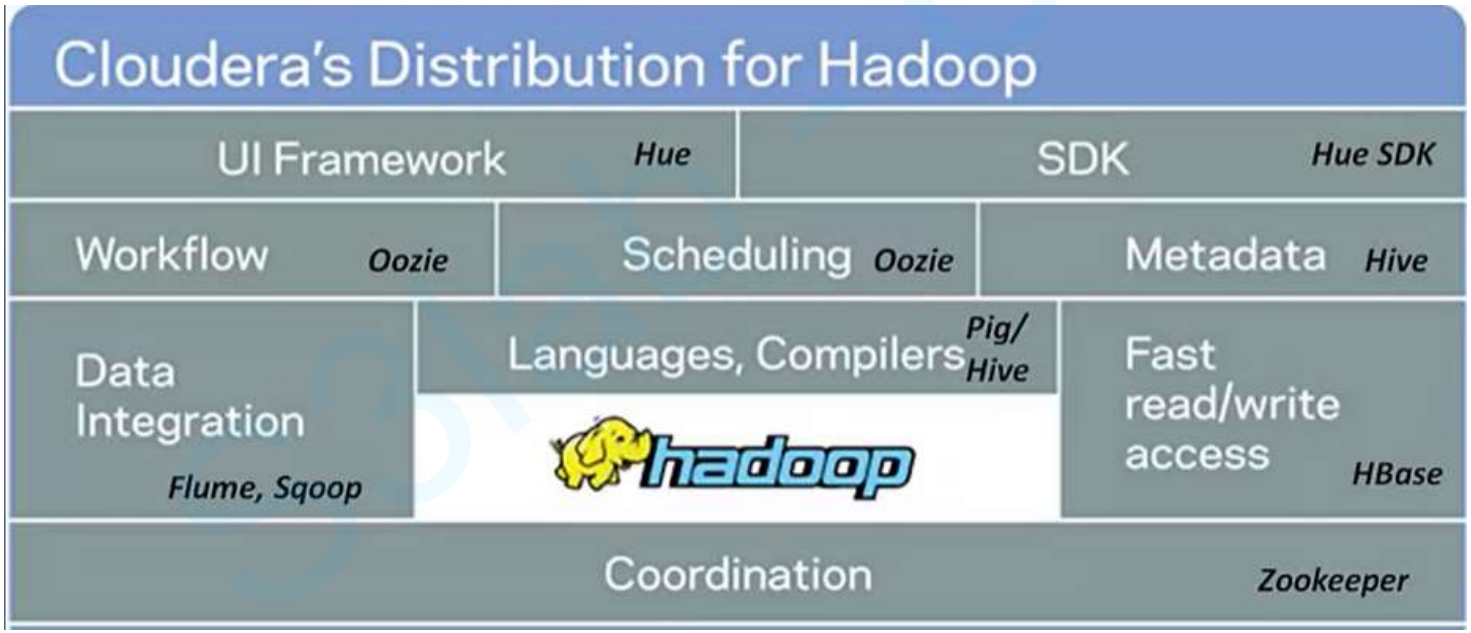


Hadoop Ecosystem history

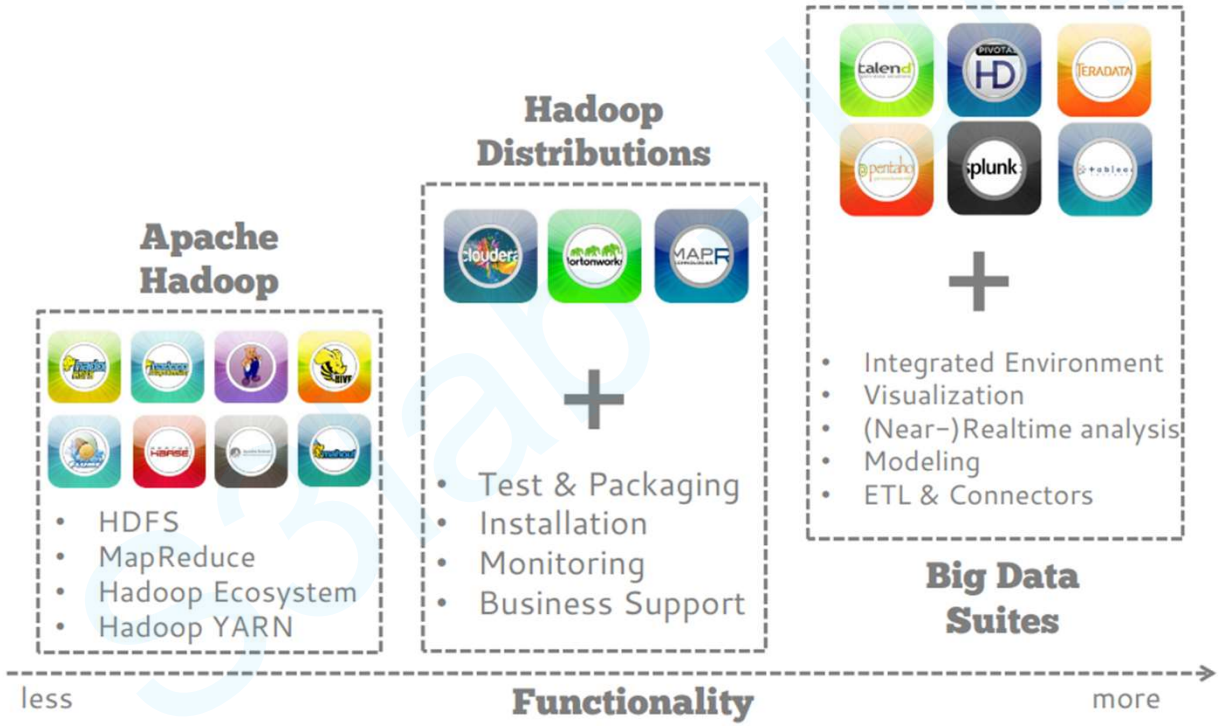
Hadoop Ecosystem History

Linked

Hadoop Ecosystem History



Ecosystem

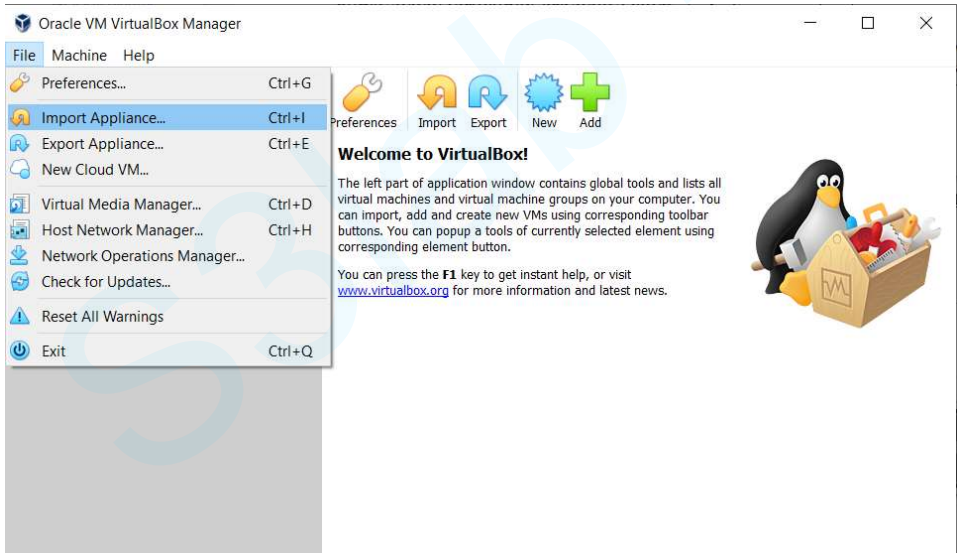

Apache Hadoop
HDFS
MapReduce
Hadoop Ecosystem
Hadoop YARN
Hadoop Distributions
Test & Packaging
Installation
Monitoring
Business Support
Integrated Environment
Visualization
(Near-)Realtime analysis
Modeling
ETL & Connectors
Big Data Suites
less
Functionality
more

Install Cloudera Quickstart VM

- Download Cloudera Quickstart VM for VirtualBox

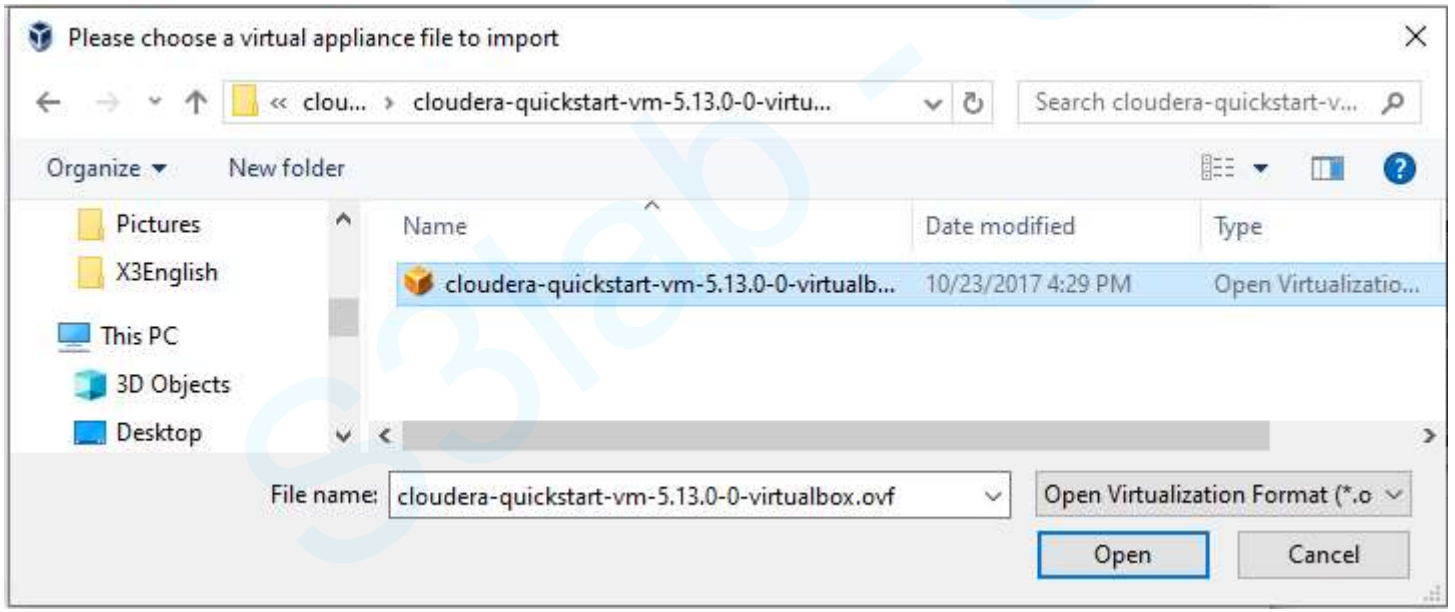
https://downloads.cloudera.com/demo_vm/virtualbox/cloudera-quickstart-vm-5.13.0-0-virtualbox.zip

- Open Oracle VM VirtualBox. Click File → Import Appliance

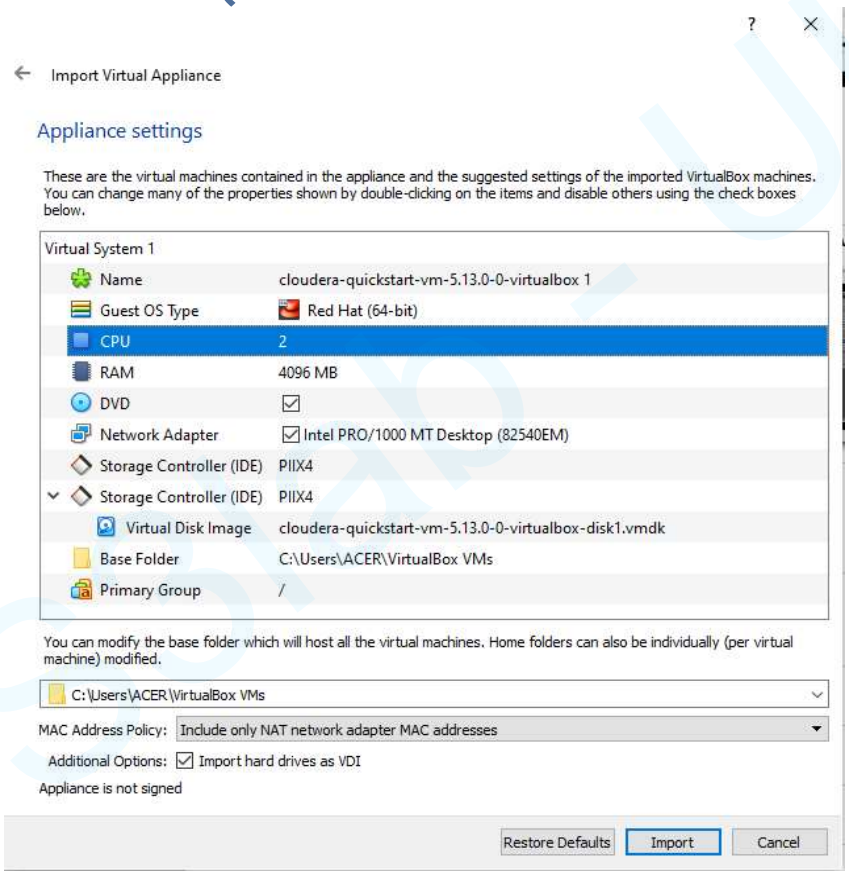


Install Cloudera Quickstart VM

Choose “cloudera-quickstart-vm-5.13.0-0-virtualbox.ovf”



Install Cloudera Quickstart VM

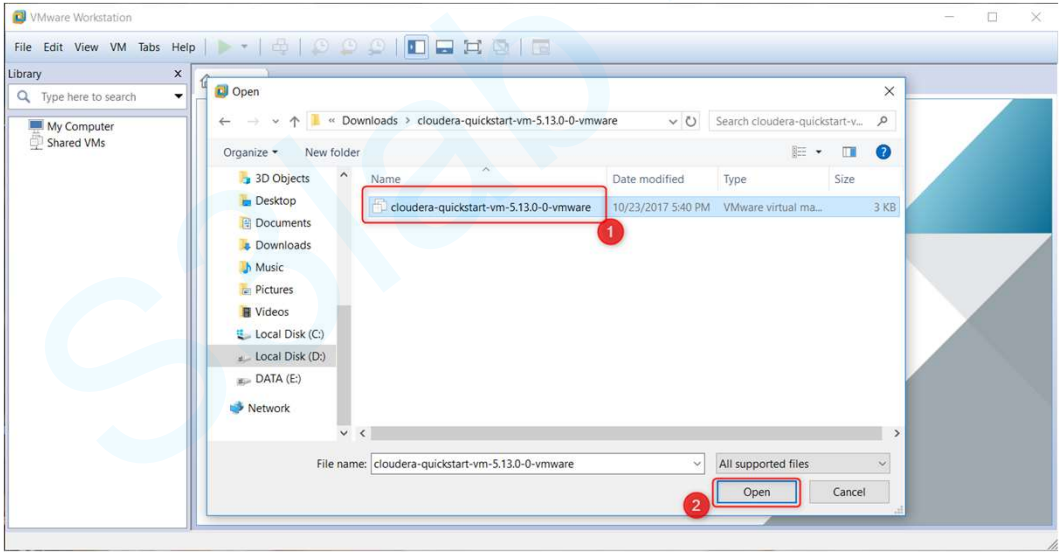


Install Cloudera Quickstart VM

VMware

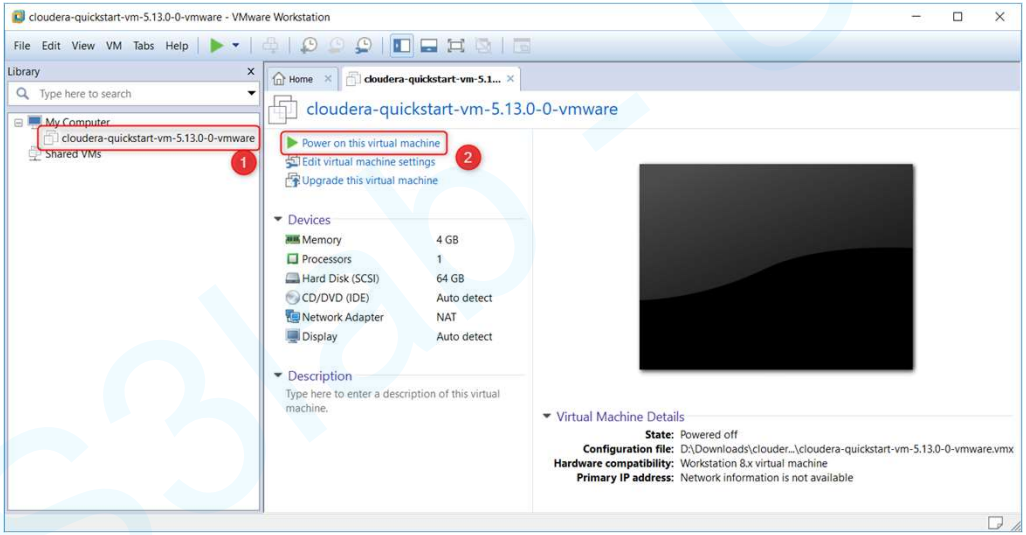
- Download Cloudera Quickstart VM for Vmware

https://downloads.cloudera.com/demo_vm/vmware/cloudera-quickstart-vm-5.13.0-0-vmware.zip



Install Cloudera Quickstart VM

VMware



Q & A

Cảm ơn đã theo dõi

Chúng tôi hy vọng cùng nhau đi đến thành công.