Hadoop Administration

Hadoop Administration::::::::

Introduction to BigData:

1. What is Big Data and how is it a problem?
2. Introduction to Hadoop
3. Introduction of Apache Hadoop and Hadoop Distributions, Apache Hadoop and Eco-system.
4. Introduction of HDFS and MapReduce
5. Hadoop cluster administrator: Roles and responsibilities

The Hadoop Distributed File System (HDFS):

1. Architecture of Hadoop Distributed File System
2. NameNode, Data Nodes.
3. High Availability of NameNode
4. Writing and Reading Files in HDFS
5. Replication rules
6. Rack awareness theory
7. Hadoop Basic Commands.

Introduction to MapReduce and YARN:

1. Introduction to MapReduce
2. MapReduce process flow
3. YARN – The Cluster Manager Architecture and Process flow
4. Exploring YARN Applications Through the Web UIs, and the Shell
5. Exploring YARN logs
6. Types of Schedulers – FIFO, FAIR and CAPACITY
7. Hands on Exercise

Introduction to Cloudera Distribution of Hadoop:

1. Introduction to Cloudera Manager Console.
2. Locating Configurations and Applying Configuration Changes
3. Managing Role Instances and Adding Services
4. Commission/De-commission nodes.
5. Troubleshooting errors checking Service Logs
6. Accessing HUE interface
7. Resource pool allocation.
8. Exploring Cloudera Manager UI

Installation of Cloudera Manager and CDH Cluster:

1. Cloudera Manager Installation

Introduction to Hive and Impala:

1. Architecture and components of Hive
2. Hive process flow and Logs
3. Hive troubleshooting.
4. Hands on Exercise

Introduction to Spark:

1. Spark Architecture
2. Spark components, process flow getting started with a Spark JOB.
3. Troubleshooting Spark Job
4. Hands on Exercise

Introduction to Hbase:

1. Introduction of Hbase and NoSql Databases
2. Architecture and process flow
3. Hbase vs HDFS

Introduction to Data Ingestion using Tools:

1. Sqoop introduction.
2. Ingesting Data using Sqoop from other RDBMS sources
3. Hands on Exercise

Introduction to JOB Scheduler:

1. Oozie and how it works in scheduling the JOBS.

Cluster Maintenance:

1. Checking HDFS Status
2. Copying Data between Clusters
3. Adding and Removing Cluster Nodes
4. Rebalancing the Cluster
5. Directory Snapshots
6. Cluster Upgrading

Introduction to Hadoop Security:

1. Need of Security (Authentication and Authorization) in Hadoop
2. ACL – how it works in Hadoop
3. Introduction to Kerberos – Components
4. Introduction to Sentry and how authorization works

Hadoop – Resume & Interview Preparation.

Big Data Tutorial

Big Data, haven't you heard this term before? I am sure you have. In the last 4 to 5 years, everyone is talking about Big Data. But do you really know what exactly is this Big Data and how is it making an impact on our lives? In this Big Data Tutorial, I will give you a complete insight about Big Data.

Below are the topics which I will cover in this Big Data Tutorial:

- Story of Big Data
- Big Data Driving Factors
- What is Big Data?
- Big Data Characteristics
- Types of Big Data
- Examples of Big Data
- Applications of Big Data
- Challenges with Big Data

Let me start this Big Data Tutorial with a short story.

Story of Big Data

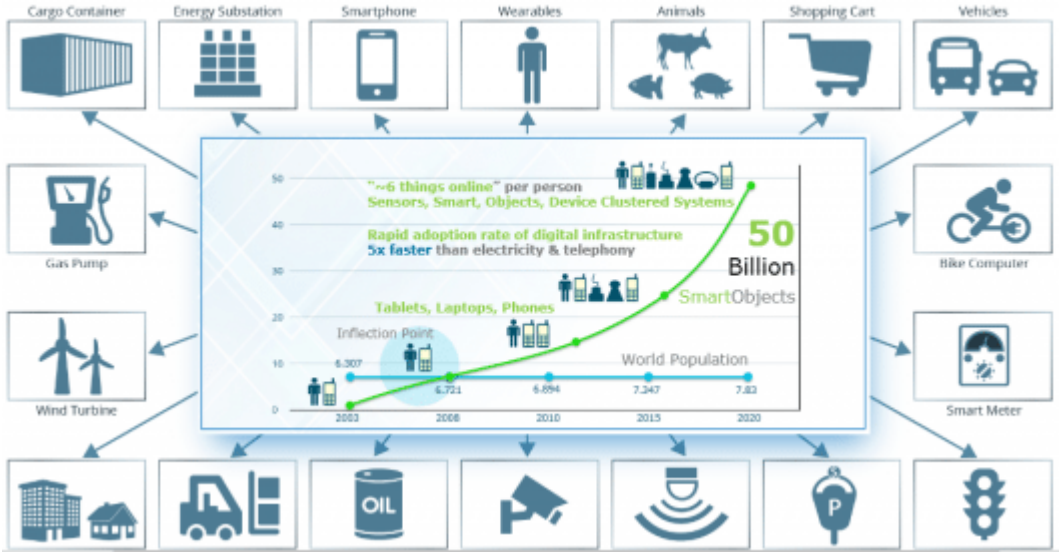
In ancient days, people used to travel from one village to another village on a horse driven cart, but as the time passed, villages became towns and people spread out. The distance to travel from one town to the other town also increased. So, it became a problem to travel between towns, along with the luggage. Out of the blue, one smart fella suggested, we should groom and feed a horse more, to solve this problem. When I look at this solution, it is not that bad, but do you think a horse can become an elephant? I don't think so. Another smart guy said, instead of 1

horse pulling the cart, let us have 4 horses to pull the same cart. What do you guys think of this solution? I think it is a fantastic solution. Now, people can travel large distances in less time and even carry more luggage.

The same concept applies on Big Data. Big Data says, till today, we were okay with storing the data into our servers because the volume of the data was pretty limited, and the amount of time to process this data was also okay. But now in this current technological world, the data is growing too fast and people are relying on the data a lot of times. Also the speed at which the data is growing, it is becoming impossible to store the data into any server.

Through this blog on Big Data Tutorial, let us explore the sources of Big Data, which the traditional systems are failing to store and process.

Big Data Driving Factors



The quantity of data on planet earth is growing exponentially for many reasons. Various sources and our day to day activities generates lots of data. With the invent of the web, the whole world has gone online, every single thing we do leaves a digital trace. With the smart objects going online, the data growth rate has increased rapidly. The major sources of Big Data are social media sites, sensor networks, digital images/videos, cell phones, purchase transaction records, web logs, medical records, archives, military surveillance, eCommerce, complex scientific research and so on. All these information amounts to around some Quintillion bytes

of data. By 2020, the data volumes will be around 40 Zettabytes which is equivalent to adding every single grain of sand on the planet multiplied by seventy-five.

What is Big Data?

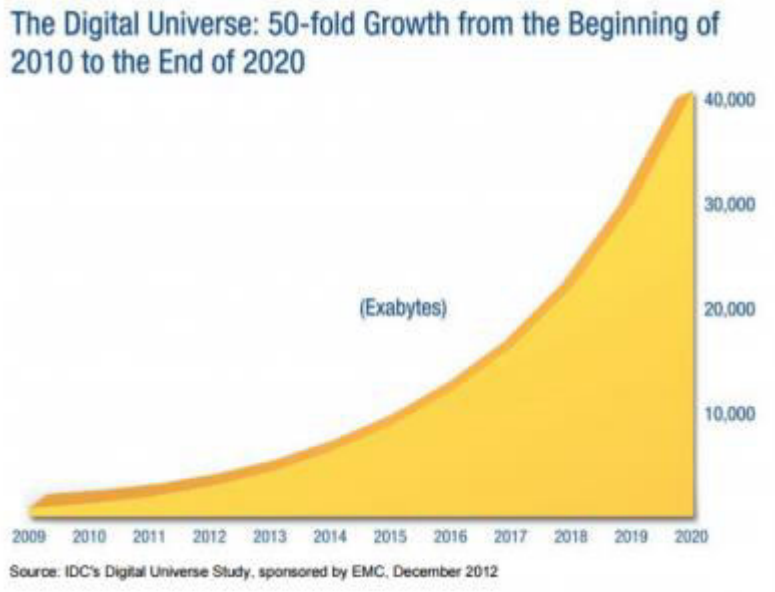
Big Data is a term used for a collection of data sets that are large and complex, which is difficult to store and process using available database management tools or traditional data processing applications. The challenge includes capturing, curating, storing, searching, sharing, transferring, analyzing and visualization of this data.

Big Data Characteristics

The five characteristics that define Big Data are: Volume, Velocity, Variety, Veracity and Value.

1. *VOLUME*

   Volume refers to the 'amount of data', which is growing day by day at a very fast pace. The size of data generated by humans, machines and their interactions on social media itself is massive. Researchers have predicted that 40 Zettabytes (40,000 Exabytes) will be generated by 2020, which is an increase of 300 times from 2005.



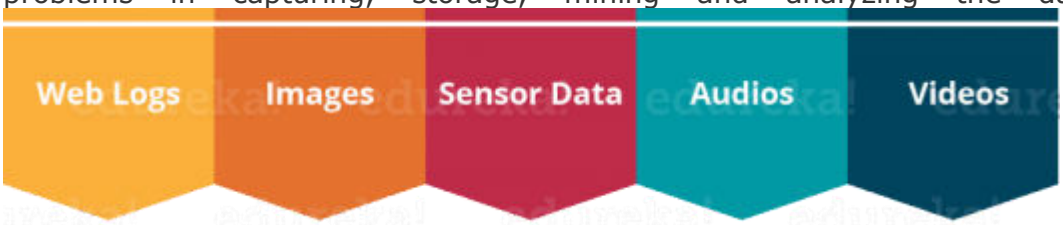
2. *VELOCITY*

   Velocity is defined as the pace at which different sources generate the data every day. This flow of data is massive and continuous. There are 1.03 billion Daily Active Users (Facebook DAU) on Mobile as of now, which is an increase of 22% year-over-year. This shows how fast the number of users are growing on social media and how fast the data is getting generated daily. If you are able to handle the velocity, you will be able to generate insights and take

decisions based on real-time data.

3. *VARIETY*

As there are many sources which are contributing to Big Data, the type of data they are generating is different. It can be structured, semi-structured or unstructured. Hence, there is a variety of data which is getting generated every day. Earlier, we used to get the data from excel and databases, now the data are coming in the form of images, audios, videos, sensor data etc. as shown in below image. Hence, this variety of unstructured data creates problems in capturing, storage, mining and analyzing the data.



4. *VERACITY*

Veracity refers to the data in doubt or uncertainty of data available due to data inconsistency and incompleteness. In the image below, you can see that few values are missing in the table. Also, a few values are hard to accept, for example – 15000 minimum value in the 3rd row, it is not possible. This inconsistency and incompleteness is Veracity.

Min	Max	Mean	SD
4.3	?	5.84	0.83
2.0	4.4	3.05	50000000
15000	7.9	1.20	0.43
0.1	2.5	?	0.76

Data available can sometimes get messy and maybe difficult to trust. With many forms of big data, quality and accuracy are difficult to control like Twitter posts with hashtags, abbreviations, typos and colloquial speech. The volume is often the reason behind for the lack of quality and accuracy in the data.

- Due to uncertainty of data, 1 in 3 business leaders don't trust the information they use to make decisions.
- It was found in a survey that 27% of respondents were unsure of how much of their data was inaccurate.
- Poor data quality costs the US economy around $3.1 trillion a year.

5. *VALUE*

   After discussing Volume, Velocity, Variety and Veracity, there is another V that should be taken into account when looking at Big Data i.e. Value. It is all well and good to have access to big data but unless we can turn it into value it is useless. By turning it into value I mean, Is it adding to the benefits of the organizations who are analyzing big data? Is the organization working on Big Data achieving high ROI (Return On Investment)? Unless, it adds to their profits by working on Big Data, it is useless.

Go through our Big Data video below to know more about Big Data:

Big Data Tutorial For Beginners | What Is Big Data | Edureka

As discussed in Variety, there are different types of data which is getting generated every day. So, let us now understand the types of data:

Types of Big Data

Big Data could be of three types:

- Structured
- Semi-Structured
- Unstructured



1. *Structured*

   The data that can be stored and processed in a fixed format is called as Structured Data. Data stored in a relational database management system (RDBMS) is one example of 'structured' data. It is easy to process structured data as it has a fixed schema. Structured Query Language (SQL) is often used to manage such kind of Data.

2. *Semi-Structured*

   Semi-Structured Data is a type of data which does not have a formal structure of a data model, i.e. a table definition in a relational DBMS, but nevertheless it has some organizational properties like tags and other markers to separate semantic elements that makes it easier to analyze. XML files or JSON documents are examples of semi-structured data.

3. *Unstructured*

   The data which have unknown form and cannot be stored in RDBMS and cannot be analyzed unless it is transformed into a structured format is called as unstructured data. Text Files and multimedia contents like images, audios, videos are example of unstructured data. The unstructured data is growing quicker than others, experts say that 80 percent of the data in an organization are unstructured.

Till now, I have just covered the introduction of Big Data. Furthermore, this Big Data tutorial talks about examples, applications and challenges in Big Data.

Examples of Big Data

Daily we upload millions of bytes of data. 90 % of the world's data has been created in last two years.

- Walmart handles more than **1 million** customer transactions every hour.
- Facebook stores, accesses, and analyzes **30+ Petabytes** of user generated data.
- **230+ millions** of tweets are created every day.
- More than **5 billion** people are calling, texting, tweeting and browsing on mobile phones worldwide.
- YouTube users upload **48 hours** of new video every minute of the day.
- Amazon handles **15 million** customer click stream user data per day to recommend products.

- **294 billion** emails are sent every day. Services analyses this data to find the spams.
- Modern cars have close to **100 sensors** which monitors fuel level, tire pressure etc. , each vehicle generates a lot of sensor data.

Applications of Big Data

We cannot talk about data without talking about the people, people who are getting benefited by Big Data applications. Almost all the industries today are leveraging Big Data applications in one or the other way.

- **Smarter Healthcare**: Making use of the petabytes of patient's data, the organization can extract meaningful information and then build applications that can predict the patient's deteriorating condition in advance.

- **Telecom**: Telecom sectors collects information, analyzes it and provide solutions to different problems. By using Big Data applications, telecom companies have been able to significantly reduce data packet loss, which occurs when networks are overloaded, and thus, providing a seamless connection to their customers.

- **Retail**: Retail has some of the tightest margins, and is one of the greatest beneficiaries of big data. The beauty of using big data in retail is to understand consumer behavior. Amazon's recommendation engine provides suggestion based on the browsing history of the consumer.

- **Traffic control**: Traffic congestion is a major challenge for many cities globally. Effective use of data and sensors will be key to managing traffic better as cities become increasingly densely populated.

- **Manufacturing**: Analyzing big data in the manufacturing industry can reduce component defects, improve product quality, increase efficiency, and save time and money.

- **Search Quality**: Every time we are extracting information from google, we are simultaneously generating data for it. Google stores this data and uses it to improve its search quality.

Someone has rightly said: ***"Not everything in the garden is Rosy!"***. Till now in this Big Data tutorial, I have just shown you the rosy picture of Big Data. But if it was so easy to leverage Big data, don't you think all the organizations would invest in it? Let me tell you upfront, that is not the case. There are several challenges which come along when you are working with Big Data.

What is Hadoop?

Hadoop is an open-source software framework used for storing and processing Big Data in a distributed manner on large clusters of commodity hardware. Hadoop is licensed under the Apache v2 license. Hadoop was developed, based on the paper written by Google on MapReduce system and it applies concepts of functional programming. Hadoop is written in the Java programming language and ranks among the highest-level Apache projects. Hadoop was developed by Doug Cutting and Michael J. Cafarella.

Hadoop-as-a-Solution

Let's understand how Hadoop provides solution to the Big Data problems that we have discussed so far.

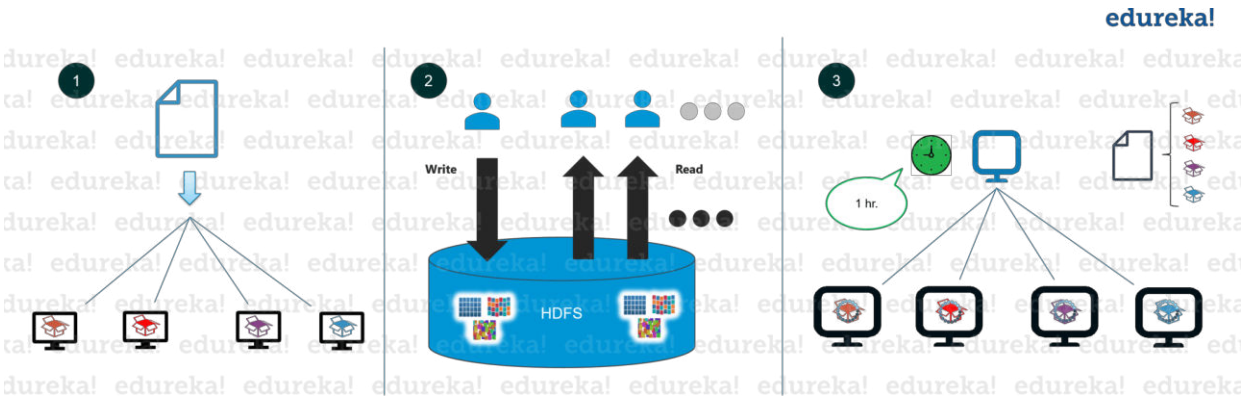


Fig: Hadoop Tutorial – Hadoop-as-a-Solution

The first problem is storing huge amount of data.

As you can see in the above image, HDFS provides a distributed way to store Big Data. Your data is stored in blocks in DataNodes and you specify the size of each block. Suppose you have 512MB of data and you have configured HDFS such that it

will create 128 MB of data blocks. Now, HDFS will divide data into 4 blocks as 512/128=4 and stores it across different DataNodes. While storing these data blocks into DataNodes, data blocks are replicated on different DataNodes to provide fault tolerance.

Hadoop follows ***horizontal scaling*** instead of vertical scaling. In horizontal scaling, you can add new nodes to HDFS cluster on the run as per requirement, instead of increasing the hardware stack present in each node.

*Next problem was storing the variety of data***.**

As you can see in the above image, in HDFS you can store all kinds of data whether it is structured, semi-structured or unstructured. In HDFS, there is *no pre-dumping schema validation.* It also follows write once and read many model. Due to this, you can just write any kind of data once and you can read it multiple times for finding insights.

*The third challenge was about processing the data faster***.**

In order to solve this, we move processing unit to data instead of moving data to processing unit. So, what does it mean by moving the computation unit to data? It means that instead of moving data from different nodes to a single master node for processing, the processing logic is sent to the nodes where data is stored so as that each node can process a part of data in parallel. Finally, all of the intermediary output produced by each node is merged together and the final response is sent back to the clien

Hadoop Features

Fig: Hadoop Tutorial – Hadoop Features

Reliability:

When machines are working in tandem, if one of the machines fails, another machine will take over the responsibility and work in a reliable and fault tolerant fashion. Hadoop infrastructure has inbuilt fault tolerance features and hence, Hadoop is highly reliable.

Economical:

Hadoop uses commodity hardware (like your PC, laptop). For example, in a small Hadoop cluster, all your DataNodes can have normal configurations like 8-16 GB RAM with 5-10 TB hard disk and Xeon processors, but if I would have used hardware-based RAID with Oracle for the same purpose, I would end up spending 5x times more at least. So, the cost of ownership of a Hadoop-based project is pretty minimized. It is easier to maintain the Hadoop environment and is economical as well. Also, Hadoop is an open source software and hence there is no licensing cost.

Scalability:

Hadoop has the inbuilt capability of integrating seamlessly with cloud-based services. So, if you are installing Hadoop on a cloud, you don't need to worry about the scalability factor because you can go ahead and procure more hardware and expand your setup within minutes whenever required.

Flexibility:

Hadoop is very flexible in terms of ability to deal with all kinds of data. We discussed "Variety" in our previous blog on ***Big Data Tutorial***, where data can be of any kind and Hadoop can store and process them all, whether it is structured, semi-structured or unstructured data.

These 4 characteristics make Hadoop a front-runner as a solution to Big Data challenges. Now that we know what is Hadoop, we can explore the core components of Hadoop. Let us understand, what are the core components of Hadoop.

Hadoop Core Components

While setting up a Hadoop cluster, you have an option of choosing a lot of services as part of your Hadoop platform, but there are two services which are always mandatory for setting up Hadoop. One is **HDFS (storage)** and the other is **YARN (processing)**. HDFS stands for **Hadoop Distributed File System**, which is a scalable storage unit of Hadoop whereas YARN is used to process the data i.e. stored in the HDFS in a distributed and parallel fashion.

HDFS

Let us go ahead with HDFS first. The main components of **HDFS** are: **NameNode** and **DataNode**. Let us talk about the roles of these two components in detail.

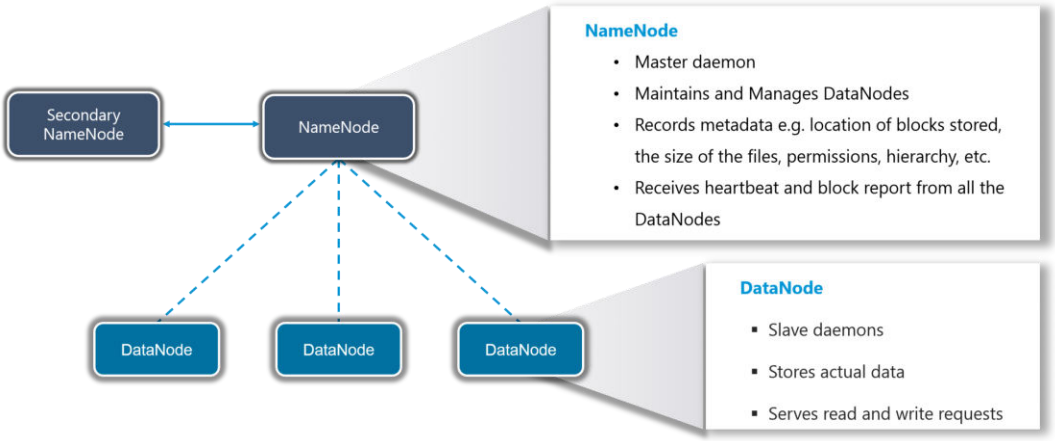


Fig: Hadoop Tutorial – HDFS

NameNode

- It is the master daemon that maintains and manages the DataNodes (slave nodes)

- It records the metadata of all the blocks stored in the cluster, e.g. location of blocks stored, size of the files, permissions, hierarchy, etc.
- It records each and every change that takes place to the file system metadata
- If a file is deleted in HDFS, the NameNode will immediately record this in the EditLog
- It regularly receives a Heartbeat and a block report from all the DataNodes in the cluster to ensure that the DataNodes are live
- It keeps a record of all the blocks in the HDFS and DataNode in which they are stored
- It has high availability and federation features which I will discuss in ***HDFS architecture*** in detail

DataNode

- It is the slave daemon which run on each slave machine
- The actual data is stored on DataNodes
- It is responsible for serving read and write requests from the clients
- It is also responsible for creating blocks, deleting blocks and replicating the same based on the decisions taken by the NameNode
- It sends heartbeats to the NameNode periodically to report the overall health of HDFS, by default, this frequency is set to 3 seconds

So, this was all about HDFS in nutshell. Now, let move ahead to our second fundamental unit of Hadoop i.e. YARN.

YARN

YARN comprises of two major component: **ResourceManager** and **NodeManager**.

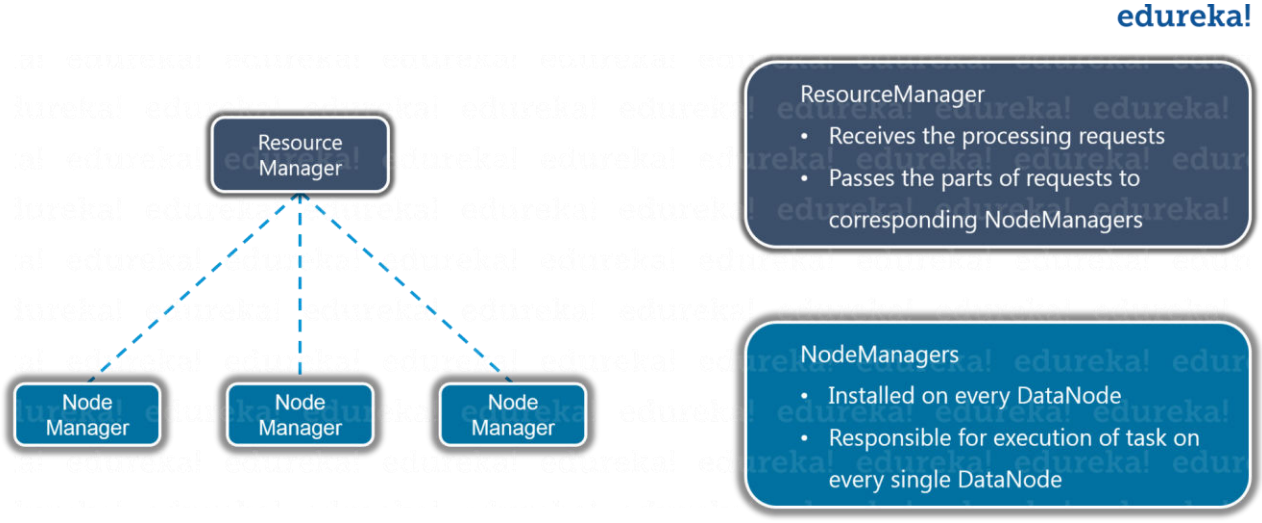


Fig: Hadoop Tutorial – YARN

ResourceManager

- It is a cluster level (one for each cluster) component and runs on the master machine
- It manages resources and schedule applications running on top of YARN

- It has two components: Scheduler & ApplicationManager
- The Scheduler is responsible for allocating resources to the various running applications
- The ApplicationManager is responsible for accepting job submissions and negotiating the first container for executing the application
- It keeps a track of the heartbeats from the Node Manager

NodeManager

- It is a node level component (one on each node) and runs on each slave machine
- It is responsible for managing containers and monitoring resource utilization in each container
- It also keeps track of node health and log management
- It continuously communicates with ResourceManager to remain up-to-date

Hadoop Ecosystem

So far you would have figured out that Hadoop is neither a programming language nor a service, it is a platform or framework which solves Big Data problems. You can consider it as a suite which encompasses a number of services for ingesting, storing and analyzing huge data sets along with tools for configuration management.

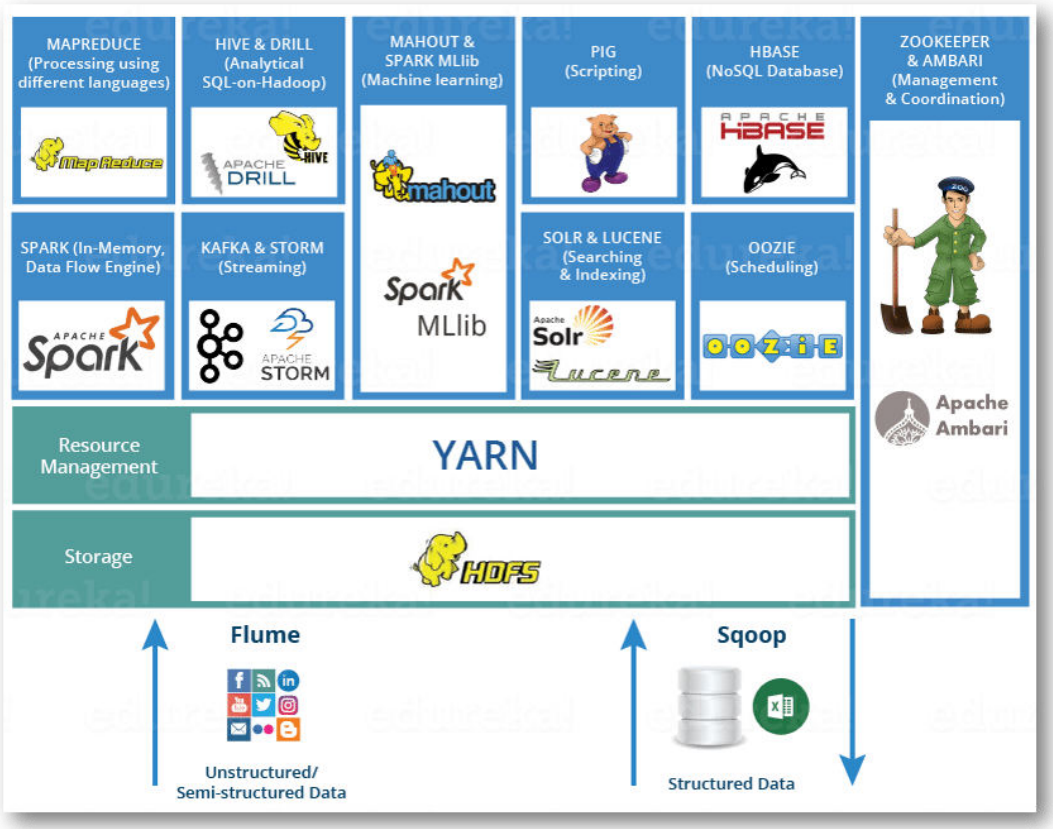


Fig: Hadoop Tutorial – Hadoop Ecosystem

We have discussed Hadoop Ecosystem and their components in detail in our ***Hadoop Ecosystem blog***. Now in this Hadoop Tutorial, let us know how **Last.fm used Hadoop as a part of their solution strategy**.

Last.fm Case Study

Last.fm is internet radio and community-driven music discovery service founded in 2002. Users transmit information to Last.fm servers indicating which songs they are listening to. The received data is processed and stored so that, the user can access it in the form of charts. Thus, Last.fm can make intelligent taste and compatible decisions for generating recommendations. The data is obtained from one of the two sources stated below:

- **scrobble:** When a user plays a track of his or her own choice and sends the information to Last.fm through a client application.
- **radio listen:** When the user tunes into a Last.fm radio station and streams a song.

Last.fm applications allow users to love, skip or ban each track they listen to. This track listening data is also transmitted to the server.

- Over 40M unique visitors and 500M page views each month
- Scrobble stats:
    - Up to 800 scrobbles per second
    - More than 40 million scrobbles per day
    - Over 75 billion scrobbles so far
- Radio stats:
    - Over 10 million streaming hours per month
    - Over 400 thousand unique stations per day
- Each scrobble and radio listen generates at least one log line

Hadoop at Last.FM

- 100 Nodes
- 8 cores per node (dual quad-core)
- 24GB memory per node
- 8TB (4 disks of 2TB each)
- Hive integration to run optimized SQL queries for analysis

Last.FM started using Hadoop in 2006 because of the growth in users from thousands to millions. With the help of Hadoop they processed hundreds of daily, monthly, and weekly jobs including website stats and metrics, chart generation (i.e. track statistics), metadata corrections (e.g. misspellings of artists), indexing for search, combining/formatting data for recommendations, data insights, evaluations

& reporting. This helped Last.FM to grow tremendously and figure out the taste of their users, based on which they started recommending musics.

I hope this blog was informative and added value to your knowledge. In our next blog on ***Hadoop Ecosystem***, we will discuss different tools present in Hadoop Ecosystem in detail.

What is Big Data?

Big data means really a big data, it is a collection of large datasets that cannot be processed using traditional computing techniques. Big data is not merely a data, rather it has become a complete subject, which involves various tools, technqiues and frameworks.

What Comes Under Big Data?

Big data involves the data produced by different devices and applications. Given below are some of the fields that come under the umbrella of Big Data.

- **Black Box Data** : It is a component of helicopter, airplanes, and jets, etc. It captures voices of the flight crew, recordings of microphones and earphones, and the performance information of the aircraft.
- **Social Media Data** : Social media such as Facebook and Twitter hold information and the views posted by millions of people across the globe.
- **Stock Exchange Data** : The stock exchange data holds information about the 'buy' and 'sell' decisions made on a share of different companies made by the customers.
- **Power Grid Data** : The power grid data holds information consumed by a particular node with respect to a base station.
- **Transport Data** : Transport data includes model, capacity, distance and availability of a vehicle.
- **Search Engine Data** : Search engines retrieve lots of data from different databases.

Thus Big Data includes huge volume, high velocity, and extensible variety of data. The data in it will be of three types.

- **Structured data** : Relational data.
- **Semi Structured data** : XML data.
- **Unstructured data** : Word, PDF, Text, Media Logs.

Benefits of Big Data

Big data is really critical to our life and its emerging as one of the most important technologies in modern world. Follow are just few benefits which are very much known to all of us:

- Using the information kept in the social network like Facebook, the marketing agencies are learning about the response for their campaigns, promotions, and other advertising mediums.
- Using the information in the social media like preferences and product perception of their consumers, product companies and retail organizations are planning their production.
- Using the data regarding the previous medical history of patients, hospitals are providing better and quick service.

Big Data Technologies

Big data technologies are important in providing more accurate analysis, which may lead to more concrete decision-making resulting in greater operational efficiencies, cost reductions, and reduced risks for the business.

To harness the power of big data, you would require an infrastructure that can manage and process huge volumes of structured and unstructured data in realtime and can protect data privacy and security.

There are various technologies in the market from different vendors including Amazon, IBM, Microsoft, etc., to handle big data. While looking into the technologies that handle big data, we examine the following two classes of technology:

Operational Big Data

This include systems like MongoDB that provide operational capabilities for real-time, interactive workloads where data is primarily captured and stored.

NoSQL Big Data systems are designed to take advantage of new cloud computing architectures that have emerged over the past decade to allow massive computations to be run inexpensively and efficiently. This makes operational big data workloads much easier to manage, cheaper, and faster to implement.

Some NoSQL systems can provide insights into patterns and trends based on real-time data with minimal coding and without the need for data scientists and additional infrastructure.

Analytical Big Data
This includes systems like Massively Parallel Processing (MPP) database systems and MapReduce that provide analytical capabilities for retrospective and complex analysis that may touch most or all of the data.

MapReduce provides a new method of analyzing data that is complementary to the capabilities provided by SQL, and a system based on MapReduce that can be scaled up from single servers to thousands of high and low end machines.

These two classes of technology are complementary and frequently deployed together.

Operational vs. Analytical Systems

	Operational	**Analytical**
Latency	1 ms - 100 ms	1 min - 100 min
Concurrency	1000 - 100,000	1 - 10
Access Pattern	Writes and Reads	Reads
Queries	Selective	Unselective
Data Scope	Operational	Retrospective
End User	Customer	Data Scientist
Technology	NoSQL	MapReduce, MPP Database

Big Data Challenges

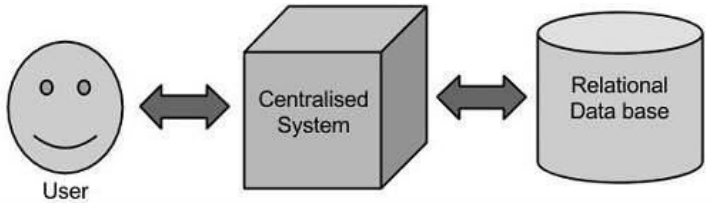
The major challenges associated with big data are as follows:

- Capturing data
- Curation
- Storage
- Searching
- Sharing
- Transfer
- Analysis
- Presentation

Hadoop - Big Data Solutions

Traditional Approach

In this approach, an enterprise will have a computer to store and process big data. Here data will be stored in an RDBMS like Oracle Database, MS SQL Server or DB2 and sophisticated softwares can be written to interact with the database, process the required data and present it to the users for analysis purpose.

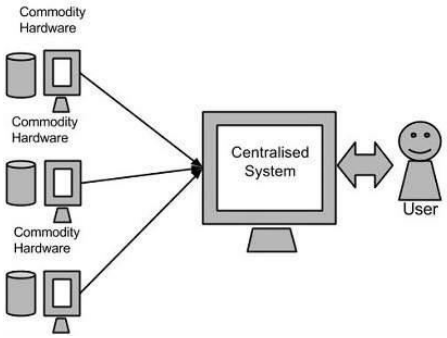


Limitation

This approach works well where we have less volume of data that can be accommodated by standard database servers, or up to the limit of the processor which is processing the data. But when it comes to dealing with huge amounts of data, it is really a tedious task to process such data through a traditional database server.

Google's Solution

Google solved this problem using an algorithm called MapReduce. This algorithm divides the task into small parts and assigns those parts to many computers connected over the network, and collects the results to form the final result dataset.

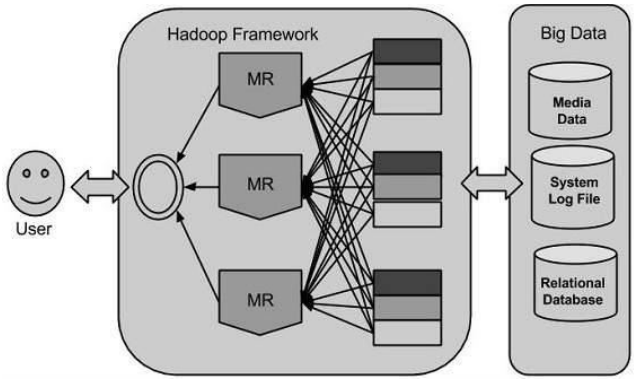


Above diagram shows various commodity hardwares which could be single CPU machines or servers with higher capacity.

Hadoop

Doug Cutting, Mike Cafarella and team took the solution provided by Google and started an Open Source Project called HADOOP in 2005 and Doug named it after his son's toy elephant. Now Apache Hadoop is a registered trademark of the Apache Software Foundation.

Hadoop runs applications using the MapReduce algorithm, where the data is processed in parallel on different CPU nodes. In short, Hadoop framework is capable enough to develop applications capable of running on clusters of computers and they could perform complete statistical analysis for a huge amounts of data.



Hadoop - Introduction

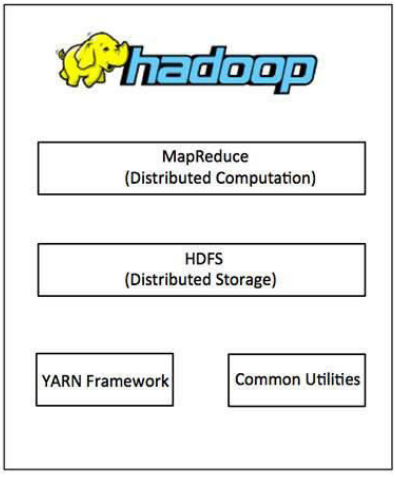
Hadoop is an Apache open source framework written in java that allows distributed processing of large datasets across clusters of computers using simple programming models. A Hadoop frame-worked application works in an environment that provides distributed storage and computation across clusters of computers. Hadoop is designed to scale up from single server to thousands of machines, each offering local computation and storage.

Hadoop Architecture

Hadoop framework includes following four modules:

- **Hadoop Common:** These are Java libraries and utilities required by other Hadoop modules. These libraries provides filesystem and OS level abstractions and contains the necessary Java files and scripts required to start Hadoop.
- **Hadoop YARN:** This is a framework for job scheduling and cluster resource management.
- **Hadoop Distributed File System (HDFS™):** A distributed file system that provides high-throughput access to application data.
- **Hadoop MapReduce:** This is YARN-based system for parallel processing of large data sets.

We can use following diagram to depict these four components available in Hadoop framework.



Since 2012, the term "Hadoop" often refers not just to the base modules mentioned above but also to the collection of additional software packages that can be installed on top of or alongside Hadoop, such as Apache Pig, Apache Hive, Apache HBase, Apache Spark etc.

MapReduce

Hadoop **MapReduce** is a software framework for easily writing applications which process big amounts of data in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

The term MapReduce actually refers to the following two different tasks that Hadoop programs perform:

- **The Map Task:** This is the first task, which takes input data and converts it into a set of data, where individual elements are broken down into tuples (key/value pairs).

- **The Reduce Task:** This task takes the output from a map task as input and combines those data tuples into a smaller set of tuples. The reduce task is always performed after the map task.

Typically both the input and the output are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

The MapReduce framework consists of a single master **JobTracker** and one slave **TaskTracker** per cluster-node. The master is responsible for resource management, tracking resource consumption/availability and scheduling the jobs component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves TaskTracker execute the tasks as directed by the master and provide task-status information to the master periodically.

The JobTracker is a single point of failure for the Hadoop MapReduce service which means if JobTracker goes down, all running jobs are halted.

Hadoop Distributed File System

Hadoop can work directly with any mountable distributed file system such as Local FS, HFTP FS, S3 FS, and others, but the most common file system used by Hadoop is the Hadoop Distributed File System (HDFS).

The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and provides a distributed file system that is designed to run on large clusters (thousands of computers) of small computer machines in a reliable, fault-tolerant manner.

HDFS uses a master/slave architecture where master consists of a single **NameNode** that manages the file system metadata and one or more slave **DataNodes** that store the actual data.

A file in an HDFS namespace is split into several blocks and those blocks are stored in a set of DataNodes. The NameNode determines the mapping of blocks to the DataNodes. The DataNodes takes care of read and write operation with the file system. They also take care of block creation, deletion and replication based on instruction given by NameNode.

HDFS provides a shell like any other file system and a list of commands are available to interact with the file system. These shell commands will be covered in a separate chapter along with appropriate examples.

How Does Hadoop Work?

Stage 1

A user/application can submit a job to the Hadoop (a hadoop job client) for required process by specifying the following items:

1. The location of the input and output files in the distributed file system.
2. The java classes in the form of jar file containing the implementation of map and reduce functions.
3. The job configuration by setting different parameters specific to the job.

Stage 2

The Hadoop job client then submits the job (jar/executable etc) and configuration to the JobTracker which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

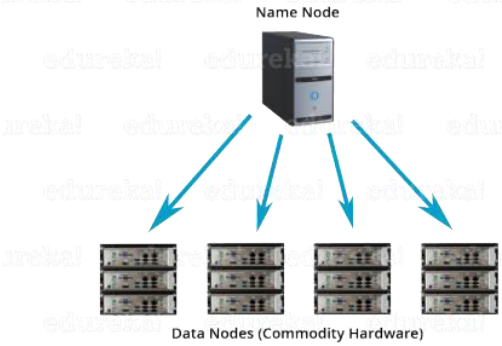
Stage 3

The TaskTrackers on different nodes execute the task as per MapReduce implementation and output of the reduce function is stored into the output files on the file system.

Advantages of Hadoop

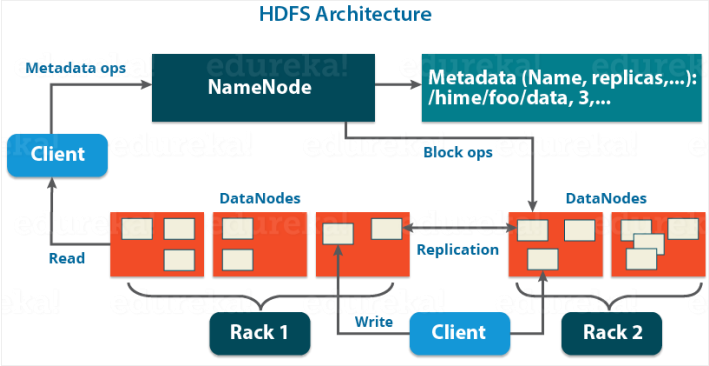
- Hadoop framework allows the user to quickly write and test distributed systems. It is efficient, and it automatic distributes the data and work across the machines and in turn, utilizes the underlying parallelism of the CPU cores.
- Hadoop does not rely on hardware to provide fault-tolerance and high availability (FTHA), rather Hadoop library itself has been designed to detect and handle failures at the application layer.
- Servers can be added or removed from the cluster dynamically and Hadoop continues to operate without interruption.
- Another big advantage of Hadoop is that apart from being open source, it is compatible on all the platforms since it is Java based.

HDFS Architecture:



Apache HDFS or **Hadoop Distributed File System** is a block-structured file system where each file is divided into blocks of a pre-determined size. These blocks are stored across a cluster of one or several machines. Apache Hadoop HDFS Architecture follows a *Master/Slave Architecture*, where a cluster comprises of a single NameNode (Master node) and all the other nodes are DataNodes (Slave nodes). HDFS can be deployed on a broad spectrum of machines that support Java. Though one can run several DataNodes on a single machine, but in the practical world, these DataNodes are spread across various machines.

NameNode:



NameNode is the master node in the Apache Hadoop HDFS Architecture that maintains and manages the blocks present on the DataNodes (slave nodes). NameNode is a very highly available server that manages the File System Namespace and controls access to files by clients. I will be discussing this High Availability feature of Apache Hadoop HDFS in my next blog. The HDFS architecture is built in such a way that the user data never resides on the NameNode. The data resides on DataNodes only.

Functions of NameNode:

- It is the master daemon that maintains and manages the DataNodes (slave nodes)
- It records the metadata of all the files stored in the cluster, e.g. The location of blocks stored, the size of the files, permissions, hierarchy, etc. There are two files associated with the metadata:
    - **FsImage:** It contains the complete state of the file system namespace since the start of the NameNode.
    - **EditLogs:** It contains all the recent modifications made to the file system with respect to the most recent FsImage.
- It records each change that takes place to the file system metadata. For example, if a file is deleted in HDFS, the NameNode will immediately record this in the EditLog.
- It regularly receives a Heartbeat and a block report from all the DataNodes in the cluster to ensure that the DataNodes are live.
- It keeps a record of all the blocks in HDFS and in which nodes these blocks are located.
- The NameNode is also responsible to take care of the **replication factor** of all the blocks which we will discuss in detail later in this HDFS tutorial blog.
- In **case of the DataNode failure**, the NameNode chooses new DataNodes for new replicas, balance disk usage and manages the communication traffic to the DataNodes.

DataNode:

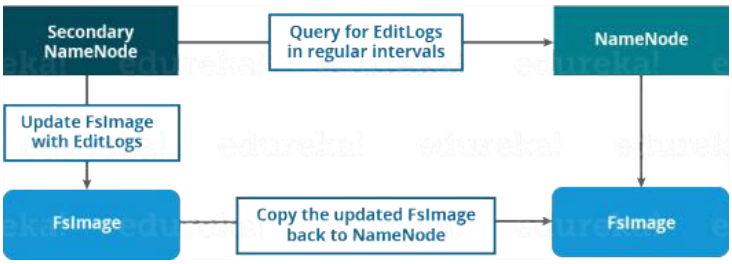
DataNodes are the slave nodes in HDFS. Unlike NameNode, DataNode is a commodity hardware, that is, a non-expensive system which is not of high quality or high-availability. The DataNode is a block server that stores the data in the local file ext3 or ext4.

Functions of DataNode:

- These are slave daemons or process which runs on each slave machine.
- The actual data is stored on DataNodes.
- The DataNodes perform the low-level read and write requests from the file system's clients.
- They send heartbeats to the NameNode periodically to report the overall health of HDFS, by default, this frequency is set to 3 seconds.

Till now, you must have realized that the NameNode is pretty much important to us. If it fails, we are doomed. But don't worry, we will be talking about how Hadoop solved this single point of failure problem in the next Apache Hadoop HDFS Architecture blog. So, just relax for now and let's take one step at a time.

Secondary NameNode:
Apart from these two daemons, there is a third daemon or a process called Secondary NameNode. The Secondary NameNode works concurrently with the primary NameNode as a **helper daemon.** And don't be confused about the Secondary NameNode being a **backup NameNode because it is not.**



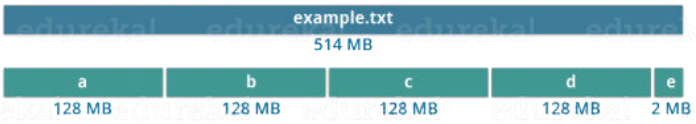
Functions of Secondary NameNode:

- The Secondary NameNode is one which constantly reads all the file systems and metadata from the RAM of the NameNode and writes it into the hard disk or the file system.
- It is responsible for combining the EditLogs with FsImage from the NameNode.
- It downloads the EditLogs from the NameNode at regular intervals and applies to FsImage. The new FsImage is copied back to the NameNode, which is used whenever the NameNode is started the next time.

Hence, Secondary NameNode performs regular checkpoints in HDFS. Therefore, it is also called CheckpointNode.

Blocks:
Now, as we know that the data in HDFS is scattered across the DataNodes as blocks. **Let's have a look at what is a block and how is it formed?**

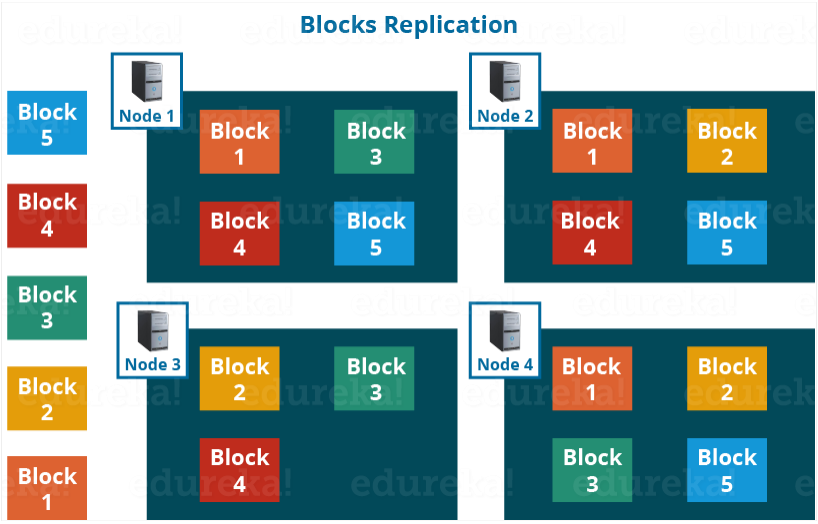
Blocks are the nothing but the smallest continuous location on your hard drive where data is stored. In general, in any of the File System, you store the data as a collection of blocks. Similarly, HDFS stores each file as blocks which are scattered throughout the Apache Hadoop cluster. The default size of each block is 128 MB in Apache Hadoop 2.x (64 MB in Apache Hadoop 1.x) which you can configure as per your requirement.



It is not necessary that in HDFS, each file is stored in exact multiple of the configured block size (128 MB, 256 MB etc.). Let's take an example where I have a file "example.txt" of size 514 MB as shown in above figure. Suppose that we are using the default configuration of block size, which is 128 MB. Then, how many blocks will be created? 5, Right. The first four blocks will be of 128 MB. But, the last block will be of 2 MB size only.

Now, you must be thinking why we need to have such a huge blocks size i.e. 128 MB?
Well, whenever we talk about HDFS, we talk about huge data sets, i.e. Terabytes and Petabytes of data. So, if we had a block size of let's say of 4 KB, as in Linux file system, we would be having too many blocks and therefore too much of the metadata. So, managing these no. of blocks and metadata will create huge overhead, which is something, we don't want.

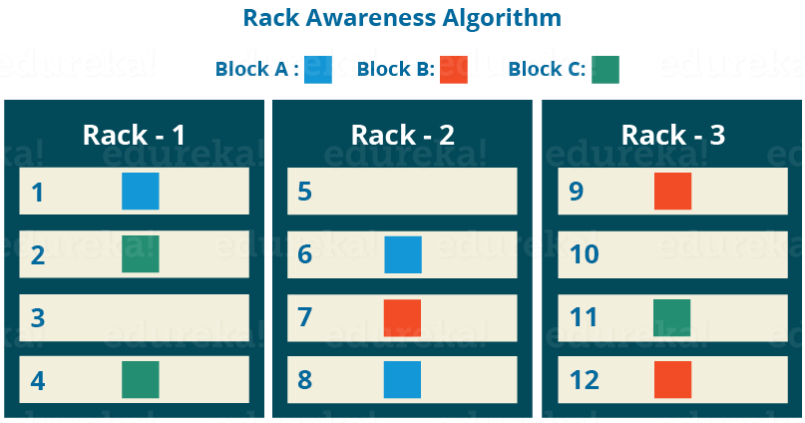
Replication Management:
HDFS provides a reliable way to store huge data in a distributed environment as data blocks. The blocks are also replicated to provide fault tolerance. The default replication factor is 3 which is again configurable. So, as you can see in the figure below where each block is replicated three times and stored on different DataNodes (considering the default replication factor):



Therefore, if you are storing a file of 128 MB in HDFS using the default configuration, you will end up occupying a space of 384 MB (3*128 MB) as the blocks will be replicated three times and each replica will be residing on a different DataNode.

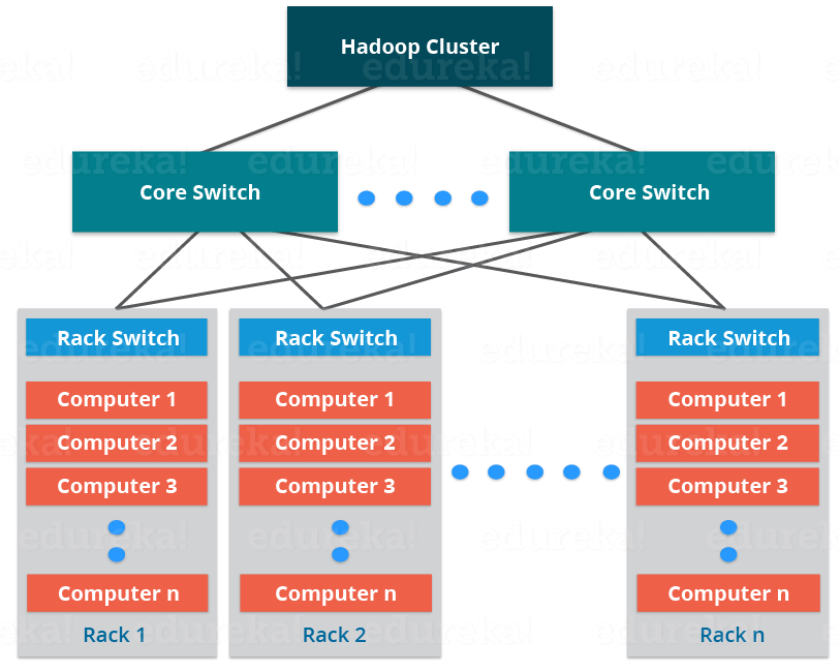
Note: The NameNode collects block report from DataNode periodically to maintain the replication factor. Therefore, whenever a block is over-replicated or under-replicated the NameNode deletes or add replicas as needed.

Rack Awareness:



Anyways, moving ahead, let's talk more about how HDFS places replica and what is rack awareness? Again, the NameNode also ensures that all the replicas are not stored on the same rack or a single rack. It follows an in-built Rack Awareness Algorithm to reduce latency as well as provide fault tolerance. Considering the replication factor is 3, the Rack Awareness Algorithm says that the first replica of a block will be stored on a local rack and the next two replicas will be stored on a different (remote) rack but, on a different DataNode within that (remote) rack as shown in the figure above. If you have more replicas, the rest of the replicas will be placed on random DataNodes provided not more than two replicas reside on the same rack, if possible.

This is how an actual Hadoop production cluster looks like. Here, you have multiple racks populated with DataNodes:



Advantages of Rack Awareness:

So, now you will be thinking why do we need a Rack Awareness algorithm? The reasons are:

- **To improve the network performance:** The communication between nodes residing on different racks is directed via switch. In general, you will find *greater network bandwidth* between machines in the same rack than the machines residing in different rack. So, the Rack Awareness helps you to have reduce write traffic in between different racks and thus providing a better write performance. Also, you will be gaining increased read performance because you are using the bandwidth of multiple racks.

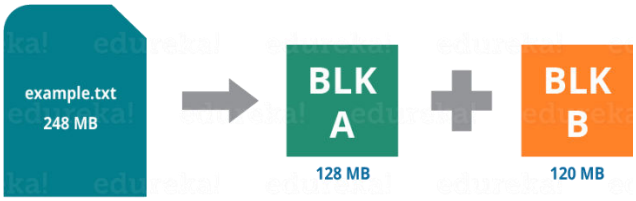
- **To prevent loss of data:** We don't have to worry about the data even if an entire rack fails because of the switch failure or power failure. And if you think about it, it will make sense, as it is said that *never put all your eggs in the same basket.*

HDFS Read/ Write Architecture:

Now let's talk about how the data read/write operations are performed on HDFS. HDFS follows Write Once – Read Many Philosophy. So, you can't edit files already stored in HDFS. But, you can append new data by re-opening the file.

HDFS Write Architecture:

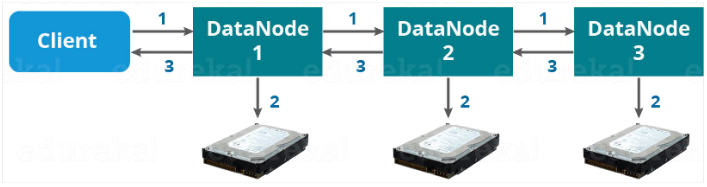
Suppose a situation where an HDFS client, wants to write a file named "example.txt" of size 248 MB.



Assume that the system block size is configured for 128 MB (default). So, the client will be dividing the file "example.txt" into 2 blocks – one of 128 MB (Block A) and the other of 120 MB (block B).
Now, the following protocol will be followed whenever the data is written into HDFS:

- At first, the HDFS client will reach out to the NameNode for a Write Request against the two blocks, say, Block A & Block B.
- The NameNode will then grant the client the write permission and will provide the IP addresses of the DataNodes where the file blocks will be copied eventually.
- The selection of IP addresses of DataNodes is purely randomized based on availability, replication factor and rack awareness that we have discussed earlier.

- Let's say the replication factor is set to default i.e. 3. Therefore, for each block the NameNode will be providing the client a list of (3) IP addresses of DataNodes. The list will be unique for each block.
- Suppose, the NameNode provided following lists of IP addresses to the client:
    - For Block A, list A = {IP of DataNode 1, IP of DataNode 4, IP of DataNode 6}
    - For Block B, set B = {IP of DataNode 3, IP of DataNode 7, IP of DataNode 9}
- Each block will be copied in three different DataNodes to maintain the replication factor consistent throughout the cluster.
- Now the whole data copy process will happen in three stages:

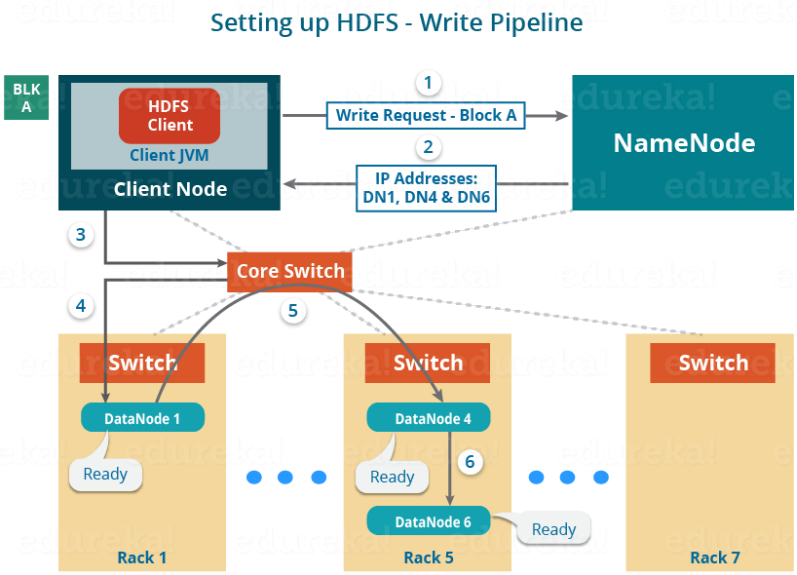


1. Set up of Pipeline
2. Data streaming and replication
3. Shutdown of Pipeline (Acknowledgement stage)

1. Set up of Pipeline:

Before writing the blocks, the client confirms whether the DataNodes, present in each of the list of IPs, are ready to receive the data or not. In doing so, the client creates a pipeline for each of the blocks by connecting the individual DataNodes in the respective list for that block. Let us consider Block A. The list of DataNodes provided by the NameNode is:

For Block A, list A = {IP of DataNode 1, IP of DataNode 4, IP of DataNode 6}.

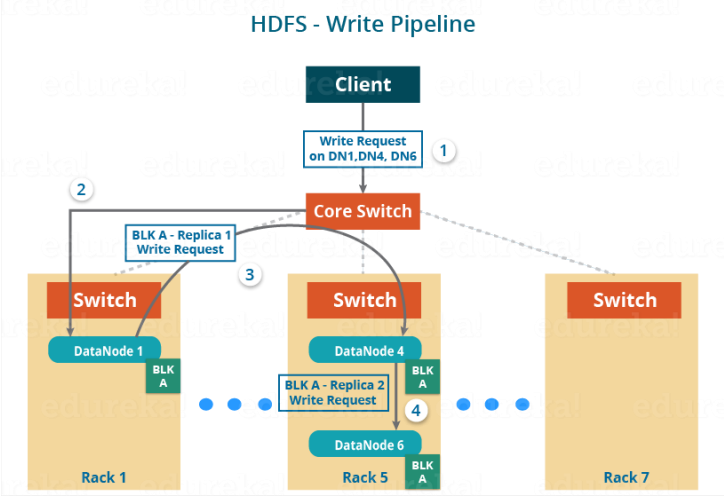


So, for block A, the client will be performing the following steps to create a pipeline:

- The client will choose the first DataNode in the list (DataNode IPs for Block A) which is DataNode 1 and will establish a TCP/IP connection.
- The client will inform DataNode 1 to be ready to receive the block. It will also provide the IPs of next two DataNodes (4 and 6) to the DataNode 1 where the block is supposed to be replicated.
- The DataNode 1 will connect to DataNode 4. The DataNode 1 will inform DataNode 4 to be ready to receive the block and will give it the IP of DataNode 6. Then, DataNode 4 will tell DataNode 6 to be ready for receiving the data.
- Next, the acknowledgement of readiness will follow the reverse sequence, i.e. From the DataNode 6 to 4 and then to 1.
- At last DataNode 1 will inform the client that all the DataNodes are ready and a pipeline will be formed between the client, DataNode 1, 4 and 6.
- Now pipeline set up is complete and the client will finally begin the data copy or streaming process.

2. Data Streaming:

As the pipeline has been created, the client will push the data into the pipeline. Now, don't forget that in HDFS, data is replicated based on replication factor. So, here Block A will be stored to three DataNodes as the assumed replication factor is 3. Moving ahead, the client will copy the block (A) to DataNode 1 only. The replication is always done by DataNodes sequentially.

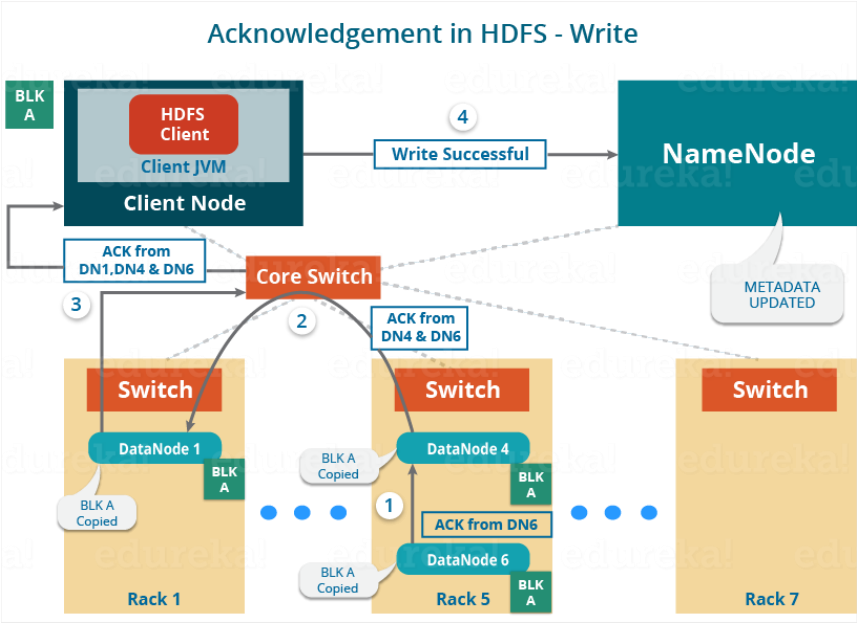


So, the following steps will take place during replication:

- Once the block has been written to DataNode 1 by the client, DataNode 1 will connect to DataNode 4.
- Then, DataNode 1 will push the block in the pipeline and data will be copied to DataNode 4.
- Again, DataNode 4 will connect to DataNode 6 and will copy the last replica of the block.

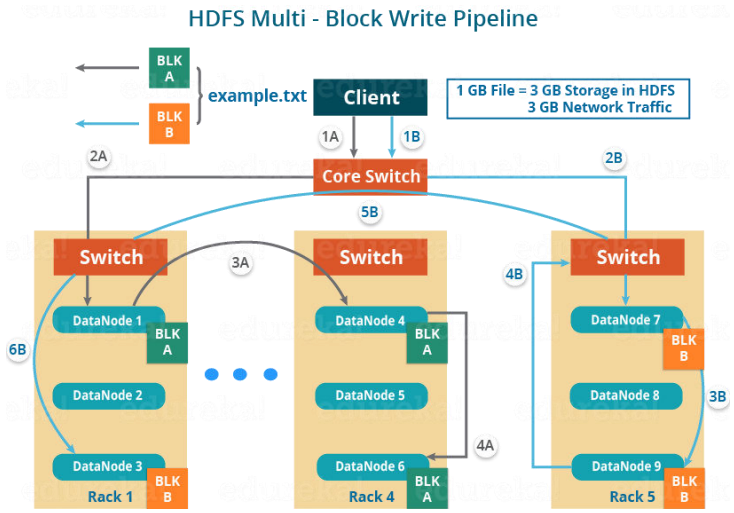
3. Shutdown of Pipeline or Acknowledgement stage:
Once the block has been copied into all the three DataNodes, a series of acknowledgements will take place to ensure the client and NameNode that the data has been written successfully. Then, the client will finally close the pipeline to end the TCP session.

As shown in the figure below, the acknowledgement happens in the reverse sequence i.e. from DataNode 6 to 4 and then to 1. Finally, the DataNode 1 will push three acknowledgements (including its own) into the pipeline and send it to the client. The client will inform NameNode that data has been written successfully. The NameNode will update its metadata and the client will shut down the pipeline.



Similarly, Block B will also be copied into the DataNodes in parallel with Block A. So, the following things are to be noticed here:

- The client will copy Block A and Block B to the first DataNode **simultaneously**.
- Therefore, in our case, two pipelines will be formed for each of the block and all the process discussed above will happen in parallel in these two pipelines.
- The client writes the block into the first DataNode and then the DataNodes will be replicating the block sequentially.

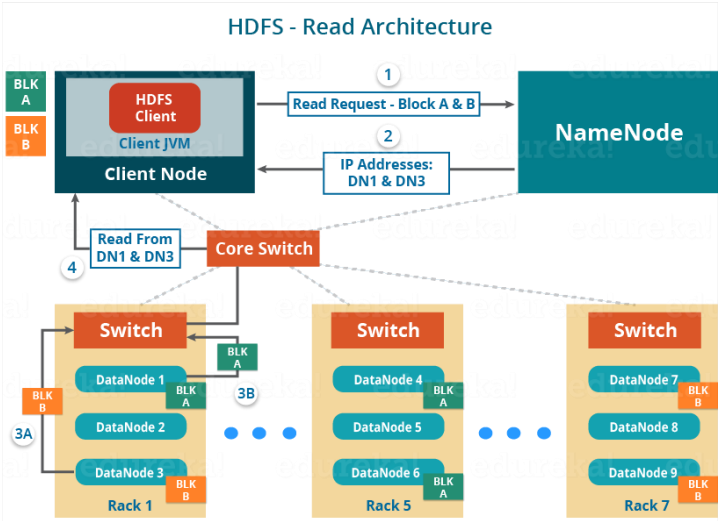


As you can see in the above image, there are two pipelines formed for each block (A and B). Following is the flow of operations that is taking place for each block in their respective pipelines:

- For Block A: 1A -> 2A -> 3A -> 4A
- For Block B: 1B -> 2B -> 3B -> 4B -> 5B -> 6B

HDFS Read Architecture:

HDFS Read architecture is comparatively easy to understand. Let's take the above example again where the HDFS client wants to read the file "example.txt" now.



Now, following steps will be taking place while reading the file:

- The client will reach out to NameNode asking for the block metadata for the file "example.txt".
- The NameNode will return the list of DataNodes where each block (Block A and B) are stored.
- After that client, will connect to the DataNodes where the blocks are stored.
- The client starts reading data parallel from the DataNodes (Block A from DataNode 1 and Block B from DataNode 3).
- Once the client gets all the required file blocks, it will combine these blocks to form a file.

While serving read request of the client, HDFS selects the replica which is closest to the client. This reduces the read latency and the bandwidth consumption. Therefore, that replica is selected which resides on the same rack as the reader node, if possible.

HDFS 2.x High Availability Cluster Architecture

HDFS 2.x High Availability Cluster Architecture and the procedure to set up an HDFS High Availability cluster. The order in which the topics have been covered in this blog are as follows:

- HDFS HA Architecture
  - Introduction
  - NameNode Availability
  - Architecture of HA
  - Implementation of HA (JournalNode and Shared storage)

Introduction:

The concept of High Availability cluster was introduced in Hadoop 2.x to solve the single point of failure problem in Hadoop 1.x. As you know from my previous blog that the ***HDFS Architecture*** follows Master/Slave Topology where NameNode acts as a master daemon and is responsible for managing other slave nodes called DataNodes. This single Master Daemon or NameNode becomes a bottleneck. Although, the introduction of Secondary NameNode did prevent us from data loss and offloading some of the burden of the NameNode but, it did not solve the availability issue of the NameNode.

NameNode Availability:

If you consider the standard configuration of HDFS cluster, the NameNode becomes a **single point of failure**. It happens because the moment the NameNode becomes unavailable, the whole cluster becomes unavailable until someone restarts the NameNode or brings a new one.
The reasons for unavailability of NameNode can be:

- A planned event like maintenance work such has upgradation of software or hardware.
- It may also be due to an unplanned event where the NameNode crashes because of some reasons.

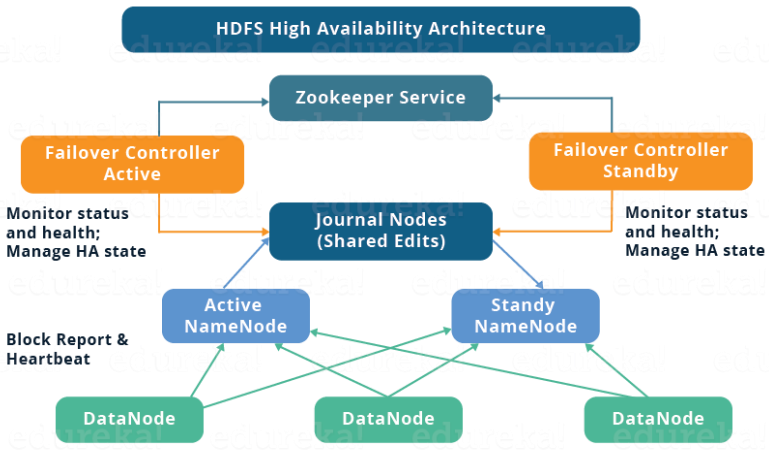
In either of the above cases, we have a downtime where we are not able to use the HDFS cluster which becomes a challenge.

HDFS HA Architecture:

Let us understand that how HDFS HA Architecture solved this critical problem of NameNode availability:

The HA architecture solved this problem of NameNode availability by allowing us to have two NameNodes in an active/passive configuration. So, we have two running NameNodes at the same time in a High Availability cluster:

- Active NameNode
- Standby/Passive NameNode.



If one NameNode goes down, the other NameNode can take over the responsibility and therefore, reduce the cluster down time. The standby NameNode serves the purpose of a backup NameNode (unlike the Secondary NameNode) which incorporate failover capabilities to the Hadoop cluster. Therefore, with the StandbyNode, we can have automatic failover whenever a NameNode crashes (unplanned event) or we can have a graceful (manually initiated) failover during the maintenance period.

There are two issues in maintaining consistency in the HDFS High Availability cluster:

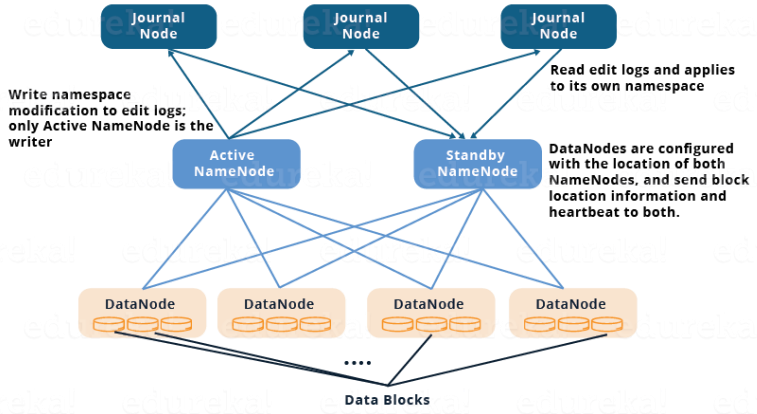
- Active and Standby NameNode should always be in sync with each other, i.e. They should have the same metadata. This will allow us to restore the Hadoop cluster to the same namespace state where it got crashed and therefore, will provide us to have fast failover.
- There should be only one active NameNode at a time because two active NameNode will lead to corruption of the data.*This kind of scenario is termed as a split-brain scenario where a cluster gets divided into smaller cluster, each one believing that it is the only active cluster.* To avoid such scenarios fencing is done. Fencing is a process of ensuring that only one NameNode remains active at a particular time.

Implementation of HA Architecture:
Now, you know that in HDFS HA Architecture, we have two NameNodes running at the same time. So, we can implement the Active and Standby NameNode configuration in following two ways:

1. Using Quorum Journal Nodes
2. Shared Storage using NFS

Let us understand these two ways of implementation taking one at a time:

1. Using Quorum Journal Nodes:



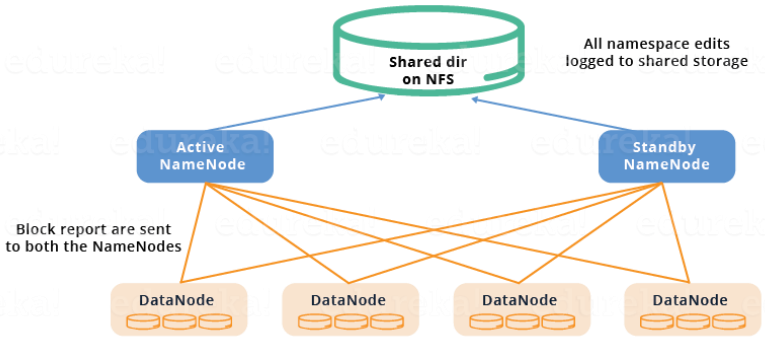
- The standby NameNode and the active NameNode keep in sync with each other through a separate group of nodes or daemons - called **JournalNodes**. The JournalNodes follows the ring topology where the nodes are connected to each other to form a ring. The JournalNode serves the request coming to it and copies the information into other nodes in the ring.This provides fault tolerance in case of JournalNode failure.
- The active NameNode is responsible for updating the EditLogs (metadata information) present in the JournalNodes.
- The StandbyNode reads the changes made to the EditLogs in the JournalNode and applies it to its own namespace in a constant manner.
- During failover, the StandbyNode makes sure that it has updated its meta data information from the JournalNodes before becoming the new Active NameNode. This makes the current namespace state synchronized with the state before failover.
- The IP Addresses of both the NameNodes are available to all the DataNodes and they send their heartbeats and block location information to both the NameNode. This provides a fast failover (less down time) as the StandbyNode has an updated information about the block location in the cluster.

Fencing of NameNode:

Now, as discussed earlier, it is very important to ensure that there is only one Active NameNode at a time. So, fencing is a process to ensure this very property in a cluster.

- The JournalNodes performs this fencing by allowing only one NameNode to be the writer at a time.
- The Standby NameNode takes over the responsibility of writing to the JournalNodes and forbid any other NameNode to remain active.
- Finally, the new Active NameNode can perform its activities safely.

2. Using Shared Storage:



- The StandbyNode and the active NameNode keep in sync with each other by using a **shared storage device**. The active NameNode logs the record of any modification done in its namespace to an EditLog present in this shared storage. The StandbyNode reads the changes made to the EditLogs in this shared storage and applies it to its own namespace.
- Now, in case of failover, the StandbyNode updates its metadata information using the EditLogs in the shared storage at first. Then, it takes the responsibility of the Active NameNode. This makes the current namespace state synchronized with the state before failover.
- The administrator must configure at least one fencing method to avoid a split-brain scenario.
- The system may employ a range of fencing mechanisms. It may include killing of the NameNode's process and revoking its access to the shared storage directory.
- As a last resort, we can fence the previously active NameNode with a technique known as STONITH, or "shoot the other node in the head". STONITH uses a specialized power distribution unit to forcibly power down the NameNode machine.

Automatic Failover:
Failover is a procedure by which a system automatically transfers control to secondary system when it detects a fault or failure. There are two types of failover:

Graceful Failover: In this case, we manually initiate the failover for routine maintenance.
Automatic Failover: In this case, the failover is initiated automatically in case of NameNode failure (unplanned event).

Apache Zookeeper is a service that provides the automatic failover capability in HDFS High Availabilty cluster. It maintains small amounts of coordination data, informs clients of changes in that data, and monitors clients for failures. Zookeeper maintains a session with the NameNodes. In case of failure, the session will expire and the Zookeeper will inform other NameNodes to initiate the failover process. In case of NameNode failure, other passive NameNode can take a lock in Zookeeper stating that it wants to become the next Active NameNode.
The ZookeerFailoverController (ZKFC) is a Zookeeper client that also monitors and manages the NameNode status. Each of the NameNode runs a ZKFC also. ZKFC is responsible for monitoring the health of the NameNodes periodically.
Now that you have understood what is High Availability in a Hadoop cluster, it's time to set it up. To set up High Availability in Hadoop cluster you have to use Zookeeper in all the nodes.
The daemons in Active NameNode are:

- Zookeeper
- Zookeeper Fail Over controller
- JournalNode
- NameNode

The daemons in Standby NameNode are:

- Zookeeper
- Zookeeper Fail Over controller
- JournalNode
- NameNode

The daemons in DataNode are:

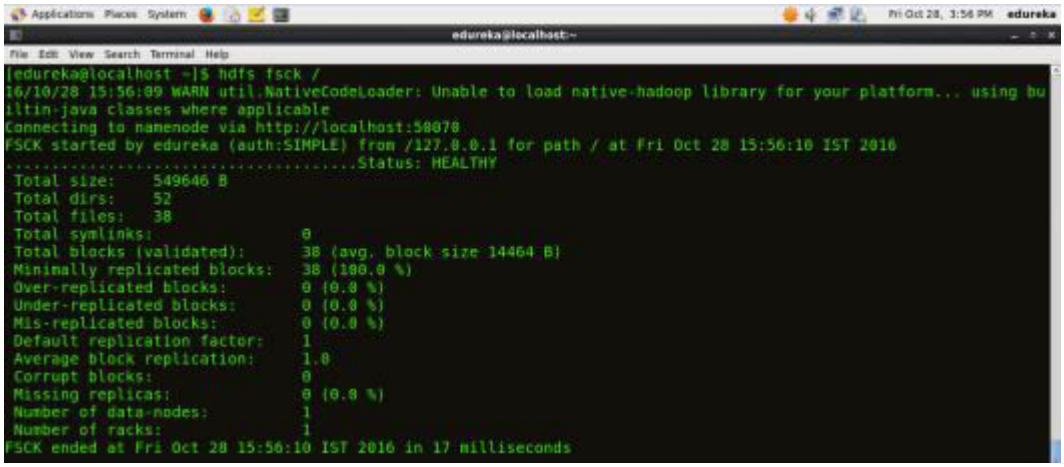
- Zookeeper
- JournalNode
- DataNode

HDFS Commands

- **fsck**

HDFS Command to check the health of the Hadoop file system.

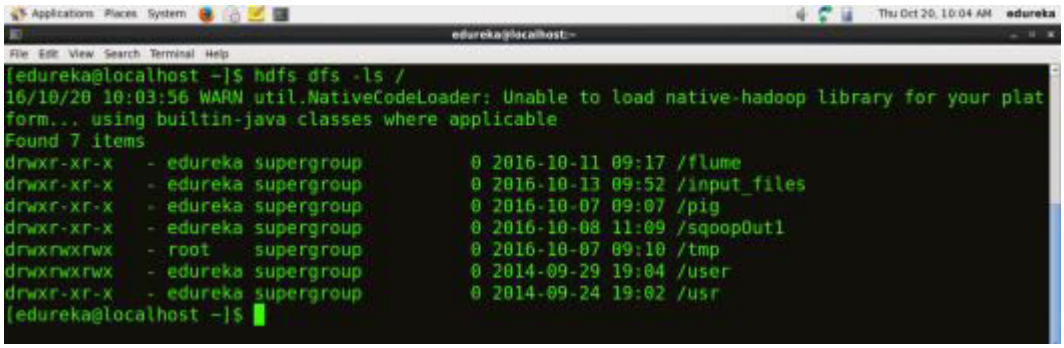
Command: **hdfs fsck /**



- **ls**

HDFS Command to display the list of Files and Directories in HDFS.

Command: **hdfs dfs –ls /**

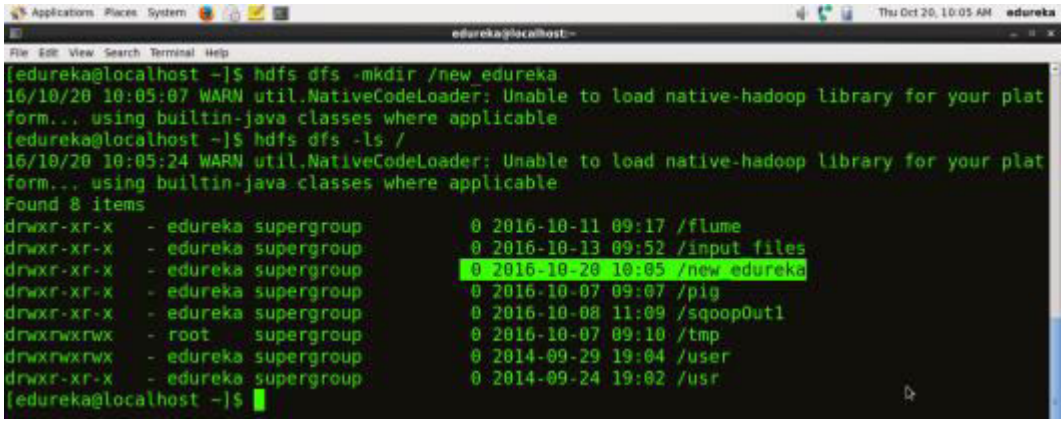


- **mkdir**

HDFS Command to create the directory in HDFS.

Usage: **hdfs dfs –mkdir /directory_name**

Command: **hdfs dfs –mkdir /new_edureka**



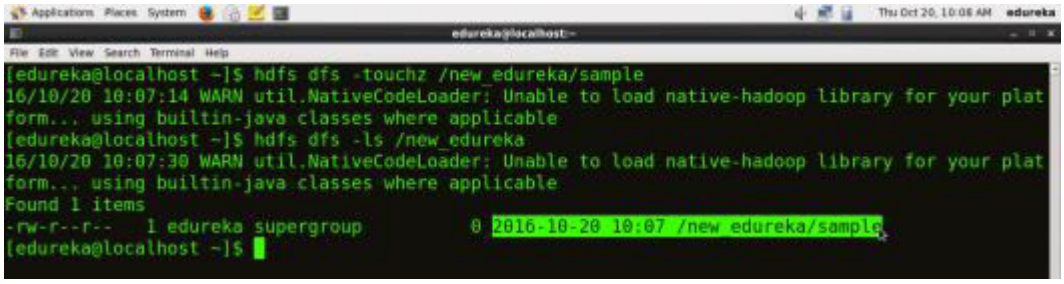
Note: *Here we are trying to create a directory named "new_edureka" in HDFS.*

- **touchz**

HDFS Command to create a file in HDFS with file size 0 bytes.

Usage: **hdfs dfs –touchz /directory/filename**

Command: **hdfs dfs –touchz /new_edureka/sample**



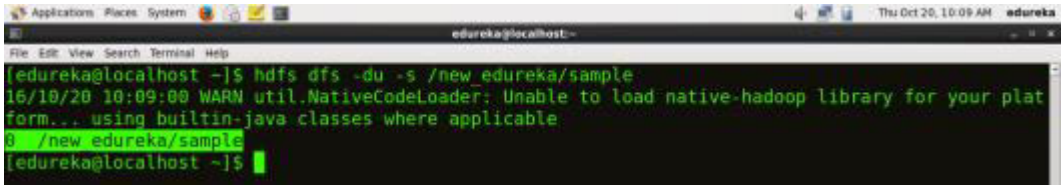
Note: *Here we are trying to create a file named "sample" in the directory "new_edureka" of hdfs with file size 0 bytes.*

- **du**

HDFS Command to check the file size.

Usage: **hdfs dfs –du –s /directory/filename**

Command: **hdfs dfs –du –s /new_edureka/sample**

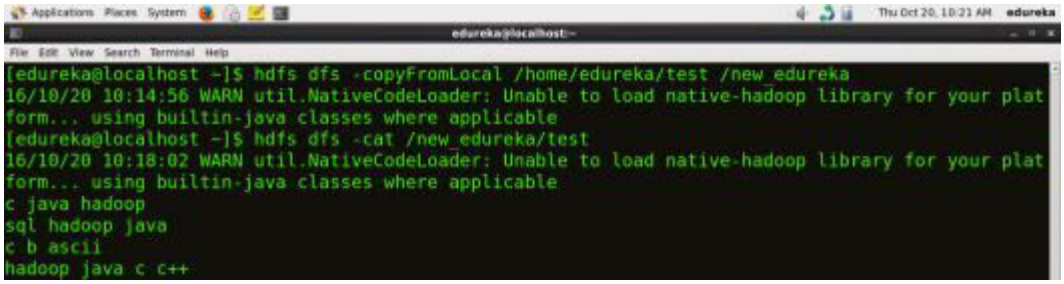


- **cat**

HDFS Command that reads a file on HDFS and prints the content of that file to the standard output.

Usage: **hdfs dfs –cat /path/to/file_in_hdfs**

Command: **hdfs dfs –cat /new_edureka/test**

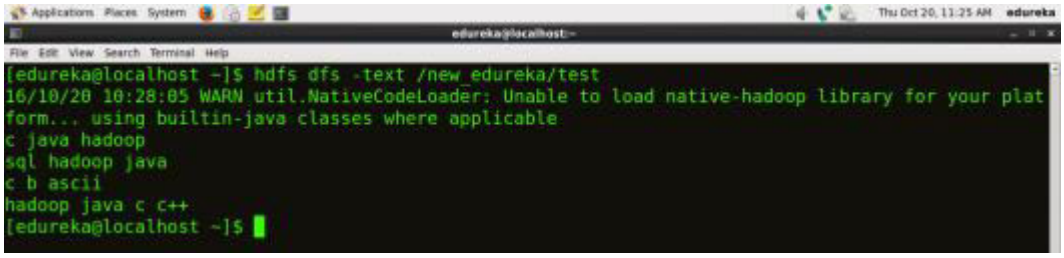


- **text**

HDFS Command that takes a source file and outputs the file in text format.

Usage: **hdfs dfs –text /directory/filename**

Command: **hdfs dfs –text /new_edureka/test**

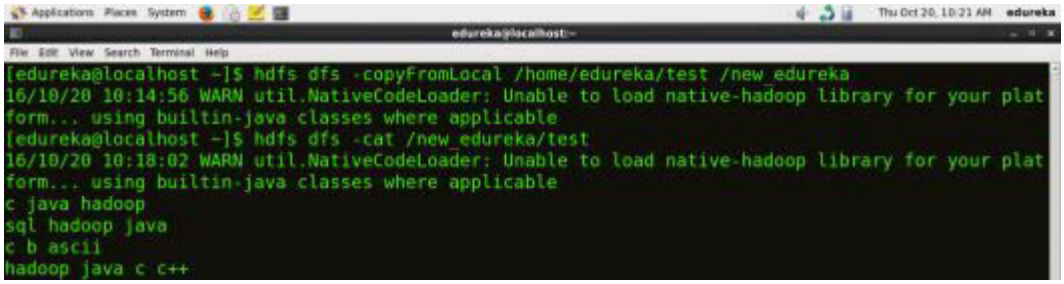


- **copyFromLocal**

HDFS Command to copy the file from a Local file system to HDFS.

Usage: **hdfs dfs -copyFromLocal <localsrc> <hdfs destination>**

Command: **hdfs dfs –copyFromLocal /home/edureka/test /new_edureka**



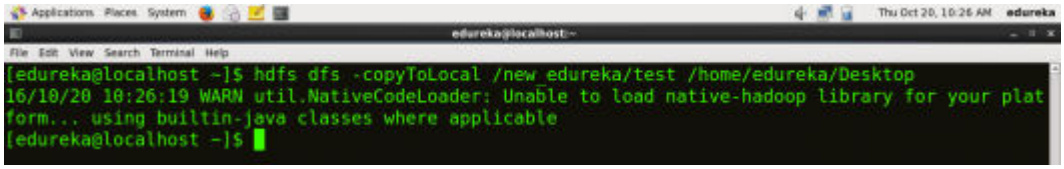
Note: *Here the test is the file present in the local directory /home/edureka and after the command gets executed the test file will be copied in /new_edureka directory of HDFS.*

- **copyToLocal**

HDFS Command to copy the file from HDFS to Local File System.

Usage: **hdfs dfs -copyToLocal <hdfs source> <localdst>**

Command: **hdfs dfs –copyToLocal /new_edureka/test /home/edureka**



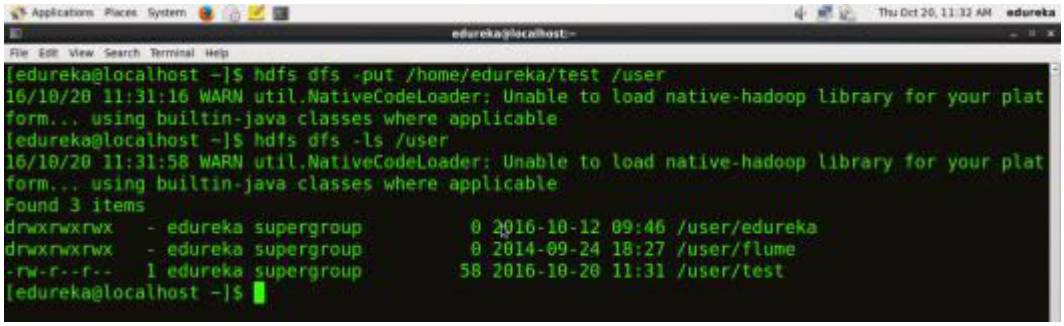
Note: *Here test is a file present in the new_edureka directory of HDFS and after the command gets executed the test file will be copied to local directory /home/edureka*

- **put**

HDFS Command to copy single source or multiple sources from local file system to the destination file system.

Usage: **hdfs dfs -put <localsrc> <destination>**

Command: **hdfs dfs –put /home/edureka/test /user**



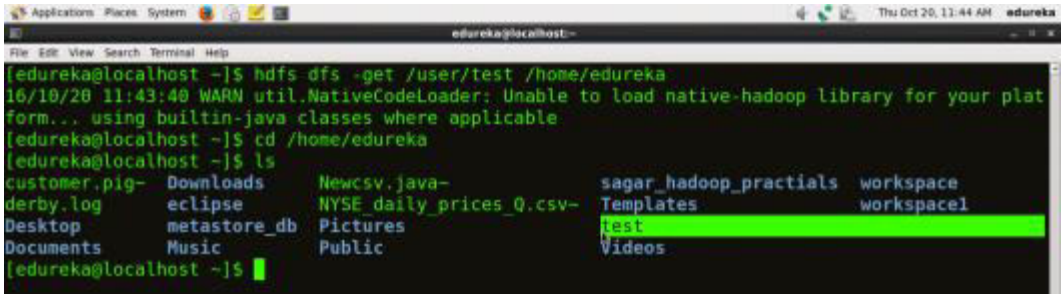
Note: *The command copyFromLocal is similar to put command, except that the source is restricted to a local file reference.*

- **get**

HDFS Command to copy files from hdfs to the local file system.

Usage: **hdfs dfs -get <src> <localdst>**

Command: **hdfs dfs –get /user/test /home/edureka**



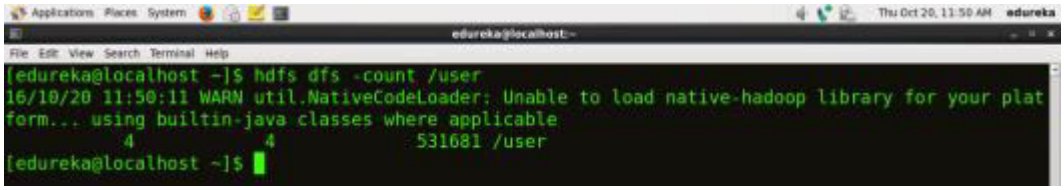
Note: *The command copyToLocal is similar to get command, except that the destination is restricted to a local file reference.*

- **count**

HDFS Command to count the number of directories, files, and bytes under the paths that match the specified file pattern.

Usage: **hdfs dfs -count <path>**

Command: **hdfs dfs –count /user**

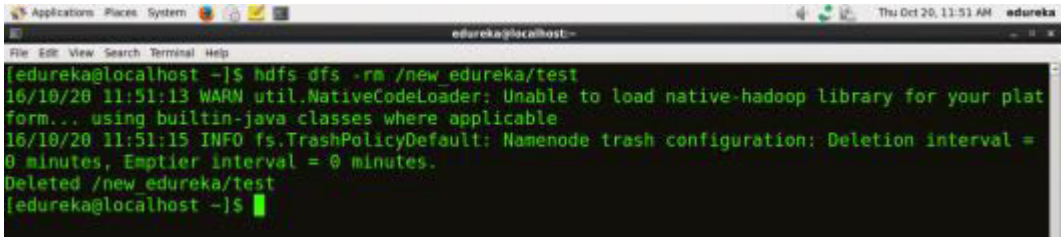


- **rm**

HDFS Command to remove the file from HDFS.

Usage: **hdfs dfs –rm <path>**

Command: **hdfs dfs –rm /new_edureka/test**

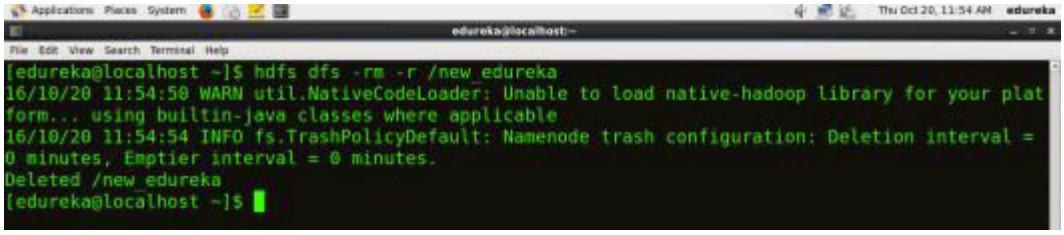


- **rm -r**

HDFS Command to remove the entire directory and all of its content from HDFS.

Usage: **hdfs dfs -rm -r <path>**

Command: **hdfs dfs -rm -r /new_edureka**



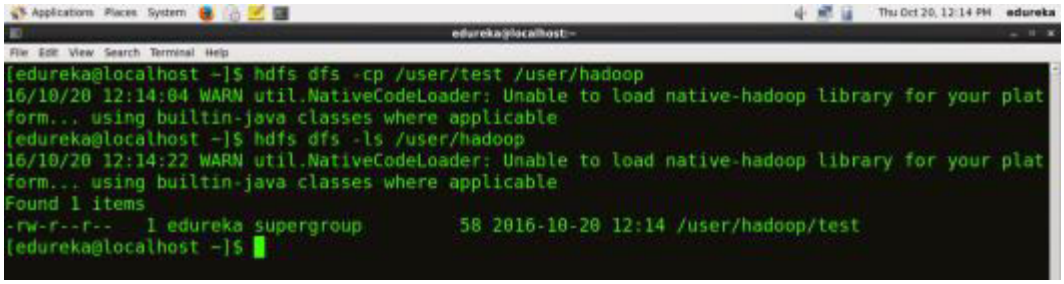
- **cp**

HDFS Command to copy files from source to destination. This command allows multiple sources as well, in which case the destination must be a directory.

Usage: **hdfs dfs -cp <src> <dest>**

Command: **hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2**

Command: **hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2 /user/hadoop/dir**

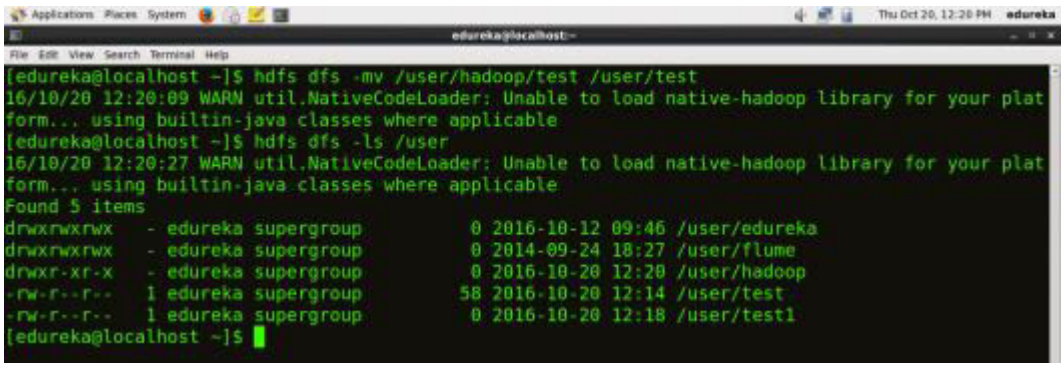


- **mv**

HDFS Command to move files from source to destination. This command allows multiple sources as well, in which case the destination needs to be a directory.

Usage: **hdfs dfs -mv <src> <dest>**

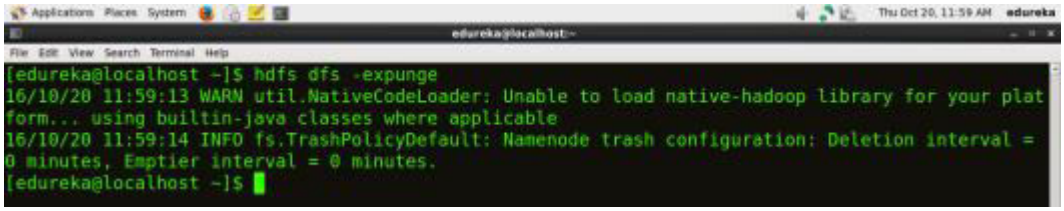
Command: **hdfs dfs -mv /user/hadoop/file1 /user/hadoop/file2**



- **expunge**

HDFS Command that makes the trash empty.

Command: hdfs dfs -expunge

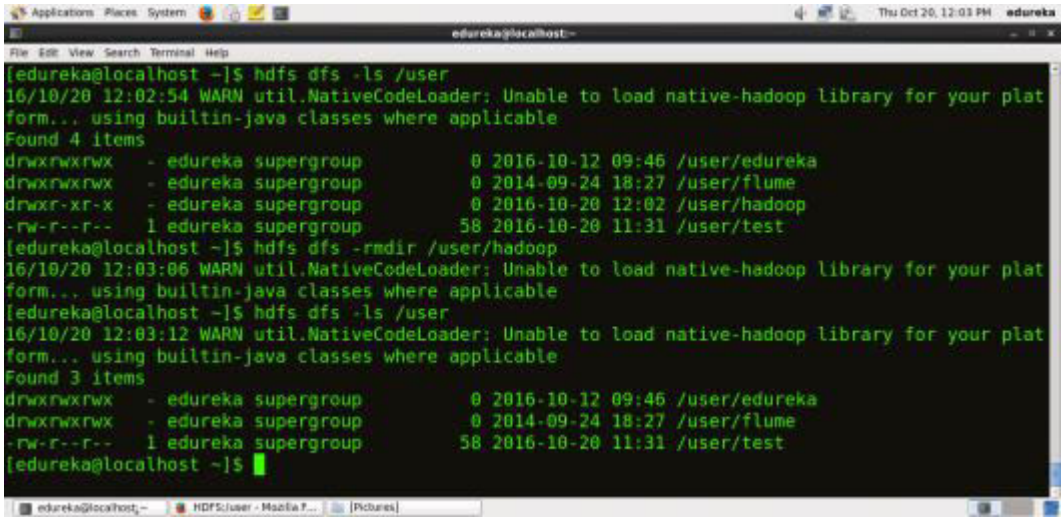


- **rmdir**

HDFS Command to remove the directory.

Usage: **hdfs dfs -rmdir <path>**

Command: **hdfs dfs –rmdir /user/hadoop**

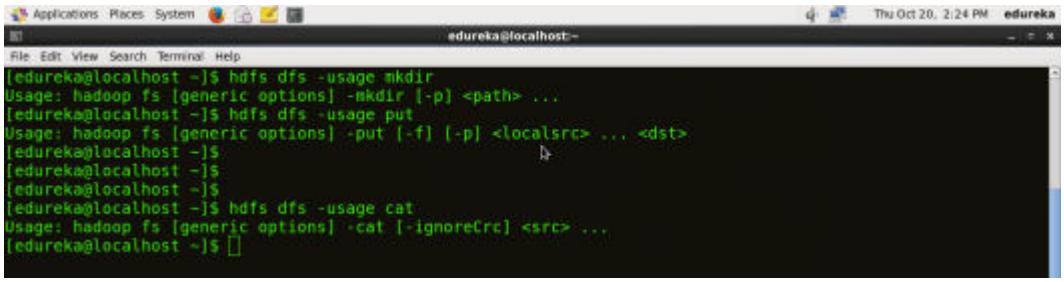


- **usage**

HDFS Command that returns the help for an individual command.

Usage: **hdfs dfs -usage <command>**

Command: **hdfs dfs -usage mkdir**



Note: *By using usage command you can get information about any command.*

- **help**

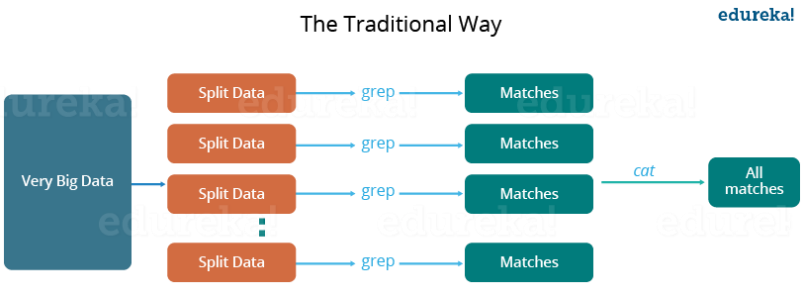
HDFS Command that displays help for given command or all commands if none is specified.

Command: **hdfs dfs -help**

MapReduce Introduction

Google released a paper on MapReduce technology in December, 2004. This became the genesis of the Hadoop Processing Model. So, MapReduce is a programming model that allows us to perform parallel and distributed processing on huge data sets.

MapReduce Tutorial: Traditional Way



Let us understand, when the MapReduce framework was not there, how parallel and distributed processing used to happen in a traditional way. So, let us take an example where I have a weather log containing the daily average temperature of the years from 2000 to 2015. Here, I want to calculate the day having the highest temperature in each year.

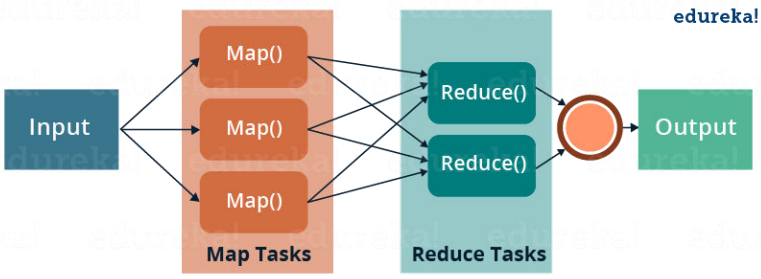
So, just like in the traditional way, I will split the data into smaller parts or blocks and store them in different machines. Then, I will find the highest temperature in each part stored in the corresponding machine. At last, I will combine the results received from each of the machines to have the final output. Let us look at the challenges associated with this traditional approach:

1. **Critical path problem:** It is the amount of time taken to finish the job without delaying the next milestone or actual completion date. So, if, any of the machines delays the job, the whole work gets delayed.
2. **Reliability problem:** What if, any of the machines which is working with a part of data fails? The management of this failover becomes a challenge.
3. **Equal split issue:** How will I divide the data into smaller chunks so that each machine gets even part of data to work with. In other words, how to equally divide the data such that no individual machine is overloaded or under utilized.
4. **Single split may fail:** If any of the machine fails to provide the output, I will not be able to calculate the result. So, there should be a mechanism to ensure this fault tolerance capability of the system.
5. **Aggregation of result:** There should be a mechanism to aggregate the result generated by each of the machines to produce the final output.

These are the issues which I will have to take care individually while performing parallel processing of huge data sets when using traditional approaches.

To overcome these issues, we have the MapReduce framework which allows us to perform such parallel computations without bothering about the issues like reliability, fault tolerance etc. Therefore, MapReduce gives you the flexibility to write code logic without caring about the design issues of the system.

MapReduce Tutorial: What is MapReduce?



MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.

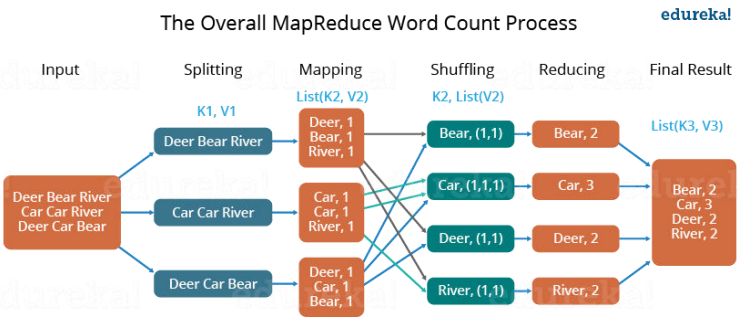
- MapReduce consists of two distinct tasks – Map and Reduce.
- As the name MapReduce suggests, reducer phase takes place after mapper phase has been completed.
- So, the first is the map job, where a block of data is read and processed to produce key-value pairs as intermediate outputs.
- The output of a Mapper or map job (key-value pairs) is input to the Reducer.
- The reducer receives the key-value pair from multiple map jobs.

- Then, the reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

MapReduce Tutorial: A Word Count Example of MapReduce
Let us understand, how a MapReduce works by taking an example where I have a text file called example.txt whose contents are as follows:

Dear, Bear, River, Car, Car, River, Deer, Car and Bear

Now, suppose, we have to perform a word count on the sample.txt using MapReduce. So, we will be finding the unique words and the number of occurrences of those unique words.



- First, we divide the input in three splits as shown in the figure. This will distribute the work among all the map nodes.
- Then, we tokenize the words in each of the mapper and give a hardcoded value (1) to each of the tokens or words. The rationale behind giving a hardcoded value equal to 1 is that every word, in itself, will occur once.
- Now, a list of key-value pair will be created where the key is nothing but the individual words and value is one. So, for the first line (Dear Bear River) we have 3 key-value pairs – Dear, 1; Bear, 1; River, 1. The mapping process remains the same on all the nodes.
- After mapper phase, a partition process takes place where sorting and shuffling happens so that all the tuples with the same key are sent to the corresponding reducer.
- So, after the sorting and shuffling phase, each reducer will have a unique key and a list of values corresponding to that very key. For example, Bear, [1,1]; Car, [1,1,1].., etc.
- Now, each Reducer counts the values which are present in that list of values. As shown in the figure, reducer gets a list of values which is [1,1] for the key Bear. Then, it counts the number of ones in the very list and gives the final output as – Bear, 2.
- Finally, all the output key/value pairs are then collected and written in the output file.

MapReduce Tutorial: Advantages of MapReduce

The two biggest advantages of MapReduce are:

1. **Parallel Processing:**

In MapReduce, we are dividing the job among multiple nodes and each node works with a part of the job simultaneously. So, MapReduce is based on Divide and Conquer paradigm which helps us to process the data using different machines. As the data is processed by multiple machine instead of a single machine in parallel, the time taken to process the data gets reduced by a tremendous amount as shown in the figure below (2).

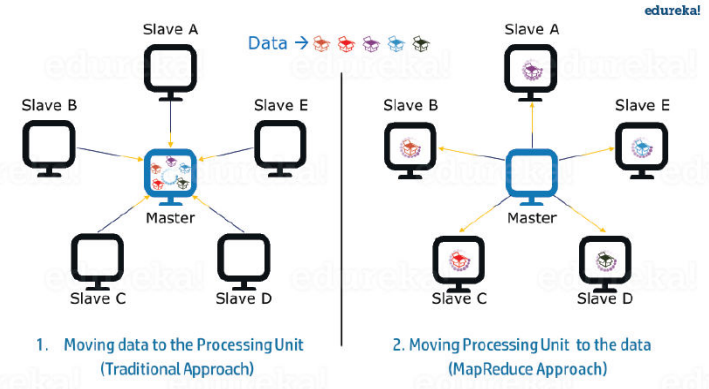


Fig.: Traditional Way Vs. MapReduce Way – MapReduce Tutorial

2. Data Locality:

Instead of moving data to the processing unit, we are moving processing unit to the data in the MapReduce Framework. In the traditional system, we used to bring data to the processing unit and process it. But, as the data grew and became very huge, bringing this huge amount of data to the processing unit posed following issues:

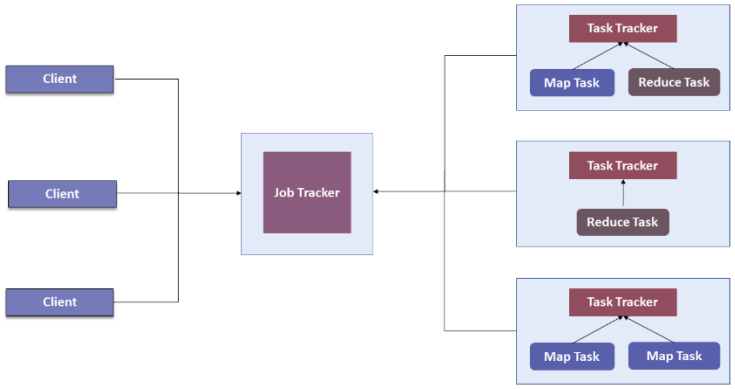
- Moving huge data to processing is costly and deteriorates the network performance.
- Processing takes time as the data is processed by a single unit which becomes the bottleneck.
- Master node can get over-burdened and may fail.

Now, MapReduce allows us to overcome above issues by bringing the processing unit to the data. So, as you can see in the above image that the data is distributed among multiple nodes where each node processes the part of the data residing on it. This allows us to have the following advantages:

- It is very cost effective to move processing unit to the data.
- The processing time is reduced as all the nodes are working with their part of the data in parallel.
- Every node gets a part of the data to process and therefore, there is no chance of a node getting overburdened.

YARN

In Hadoop version 1.0 which is also referred to as MRV1(MapReduce Version 1), MapReduce performed both processing and resource management functions. It consisted of a Job Tracker which was the single master. The Job Tracker allocated the resources, performed scheduling and monitored the processing jobs. It assigned map and reduce tasks on a number of subordinate processes called the Task Trackers. The Task Trackers periodically reported their progress to the Job Tracker.



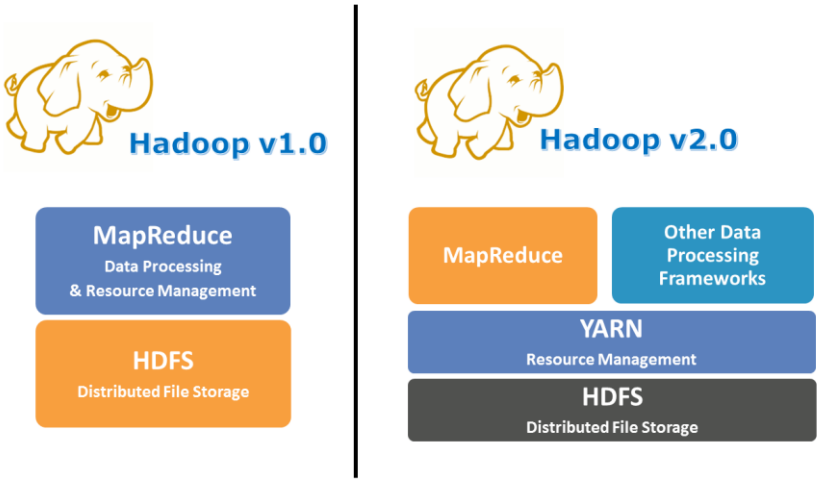
This design resulted in scalability bottleneck due to a single Job Tracker. IBM mentioned in its article that according to Yahoo!, the practical limits of such a design are reached with a cluster of 5000 nodes and 40,000 tasks running concurrently. Apart from this limitation, the utilization of computational resources is inefficient in MRV1. Also, the Hadoop framework became limited only to MapReduce processing paradigm.

To overcome all these issues, YARN was introduced in Hadoop version 2.0 in the year 2012 by Yahoo and Hortonworks. The basic idea behind YARN is to relieve MapReduce by taking over the responsibility of Resource Management and Job Scheduling. YARN started to give Hadoop the ability to run non-MapReduce jobs within the Hadoop framework.

With the introduction of YARN, the ***Hadoop ecosystem*** was completely revolutionalized. It became much more flexible, efficient and scalable. When Yahoo went live with YARN in the first quarter of 2013, it aided the company to shrink the size of its Hadoop cluster from 40,000 nodes to 32,000 nodes. But the number of jobs doubled to 26 million per month.

Introduction to Hadoop YARN

Now that I have enlightened you with the need for YARN, let me introduce you to the core component of Hadoop v2.0, *YARN*. YARN allows different data processing methods like graph processing, interactive processing, stream processing as well as batch processing to run and process data stored in HDFS. Therefore YARN opens up Hadoop to other types of distributed applications beyond MapReduce.



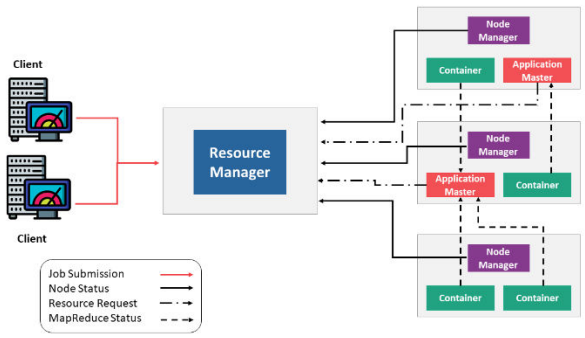
YARN enabled the users to perform operations as per requirement by using a variety of tools like ***Spark*** for real-time processing, ***Hive*** for SQL, ***HBase*** for NoSQL and others.

Apart from Resource Management, YARN also performs Job Scheduling. YARN performs all your processing activities by allocating resources and scheduling tasks. Apache Hadoop YARN Architecture consists of the following main components :

1. **Resource Manager:** Runs on a master daemon and manages the resource allocation in the cluster.
2. **Node Manager:** They run on the slave daemons and are responsible for the execution of a task on every single Data Node.
3. **Application Master:** Manages the user job lifecycle and resource needs of individual applications. It works along with the Node Manager and monitors the execution of tasks.
4. **Container:** Package of resources including RAM, CPU, Network, HDD etc on a single node.

Components of YARN

You can consider YARN as the brain of your Hadoop Ecosystem. The image below represents the YARN Architecture.



The **first component** of YARN Architecture is,

Resource Manager

- It is the ultimate authority in resource allocation.
- On receiving the processing requests, it passes parts of requests to corresponding node managers accordingly, where the actual processing takes place.
- It is the arbitrator of the cluster resources and decides the allocation of the available resources for competing applications.
- Optimizes the cluster utilization like keeping all resources in use all the time against various constraints such as capacity guarantees, fairness, and SLAs.
- It has two major components: a) Scheduler b) Application Manager

a) Scheduler

- The scheduler is responsible for allocating resources to the various running applications subject to constraints of capacities, queues etc.
- It is called a pure scheduler in ResourceManager, which means that it does not perform any monitoring or tracking of status for the applications.
- If there is an application failure or hardware failure, the Scheduler does not guarantee to restart the failed tasks.
- Performs scheduling based on the resource requirements of the applications.
- It has a pluggable policy plug-in, which is responsible for partitioning the cluster resources among the various applications. There are two such plug-ins: **Capacity Scheduler** and **Fair Scheduler**, which are currently used as Schedulers in ResourceManager.

b) Application Manager

- It is responsible for accepting job submissions.
- Negotiates the first container from the Resource Manager for executing the application specific Application Master.
- Manages running the Application Masters in a cluster and provides service for restarting the Application Master container on failure.

Coming to the **second component** which is :

Node Manager

- It takes care of individual nodes in a Hadoop cluster and manages user jobs and workflow on the given node.
- It registers with the Resource Manager and sends heartbeats with the health status of the node.
- Its primary goal is to manage application containers assigned to it by the resource manager.
- It keeps up-to-date with the Resource Manager.
- Application Master requests the assigned container from the Node Manager by sending it a Container Launch Context(CLC) which includes everything the application needs in order to run. The Node Manager creates the requested container process and starts it.
- Monitors resource usage (memory, CPU) of individual containers.
- Performs Log management.
- It also kills the container as directed by the Resource Manager.

The **third component** of Apache Hadoop YARN is,

Application Master

- An application is a single job submitted to the framework. Each such application has a unique Application Master associated with it which is a framework specific entity.
- It is the process that coordinates an application's execution in the cluster and also manages faults.
- Its task is to negotiate resources from the Resource Manager and work with the Node Manager to execute and monitor the component tasks.
- It is responsible for negotiating appropriate resource containers from the ResourceManager, tracking their status and monitoring progress.
- Once started, it periodically sends heartbeats to the Resource Manager to affirm its health and to update the record of its resource demands.

The **fourth component** is:

Container

- It is a collection of physical resources such as RAM, CPU cores, and disks on a single node.
- YARN containers are managed by a container launch context which is container life-cycle(CLC). This record contains a map of environment variables, dependencies stored in a remotely accessible storage, security tokens, payload for Node Manager services and the command necessary to create the process.
- It grants rights to an application to use a specific amount of resources (memory, CPU etc.) on a specific host.

Application Submission in YARN

Refer to the image and have a look at the steps involved in application submission of Hadoop YARN:

1) Submit the job

2) Get Application ID

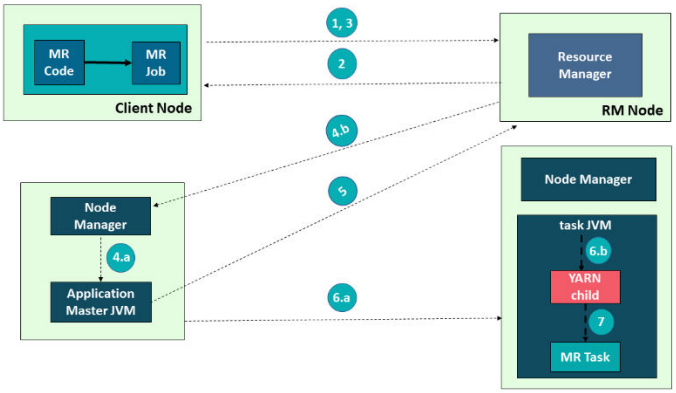
3) Application Submission Context

4 a) Start Container Launch
b) Launch Application Master

5) Allocate Resources

6 a) Container
b) Launch

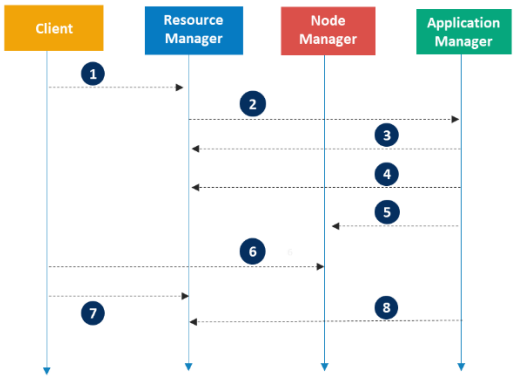
7) Execute



Application Workflow in Hadoop YARN

Refer to the given image and see the following steps involved in Application workflow of Apache Hadoop YARN:

1. Client submits an application
2. Resource Manager allocates a container to start Application Manager
3. Application Manager registers with Resource Manager
4. Application Manager asks containers from Resource Manager
5. Application Manager notifies 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. Application Manager unregisters with Resource Manager



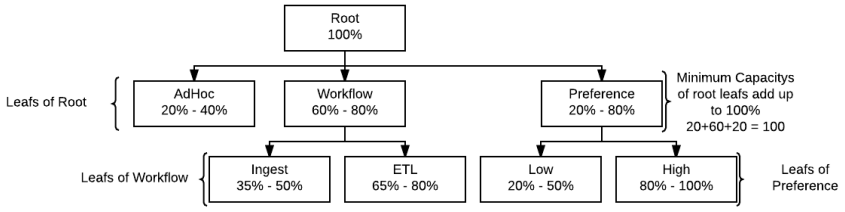
YARN – The Capacity Scheduler

Understanding the basic functions of the YARN Capacity Scheduler is a concept I deal with typically across all kinds of deployments. While Capacity Management has many facets from sharing, chargeback, and forecasting the focus of this blog will be on the primary features available for platform operators to use. In addition to the basic features some gotcha's will be reviewed that are commonly ran into when designing or utilizing the queues.

CAPACITY AND HIERARCHICAL DESIGN

YARN defines a minimum allocation and a maximum allocation for the resources it is scheduling for: Memory and/or Cores today. Each server running a worker for YARN has a NodeManager that is providing an allocation of resources which could be memory and/or cores that can be used for scheduling. This aggregate of resources from all Node Managers is provided as the 'root' of all resources the capacity scheduler has available.

The fundamental basics of the Capacity Scheduler are around how queues are laid out and resources are allocated to them. Queues are laid out in a hierarchical design with the topmost parent being the 'root' of the cluster queues, from here leaf (child) queues can be assigned from the root, or branches which can have leafs on themselves. Capacity is assigned to these queues as min and max percentages of the parent in the hierarchy. The minimum capacity is the amount of resources the queue should expect to have available to it if everything is running maxed out on the cluster. The maximum capacity is an elastic like capacity that allows queues to make use of resources which are not being used to fill minimum capacity demand in other queues.

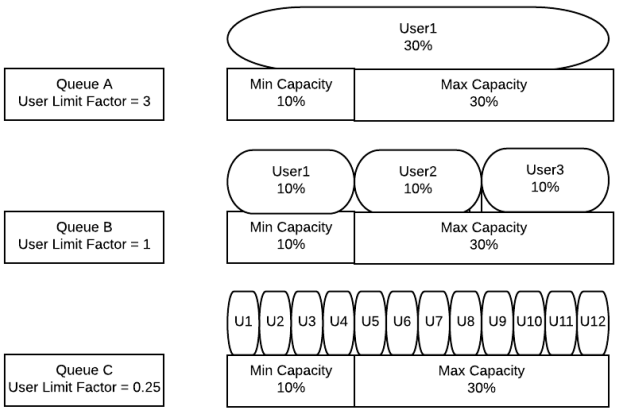


Children Queues like in the figure above inherit the resources of their parent queue. For example, with the Preference branch the Low leaf queue gets 20% of the Preference 20% minimum capacity while the High lead gets 80% of the 20% minimum capacity. Minimum Capacity always has to add up to 100% for all the leafs under a parent.

MINIMUM USER PERCENTAGE AND USER LIMIT FACTOR

Minimum User Percentage and User Limit Factor are ways to control how resources get assigned to users within the queues they are utilizing. The Min User Percentage is a soft limit on the smallest amount of resources a single user should get access to if they are requesting it. For example a min user percentage of 10% means that 10 users would each be getting 10% assuming they are all asking for it; this value is soft in the sense that if one of the users is asking for less we may place more users in queue.

User Limit Factor is a way to control the max amount of resources that a single user can consume. User Limit Factor is set as a multiple of the queues minimum capacity where a user limit factor of 1 means the user can consume the entire minimum capacity of the queue. If the user limit factor is greater than 1 it's possible for the user to grow into maximum capacity and if the value is set to less than 1, such as 0.5, a user will only be able to obtain half of the minimum capacity of the queue. Should you want a user to be able to also grow into the maximum capacity of a queue setting the value greater than 1 will allow the minimum capacity to be overtaken by a user that many times.



A common design point that may initially be non-intuitive is creation of queues by workloads and not by applications and then using the user-limit-factor to prevent individual takeover of queues by a single user by using a value of less than 1.0. This model supports simpler

operations by not allowing queue creation to spiral out of control by creating one per LoB but by creating queues by workloads to create predictable queue behaviors. Once a cluster is at full use and has applications waiting in line to run the lower user-limit-factors will be key to controlling resource sharing between tenants.

Initially this may not provide the cluster utilization at the start of their Hadoop platform journey due to using a smaller user limit to limit user resources, there are many approaches but one to consider is that initially it may be justified to allow a single tenant to take over a tiny cluster (say 10 nodes) but when expanding the platform footprint lower the user limit factor so that each tenant keeps the same amount of allocatable resources as before you added new nodes. This lets the user keep his initial experience, but be capped off from taking over the entire cluster as it grows making room for new users easily to get allocated capacity while not degrading the original user's experience.

ARCHETYPES

Designing queue archetypes to describe effective behavior of the tenant in the queue provides a means to measure changes against to see if they align or deviate from the expectation. While by no means a complete list of workload behaviors the below is a good list to start from. Create your own definitions of the types of queues your organization needs as the types of patterns applications and end users are performing.

- **AD-HOC**
  - This is where random user queries, unknown, and new workloads may be ran, there is no expectation as to resource allocation behaviors but can work as a good place to initially run applications to get a feel for per application tuning needs.
- **PREFERENCE**
  - These are applications that should get resources before and retain them longer. This could be for many reasons such as catch up applications, emergency runs, or other operational needs.
- **MACHINE LEARNING**
  - Machine Learning applications can be typically characterized by their long run times and large or intensive resource requirements. Termination of a task for some machine learning workloads can have long running impacts by greatly extending the duration
- **DASHBOARDING**
  - Low concurrency (refresh rate) but high number of queries per day. Dashboards need to be refreshed in an expected time but are very predictable workloads
- **EXPLORATION**
  - Exploration users have a need for low latency queries and need throughput to churn through very large datasets. Resources will likely be held for use the entire time the user is exploring to provide an interactive experience
- **BATCH WORKFLOW**
  - Designed to provide general computing needs for transformation and batch workloads. Setup is concerned most frequently with throughput of applications not individual application latency
- **ALWAYS ON**
  - Applications that are always running with no concept of completion. Applications that keep resources provisioned while waiting for new work to arrive. Slider deployed instances such as LLAP

CONTAINER CHURN

Churn within a queue is best described as a constant existing and starting of new containers. This behavior is very important to have a well behaving cluster were queues are quickly rebalanced to their minimum capacity and to balance capacity of the queue between its users fairly. The anti-pattern to churn is the long lived container that allocates itself and never releases as it can prevent proper rebalancing of resources in some cases it can completely block other applications from launching or queues from getting their capacity back. When not using preemption if a queue was to grow into elastic space but never release its containers the elastic space will never be given back to queues that have been guaranteed it. If a application running in your queues has long lived containers make special note and consider ways to mitigate its impact to other users with features like user-limit-factors, preemption, or even dedicated queues without elastic capacity.

FEATURES & BEHAVIORS OF THE CAPACITY SCHEDULER

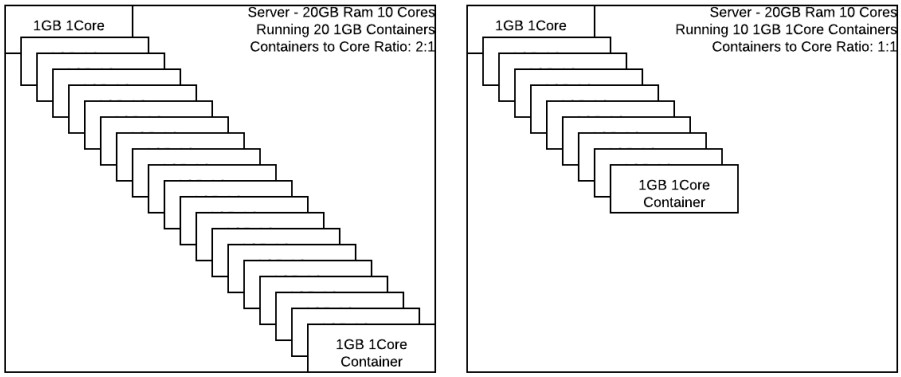
CPU SCHEDULING (DOMINANT RESOURCE FAIRNESS)

CPU Scheduling is not enabled by default and allows for the oversubscription of cores with no enforcement of use or preferred allocation taken into account. Many batch driven workloads may experience a throughput reduction if CPU scheduling is used, but may be required for strict SLA guarantees. A method called Dominant Resource Fairness (DRF) is used to decide what resource type to weight the utilization as: DRF takes the most used resource and treats you as using the highest percent for scheduling.
There are 2 primary parts to CPU Scheduling

1. Allocation and Placement
2. Enforcement

Allocation and placement is solved simply by enabling CPU Scheduling so that the Dominant Resource Fairness algorithm and VCores Node Manager's report begin getting used by the scheduler. This solves for some novel problems like an end user that used to schedule his

Spark applications with 1 or 2 executors but 8 cores per, and then all other tasks that ran on these nodes due to the excess of available memory were impacted. By enabling simple CPU Scheduling other tasks would no longer be provisioned onto the servers were all the cores are being utilized and find other prefered locations for task placement in the cluster.
Enforcement is solved by utilizing CGroups. This allows for YARN to ensure that if a task asked for a single vcore that they cannot utilize more than what was requested. Sharing of vcores when not used can be enabled or a strict enforcement of only what was scheduled can be done. The Node Manager's can also be configured as to what the max amount of CPU use on the server they will allow all tasks to sum up to, this allows for cores to be guaranteed to OS functions.



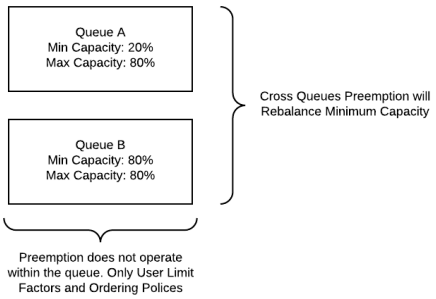
The above figure gives an idea of how many concurrent containers can change if limited to the smallest resource (typically CPU cores.) It's unlikely a true 1:1 Core to Container ratio is what will be required but this aspect of tuning is best left to monitoring historical system

metrics and then increasing or decrease the number of NodeManager VCores available for scheduler to allow more or less containers on a given server group.

PREEMPTION

When applications are making use of elastic capacity in their queues and another application comes along to demand back their minimum capacity (which is being used in another queue as elastic) traditionally the application would have to wait for the task to finish to get its resource allocations. With Preemption enabled resources in other queues can be reclaimed to provide the minimum capacity back to the queue that needs it. Preemption will try not to outright kill a application, and will take reducers last as they have to repeat more work then mappers if they have to re-run. From an ordering perspective Preemption looks at the youngest application and most over-subscribed ones first for task reclamation.

Preemption has some very specific behaviors and some of them don't function as expected for users. The most commonly expected behavior is for the queues to preempt within themselves to balance the resources over all users. That assumption is wrong as preemption only works across queues today, resources that are mis-balanced within a queue between users needs to look into other ways to control this such as User Limit Factors, Improved Queue Churned, and Queue FIFO/FAIR Policies.



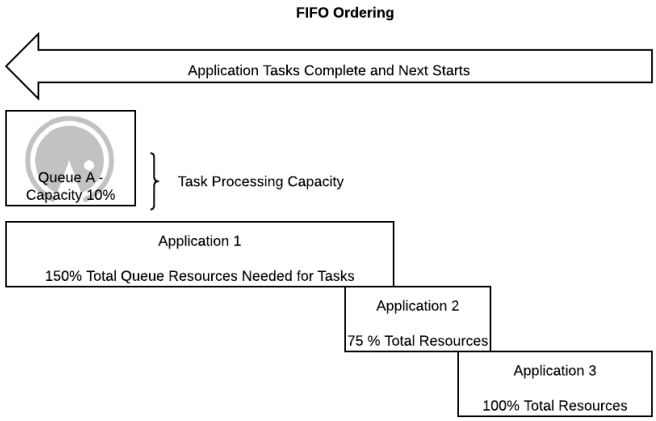
Another behavior is that Preemption will not preempt resources if it cannot provide enough to fulfill another resource allocation request. While typically this is not an issue with large cluster for small clusters with large maximum container sizes could run into a scenario were Preemption is not configured to reclaim a container of the largest possible size and therefore will do nothing at all. The main properties used to control this behavior is the Total Preemption Per Round and the Natural Termination Factor. Total Preemption Per Round is the percentage of resources on the cluster that can be preempted at once, the Natural Termination Factor is the percent of resources out of the total cluster (100%) requested that will be preempted up to the Total Preemption Per Round.

The last misconception is that Preemption will rebalance max (elastic) capacity use for between queues that wish to grow into it. Maximum Capacity is given on a first come first serve basis only. If a single queue has taken over all the of the clusters capacity, and another application start in a queue that needs its minimum capacity back, only the minimum capacity will be preempted and all of the maximum capacity being used by the other queue will remain until the containers churn naturally.

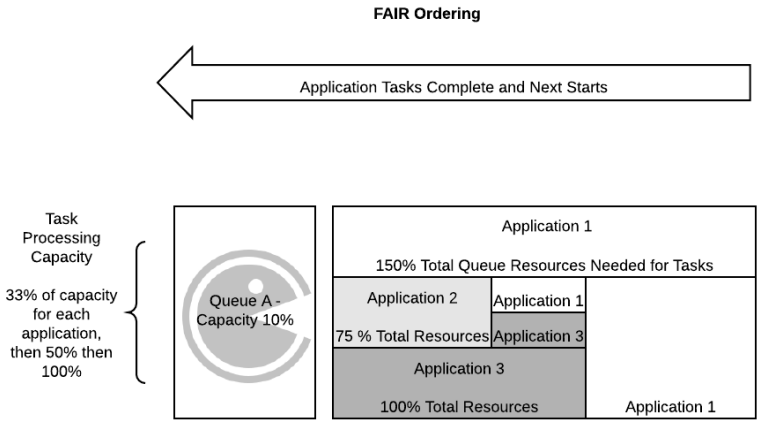
QUEUE ORDERING POLICIES

There are currently two types of ordering policies the Capacity Scheduler supports today: FIFO and FAIR. The default that queues start with is FIFO which in my experience is not the behavior that clients expect from their queues. By configuring between FIFO and FAIR at the Queue Leaf level you can create behaviors that lead to throughput driven processing or shared fair processing between applications running. One important thing to know about ordering policies is that they operated at the application level in the queue and do not care about which user owns the applications.

With the FIFO policy applications are evaluated in order of oldest to youngest for resource allocation. If an application has outstanding resource requests they are immediately fulfilled first come first serve. The result of this is that should an application have enough outstanding requests that they would consume the entire queue multiple times before completing they will block out other applications from starting while they are first allocated resources as the oldest application. It is this very behavior that comes most unexpected to users and creates the most discontent as a users can even block their own applications with their own applications!



That behavior is easily solved for today by using the FAIR ordering policy. When using the FAIR ordering policy on a leaf queue applications are evaluated for resource allocation requests by first the application using the least resources first and most last. This way new applications entering the queue who have no resources for processing will be first asked for their required allocations to get started. Once all applications in the queue have resources they get balanced fairly between all users asking for them.



It's important to note that this behavior only occurs if you have good container churn in your queue. Because Preemption does not exist inside of a queue resources cannot be forcefully redistributed within it and the FAIR ordering policy only is concerned with new allocations of resources not current ones; what does this mean? That if the queue is currently utilized with tasks which never complete or run for long

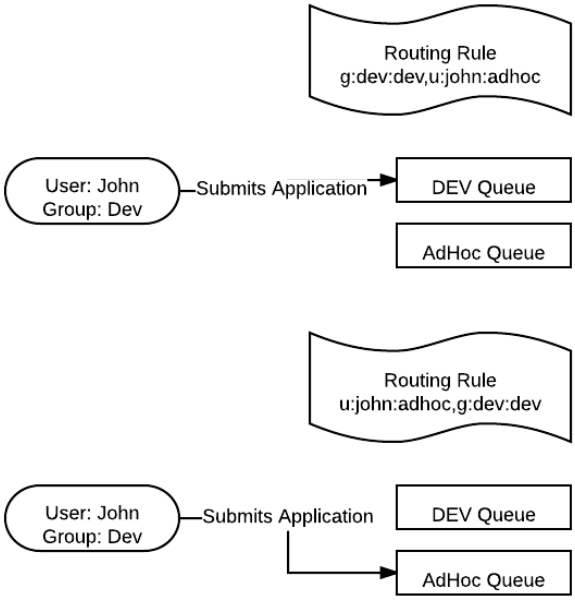
periods of time without allowing for container churn to occur in the queue will hold the resources and still prevent applications from executing.

USERNAME AND APPLICATION DRIVEN CALCULATIONS

Calculations in the Capacity Scheduler look at two primary attributes when attempting to provide allocations: User Name and Application ID. When it comes to sharing resources between users in a queue aspects like Minimum User Percentage and User Limit Factor both look at the User Name itself; this can obviously cause some conflicting problems if you're using a service account for multiple users to run jobs as only 1 user will appear to the Capacity Scheduler. Within a Queue which application gets resource allocations are driven by the leaf queues ordering policies: FIFO or FAIR which only care about the application and not which user is running it. In FIFO resources are allocated first to the oldest application in the queue and only when it no longer requires any will the next application gets an allocation. With FAIR applications that are using the least about of resources are first asking if there are pending allocations for them and fulfilled if so, if not the next application with least resources is checked; this helps to evenly share queues with applications that would normally consume them.

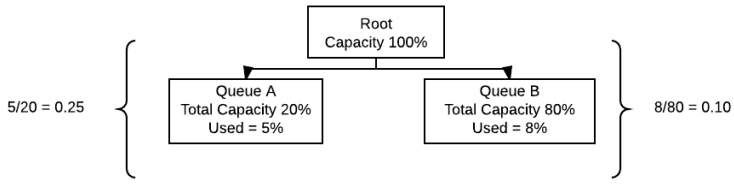
DEFAULT QUEUE MAPPING

Normally to target a specific queue the user provides some configuration information telling the client tools what queue to request. But commonly users utilize tools that have a hard time passing configurations downstream to target specific queues. With Default Queue Mappings we can route an entity by its username or groups it belongs to into specific queues. Take note that the default queue routing configuration matches whichever routing attribute comes first. So if a group mapping is provided before the user mapping that matches the user he is routed to the queue for the group.



PRIORITY

When resources are allocated over multiple queues the one with the lowest relative capacity gets resources first. If you're in a scenario where you want to have a high priority queue that receives resources before others then changing to a higher priority is a simple way to do this. Today using queue priorities with LLAP and Tez allows for more interactive workloads as these queues can be assigned resources at a higher priority to reduce the end latency that an end user may experience.

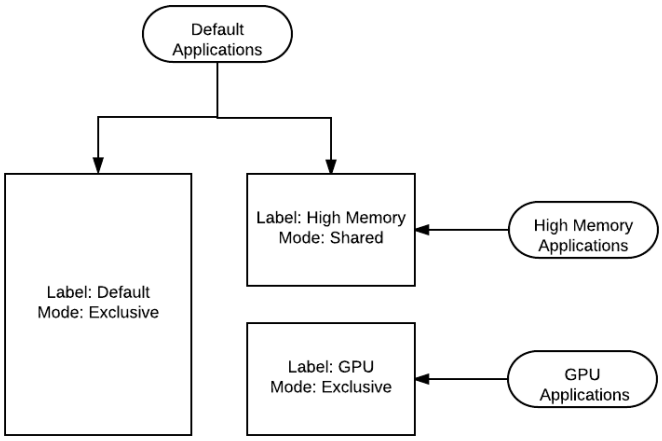


Above the two queues provide an example of how relative capacity is used unmodified by queue priorities. In this case even while Queue A is smaller than Queue B and while Queue B is using more absolute resources it is elected to continue receiving them first because its relative capacity is lower than that of Queue A. If the desired behavior required Queue A to always receive resource allocations first the Queue priority should be increased above that of Queue B. When assigning priority a higher value represents higher priority.

LABELS

Labels are better describe as partitions of the cluster. All clusters start with a default label or partition that is exclusive in the sense that as new labeled partitions are added to the cluster they are not shared with the original default cluster partition. Labels be defined as exclusive or shared when created and a node can only have a single label assigned to it. More common uses of labels is for the targeting of GPU hardware in the cluster or to deploy licensed software against only a specific subset of the cluster. Today LLAP also uses Labels to leverage dedicated hosts for long running processes.

Shared labels allow applications in other labeled partitions such as the default cluster to grow into them and utilize the hardware if no specific applications are asking for the label. If an application comes along that targets the label specifically the other applications that were utilizing it will be preempted off the labeled node so the application needing it can make use of it. Exclusive labels are just that, exclusive and will not be shared by any others; Only applications specifically targeting labels will run solely on them. Access to labeled partitions is provided to leaf queues so users able to submit to them are able to target the label. If you wish to autoroute users to labels say creating a GPU queue that automatically uses the GPU label this is possible.



QUEUE NAMES

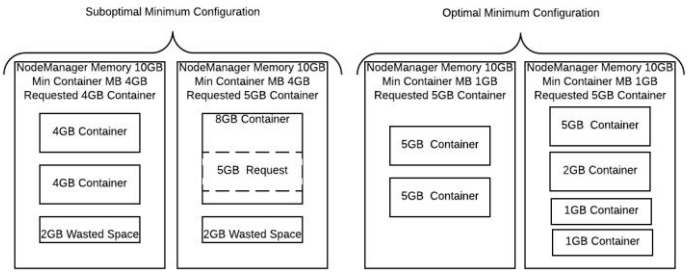
Queue leaf names have to be unique with the Capacity Scheduler. For example if you created a queue in the capacity scheduler as root.adhoc.dev dev will have to be unique as a leaf over all queue names and you cannot have a root.workflow.dev queue as it would no longer be unique. This is in line with how queues are specified for submission by only using the leaf name and not the entire composite queue name. Parents of leafs are never submitted to directly and have no need to be unique so you can have root.adhoc.dev and root.adhoc.qa without issue as both dev and qa are unique leaf names.

LIMITING APPLICATIONS PER QUEUE

Spawning of a queue by launching many applications into it so that none can effectively complete can create bottlenecks and impact SLAs. At the very worst the entire queue becomes deadlocked and nothing is able to process without and admin physically killing jobs to free up resources for compute tasks. This is easily prevented by placing a limit on the total number of applications allowed to run in the leaf queue, alternatively it's possible to control the % of resources of the leaf that can be used by Application Masters. By default this value is typically rather large over 10,000 applications (or 20% of leaf resources) and can be configured per leaf if needed otherwise the value is inherited from prior parent queues of the leaf.

CONTAINER SIZING

An unknown to many people utilizing the Capacity Scheduler is that containers are multiples in size of the minimum allocation. For example if your minimum scheduler mb of memory per container was 1gb and you requested a 4.5gb sized container the scheduler will round this request up to 5gb. With very high minimums this can create large resource wastage problems for example with a 4gb minimum if we requested 5gb we would be served 8gb providing us with 3 extra GB that we never even planned on using! When configuring and minimum and maximum container size the maximum should be evenly divisible by the minimum.



Fair Scheduling?

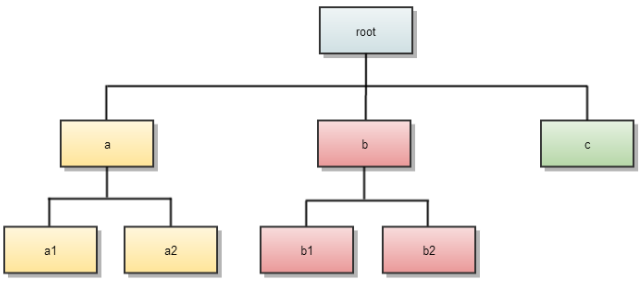
Fair scheduling is a method of assigning resources to applications such that all applications get, on average, an equal share of resources over time. Hadoop 2.x is capable of scheduling multiple resource types.

By default, the Fair Scheduler bases scheduling fairness decisions only on memory. It can be configured to schedule resources based on memory and CPU. When only one application is running, that application uses the entire cluster. When other applications are submitted, resources that free up are assigned to the new applications, so that each application eventually gets approximately the same amount of resources. Unlike the default Hadoop scheduler, which forms a queue of applications, this lets short applications finish in reasonable time while not starving long-lived applications. It is also a reasonable way to share a cluster between a number of users. Finally, fair sharing also uses priorities applied as weights to determine the fraction of total resources that each application should get.

Scheduling Queues

The scheduler organizes applications further into *queues*, and shares resources fairly between these queues. By default, all users share a single queue, named default. If an application specifically lists a queue in a container resource request, the request is submitted to that queue. You can also assign queues based on the user name included with the request through configuration. Within each queue, a scheduling policy is used to share resources between the running applications. The default is memory-based fair sharing, but FIFO and multi-resource with Dominant Resource Fairness can also be configured.

Queues can be arranged in a hierarchy to divide resources, and they can be configured with weights to share the cluster in specific proportions. The Fair Scheduler uses a concept called a *queue path* to configure a hierarchy of queues. The queue path is the full path of the queue's hierarchy, starting at *root*. The following example has three top-level child-queues a, b, and c and some sub-queues for a and b:

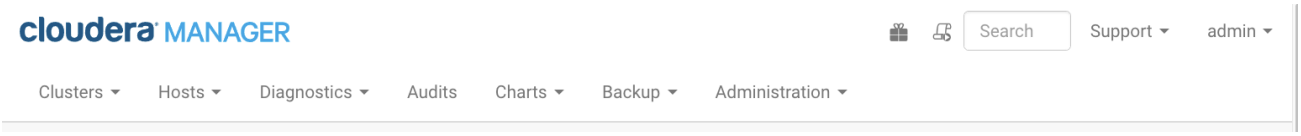


In addition to providing fair sharing, the Fair Scheduler allows assigning guaranteed minimum shares to queues, which is useful for ensuring that certain users, groups or production applications always get sufficient resources. When a queue contains apps, it gets at least its minimum share, but when the queue does not need its full guaranteed share, the excess is split between other running apps. This lets the scheduler guarantee capacity for queues while utilizing resources efficiently when these queues do not contain applications.

Cloudera Manager Admin Console

Cloudera Manager Admin Console is the web-based UI that you use to configure, manage, and monitor CDH.

If no services are configured when you log into the Cloudera Manager Admin Console, the Cloudera Manager installation wizard displays. If services have been configured, the Cloudera Manager top navigation bar:



- **Clusters > *cluster_name***
  - **Services** - Display individual services, and the Cloudera Management Service. In these pages you can:
    - View the status and other details of a service instance or the role instances associated with the service
    - Make configuration changes to a service instance, a role, or a specific role instance
    - Add and delete a service or role
    - Stop, start, or restart a service or role.
    - View the commands that have been run for a service or a role
    - View an audit event history
    - Deploy and download client configurations
    - Decommission and recommission role instances
    - Enter or exit maintenance mode
    - Perform actions unique to a specific type of service. For example:
      - Enable HDFS high availability or NameNode federation
      - Run the HDFS Balancer
      - Create HBase, Hive, and Sqoop directories

- **Cloudera Manager Management Service** - Manage and monitor the Cloudera Manager Management Service. This includes the following roles: Activity Monitor, Alert Publisher, Event Server, Host Monitor, Navigator Audit Server, Navigator Metadata Server, Reports Manager, and Service Monitor.
- **Cloudera Navigator** - Opens the Cloudera Navigator user interface.
- **Hosts** - Displays the hosts in the cluster.
- **Reports** - Create reports about the HDFS, MapReduce, YARN, and Impala usage and browse HDFS files, and manage quotas for HDFS directories.
- **Utilization Report** - Opens the **Cluster Utilization Report.** displays aggregated utilization information for YARN and Impala jobs.
- ***MapReduce_service_name* Jobs** - Query information about MapReduce jobs running on your cluster.
- ***YARN_service_name* Applications** - Query information about YARN applications running on your cluster.
- ***Impala_service_name* Queries** - Query information about Impala queries running on your cluster.
- **Dynamic Resource Pools** - Manage dynamic allocation of cluster resources to YARN and Impala services by specifying the relative weights of named pools.
- **Static Service Pools** - Manage static allocation of cluster resources to HBase, HDFS, Impala, MapReduce, and YARN services.

- **Hosts - Display the hosts managed by Cloudera Manager.**
  - **All Hosts** - Displays a list of manage hosts in the cluster.
  - **Roles** - Displays the roles deployed on each host.
  - **Host Templates** - Create and manage **Host Templates**, which define sets of role groups that can be used to easily expand a cluster.
  - **Disks Overview** - Displays the status of all disks in the cluster.
  - **Parcels** - Displays parcels available in the cluster and allows you to download, distribute, and activate new parcels.
    In this page you can:
  - View the status and a variety of detail metrics about individual hosts
  - Make configuration changes for host monitoring
  - View all the processes running on a host
  - Run the Host Inspector
  - Add and delete hosts
  - Create and manage host templates
  - Manage parcels
  - Decommission and recommission hosts
  - Make rack assignments
  - Run the host upgrade wizard

- **Diagnostics - Review logs, events, and alerts to diagnose problems. The subpages are:**
  - **Events** - Search for and displaying events and alerts that have occurred.
    - **Logs** - Search logs by service, role, host, and search phrase as well as log level (severity).
    - **Server Log** -Display the Cloudera Manager Server log.

- **Audits - Query and filter audit events across clusters, including logins, across clusters.**
- **Charts** - Query for metrics of interest, display them as charts, and display personalized chart dashboards.
- **Backup** - Manage replication schedules and snapshot policies.

- **Administration - Administer Cloudera Manager. The subpages are:**
  - **Settings** - Configure Cloudera Manager.
  - **Alerts** - Display when alerts will be generated, configure alert recipients, and send test alert email.
  - **Users** - Manage Cloudera Manager users and user sessions.
  - **Security** - Generate Kerberos credentials and inspect hosts.
  - **License** - Manage Cloudera licenses.
  - **Language** - Set the language used for the content of activity events, health events, and alert email messages.
  - **AWS Credentials** - Configure S3 connectivity to Cloudera Manager.
  - **Parcel** Icon - link to the Hosts > Parcels page.
  - **Running Commands Indicator** - displays the number of commands currently running for all services or roles.
  - **Search** - Supports searching for services, roles, hosts, configuration properties, and commands. You can enter a partial string and a drop-down list with up to sixteen entities that match will display.

- **Support - Displays various support actions. The subcommands are:**
  - **Send Diagnostic Data** - Sends data to Cloudera Support to support troubleshooting.
  - **Support Portal (Cloudera Enterprise)** - Displays the Cloudera Support portal.
  - **Mailing List (Cloudera Express)** - Displays the Cloudera Manager Users list.
  - **Scheduled Diagnostics: Weekly** - Configure the frequency of automatically collecting diagnostic data and sending to Cloudera support.

- **Help**
- **Installation Guide**
- **API Documentation**
- **Release Notes**
  - **About** - Version number and build details of Cloudera Manager and the current date and time stamp of the Cloudera Manager server.
- **Logged-in User Menu** - The currently logged-in user. The subcommands are:
  - **Change Password** - Change the password of the currently logged in user.
  - **Logout**

Starting and Logging into the Admin Console

1. In a web browser, enter `http://`*Server host*`:7180`, where *Server host* is the FQDN or IP address of the host where the Cloudera Manager Server is running.
   The login screen for Cloudera Manager Admin Console displays.
2. Log into Cloudera Manager Admin Console using the credentials assigned by your administrator. User accounts are assigned roles that constrain the features available to you.

Displaying the Cloudera Manager Server Version and Server Time

To display the version, build number, and time for the Cloudera Manager Server:

1. Open the Cloudera Manager Admin Console.
2. Select **Support** > **About**.

Cloudera Manager Admin Console Home Page

When you start the Cloudera Manager Admin Console, the **Home** > **Status** tab displays.
You can also go to the **Home** > **Status** tab by clicking the Cloudera Manager logo in the top navigation bar.

Status

The Status tab contains:

- **Clusters** - The clusters being managed by Cloudera Manager. Each cluster is displayed either in summary form or in full form depending on the configuration of the **Administration > Settings > Other > Maximum Cluster Count Shown In Full** property. When the number of clusters exceeds the value of the property, only cluster summary information displays.
  - **Summary Form** - A list of links to cluster status pages. Click **Customize** to jump to the **Administration> Settings > Other > Maximum Cluster Count Shown In Full** property.
  - **Full Form** - A separate section for each cluster containing a link to the cluster status page and a table containing links to the Hosts page and the status pages of the services running in the cluster.
  - **Cloudera Management Service** - A table containing a link to the Cloudera Manager Service. The Cloudera Manager Service has a menu of actions that you select by clicking  .
  - **Charts** - A set of charts (dashboard) that summarize resource utilization (IO, CPU usage) and processing metrics.
    Click a line, stack area, scatter, or bar chart to expand it into a full-page view with a legend for the individual charted entities as well more fine-grained axes divisions.
    By default the time scale of a dashboard is 30 minutes. To change the time scale, click a duration link 

    at the top-right of the dashboard.
    To set the dashboard type, click  and select one of the following:

- **Custom** - displays a custom dashboard.
- **Default** - displays a default dashboard.
- **Reset** - resets the custom dashboard to the predefined set of charts, discarding any customizations.

All Health Issues

Displays all health issues by cluster. The number badge has the same semantics as the per service health issues reported on the Status tab.

- By default only Bad health test results are shown in the dialog box. To display Concerning health test results, click the **Also show *n* concerning issue(s)** link.
- To group the health test results by entity or health test, click the buttons on the **Organize by Entity/Organize by Health Test** switch.
- Click the link to display the Status page containing with details about the health test result.

All Configuration Issues

Displays all configuration issues by cluster. The number badge has the same semantics as the per service configuration issues reported on the Status tab. By default only notifications at the Error severity level are listed, grouped by service name are shown in the dialog box. To display Warning notifications, click the **Also show *n* warning(s)** link. Click the message associated with an error or warning to be taken to the configuration property for which the notification has been issued where you can address the issue.

All Recent Commands

Displays all commands run recently across the clusters. A badge 

indicates how many recent commands are still running. Click the command link to display details about the command and child commands

Viewing Running and Recent Commands For a Cluster

The indicator positioned just to the left of the Search field on the right side of the Admin Console main navigation bar displays the number of commands currently running for all services or roles. To display the running commands, click the indicator.

To display all commands that have run and finished recently, do one of the following:

- Click the **All Recent Commands** button in the window that pops up when you click the indicator. This command displays information on all running and recent commands in the same form, as described below.
- Click the Cloudera Manager logo in the main navigation bar and click the **All Recent Commands** tab.

Select a value from the pager 

to control how many commands are listed, or click the arrows to view pages.

The command indicator shows the number of commands running on all clusters you are managing. Likewise, **All Recent Commands** shows all commands that were run and finished within the search time range you specified, across all your managed clusters.

Viewing Running and Recent Commands for a Service or Role

For a selected service or role instance, the **Commands** tab shows which commands are running or have been run for that instance, and what the status, progress, and results are. For example, if you go to the HDFS service shortly after you have installed your cluster and look at the **Commands** tab, you will see recent commands that created the directories, started the HDFS role instances (the NameNode, Secondary NameNode, and DataNode instances), and the command that initially formatted HDFS on the NameNode. This information is useful if a service or role seems to be taking a long time to start up or shut down, or if services or roles are not running or do not appear to have been started correctly. You can view both the status and progress of currently running commands, as well as the status and results of commands run in the past.

1. Click the **Clusters** tab on the top navigation bar.
2. Click the service name to go to the Status tab for that service.
3. For a role instance, click the **Instances** tab and select the role instance name to go to its Status tab.
4. Click the **Commands** tab.

Viewing Service Instance Details

1. Do one of the following:
   - In the **Home** > **Status** tab, if the cluster is displayed in full form, click *ServiceName* in a *ClusterName*table.
   - In the **Home** > **Status** tab, click *ClusterName* and then click *ServiceName*.
   - Select **Clusters** > ***ClusterName*** > ***ServiceName***.

2. Click the **Instances** tab on the service's navigation bar. This shows all instances of all role types configured for the selected service.

You can also go directly to the Instances page to view instances of a specific role type by clicking one of the links under the **Role Counts** column. This will show only instances of the role type you selected.

The Instances page displays the results of the configuration validation checks it performs for all the role instances for this service.

Note: The information on this page is always the **Current** information for the selected service and roles. This page does not support a historical view: thus, the Time Range Selector is not available.

The information on this page shows:

- The name of the role instance. Click the name to view the role status for that role.
- The host on which it is running. Click the hostname to view the host status details for the host.
- The rack assignment.
- The status. A single value summarizing the state and health of the role instance.
- Whether the role is currently in maintenance mode. If the role has been set into maintenance mode explicitly, you will see the following icon (). If it is in effective maintenance mode due to the service or its host having been set into maintenance mode, the icon will be this ().
- Whether the role is currently decommissioned.

You can sort or filter the Instances list by criteria in any of the displayed columns:

- **Sort**

1. Click the column header by which you want to sort. A small arrow indicates whether the sort is in ascending or descending order.
2. Click the column header again to reverse the sort order.

- **Filter** - Type a property value in the Search box or select the value from the facets at the left of the page.

Host Details

You can view detailed information about each host, including:

- Name, IP address, rack ID
- Health status of the host and last time the Cloudera Manager Agent sent a heartbeat to the Cloudera Manager Server
- Number of cores
- System load averages for the past 1, 5, and 15 minutes
- Memory usage
- File system disks, their mount points, and usage
- Health test results for the host
- Charts showing a variety of metrics and health test results over time.
- Role instances running on the host and their health
- CPU, memory, and disk resources used for each role instance

To view detailed host information:

1. Click the **Hosts** tab.
2. Click the name of one of the hosts. The Status page is displayed for the host you selected.
3. Click tabs to access specific categories of information. Each tab provides various categories of information about the host, its services, components, and configuration.

Status

The Status page is displayed when a host is initially selected and provides summary information about the status of the selected host. Use this page to gain a general understanding of work being done by the system, the configuration, and health status.
If this host has been decommissioned or is in maintenance mode, you will see the following icon(s) (,) in the top bar of the page next to the status message.

Details

This panel provides basic system configuration such as the host's IP address, rack, health status summary, and disk and CPU resources. This information summarizes much of the detailed information provided in other panes on this tab. To view details about the Host agent, click the Host Agent link in the Details section.

Health Tests

Cloudera Manager monitors a variety of metrics that are used to indicate whether a host is functioning as expected. The Health Tests panel shows health test results in an expandable/collapsible list, typically with the specific metrics that the test returned. (You can Expand All or Collapse All from the links at the upper right of the Health Tests panel).

- The color of the text (and the background color of the field) for a health test result indicates the status of the results. The tests are sorted by their health status – Good, Concerning, Bad, or Disabled. The list of entries for good and disabled health tests are collapsed by default; however, Bad or Concerning results are shown expanded.
- The text of a health test also acts as a link to further information about the test. Clicking the text will pop up a window with further information, such as the meaning of the test and its possible results, suggestions for actions you can take or how to make configuration changes related to the test. The help text for a health test also provides a link to the relevant monitoring configuration section for the service. See Configuring Monitoring Settings for more information.

Health History

The Health History provides a record of state transitions of the health tests for the host.

- Click the arrow symbol at the left to view the description of the health test state change.
- Click the **View** link to open a new page that shows the state of the host at the time of the transition. In this view some of the status settings are greyed out, as they reflect a time in the past, not the current status.

File Systems

The File systems panel provides information about disks, their mount points and usage. Use this information to determine if additional disk space is required.

Roles

Use the Roles panel to see the role instances running on the selected host, as well as each instance's status and health. Hosts are configured with one or more role instances, each of which corresponds to a service. The role indicates which daemon runs on the host. Some examples of roles include the NameNode, Secondary NameNode, Balancer, JobTrackers, DataNodes, RegionServers and so on. Typically a host will run multiple roles in support of the various services running in the cluster.

Clicking the role name takes you to the role instance's status page.

You can delete a role from the host from the Instances tab of the Service page for the parent service of the role. You can add a role to a host in the same way. See Role Instances.

Charts

Charts are shown for each host instance in your cluster.

See Viewing Charts for Cluster, Service, Role, and Host Instances for detailed information on the charts that are presented, and the ability to search and display metrics of your choice.

Processes

The Processes page provides information about each of the processes that are currently running on this host. Use this page to access management web UIs, check process status, and access log information.

Note: The Processes page may display exited startup processes. Such processes are cleaned up within a day.

The Processes tab includes a variety of categories of information.

- **Service** - The name of the service. Clicking the service name takes you to the service status page. Using the triangle to the right of the service name, you can directly access the tabs on the role page (such as the Instances, Commands, Configuration, Audits, or Charts Library tabs).
- **Instance** - The role instance on this host that is associated with the service. Clicking the role name takes you to the role instance's status page. Using the triangle to the right of the role name, you can directly access the tabs on the role page (such as the Processes, Commands, Configuration, Audits, or Charts Library tabs) as well as the status page for the parent service of the role.
- **Name** - The process name.
- **Links** - Links to management interfaces for this role instance on this system. These is not available in all cases.
- **Status** - The current status for the process. Statuses include stopped, starting, running, and paused.
- **PID** - The unique process identifier.
- **Uptime** - The length of time this process has been running.
- **Full log file** - A link to the full log (a file external to Cloudera Manager) for this host log entries for this host.
- **Stderr** - A link to the stderr log (a file external to Cloudera Manager) for this host.
- **Stdout** - A link to the stdout log (a file external to Cloudera Manager) for this host.

Resources

The Resources page provides information about the resources (CPU, memory, disk, and ports) used by every service and role instance running on the selected host.

Each entry on this page lists:

- The service name
- The name of the particular instance of this service
- A brief description of the resource
- The amount of the resource being consumed or the settings for the resource

The resource information provided depends on the type of resource:

- **CPU** - An approximate percentage of the CPU resource consumed.
- **Memory** - The number of bytes consumed.
- **Disk** - The disk location where this service stores information.
- **Ports** - The port number being used by the service to establish network connections.

Commands

The Commands page shows you running or recent commands for the host you are viewing. See Viewing Running and Recent Commands for more information.

Configuration

Minimum Required Role: Full Administrator

The Configuration page for a host lets you set properties for the selected host. You can set properties in the following categories:

- **Advanced** - Advanced configuration properties. These include the Java Home Directory, which explicitly sets the value of `JAVA_HOME` for all processes. This overrides the auto-detection logic that is normally used.
- **Monitoring** - Monitoring properties for this host. The monitoring settings you make on this page will override the global host monitoring settings you make on the Configuration tab of the Hosts page. You can configure monitoring properties for:
  - health check thresholds
  - the amount of free space on the filesystem containing the Cloudera Manager Agent's log and process directories
  - a variety of conditions related to memory usage and other properties
  - alerts for health check events

  For some monitoring properties, you can set thresholds as either a percentage or an absolute value (in bytes).
- **Other** - Other configuration properties.
- **Parcels** - Configuration properties related to parcels. Includes the **Parcel Director** property, the directory that parcels will be installed into on this host. If the `parcel_dir` variable is set in the Agent's `config.ini` file, it will override this value.
- **Resource Management** - Enables resource management using control groups (cgroups).

For more information, see the description for each or property or see Modifying Configuration Properties Using Cloudera Manager.

Components

The Components page lists every component installed on this host. This may include components that have been installed but have not been added as a service (such as YARN, Flume, or Impala).

This includes the following information:

- **Component** - The name of the component.
- **Version** - The version of CDH from which each component came.

- **Component Version** - The detailed version number for each component.

Audits

The Audits page lets you filter for audit events related to this host. See Lifecycle and Security Auditing for more information.

Charts Library

The Charts Library page for a host instance provides charts for all metrics kept for that host instance, organized by category. Each category is collapsible/expandable

Host Inspector

You can use the host inspector to gather information about hosts that Cloudera Manager is currently managing. You can review this information to better understand system status and troubleshoot any existing issues. For example, you might use this information to investigate potential DNS misconfiguration.

The inspector runs tests to gather information for functional areas including:

- Networking
- System time
- User and group configuration
- HDFS settings
- Component versions

Common cases in which this information is useful include:

- Installing components
- Upgrading components
- Adding hosts to a cluster
- Removing hosts from a cluster

Running the Host Inspector

1. Click the **Hosts** tab and select **All Hosts**.
2. Click the **Inspect All Hosts** button. Cloudera Manager begins several tasks to inspect the managed hosts.
3. After the inspection completes, click **Download Result Data** or **Show Inspector Results** to review the results.

The results of the inspection displays a list of all the validations and their results, and a summary of all the components installed on your managed hosts.

If the validation process finds problems, the **Validations** section will indicate the problem. In some cases the message may indicate actions you can take to resolve the problem. If an issue exists on multiple hosts, you may be able to view the list of occurrences by clicking a small triangle that appears at the end of the message.

The **Version Summary** section shows all the components that are available from Cloudera, their versions (if known) and the CDH distribution to which they belong (CDH 3 or CDH 4). If you are running CDH 3, the Version will be listed as "Unavailable". Version identification is not available with CDH 3. In a CDH 3 cluster, CDH 4 components will be listed as "Not installed or path incorrect".

If you are running multiple clusters with both CDH 3 and CDH 4, the lists will be organized by distribution (CDH 3 or CDH 4). The hosts running that version are shown at the top of each list.

Viewing Past Host Inspector Results

You can view the results of a past host inspection by looking for the Host Inspector command using the **Recent Commands** feature.

1. Click the Running Commands indicator () located just to the left of the Search box at the right hand side of the navigation bar.
2. Click the **Recent Commands** button.
3. If the command is too far in the past, you can use the Time Range Selector to move the time range back to cover the time period you want.
4. When you find the Host Inspector command, click its name to display its sub-commands.
5. Click the **Show Inspector Results** button to view the report.

Cloudera Installation:

Pre-check:

Step 1:
Host file update
/ets/hosts

Step 2:
Update hostname in /etc/sysconfig/network
[root@nn1 ~]# cat /etc/sysconfig/network
NETWORKING=yes
HOSTNAME=nn1.ggvuyiseqaae5c4q5oieovosdh.bx.internal.cloudapp.net

Step 3:
vi /etc/sysconfig/selinux
SELINX=disabled
getenforce

Step 4:
Stop firewall
systemctl stop firewalld.service
systemctl disable firewalld.service

Step 5:
echo 1 > /proc/sys/vm/swappiness

Step 6:
Disable Transparent Hugepages,
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag

Step 7:
Disable the tuned Service
1. Ensure that the tuned service is started:
systemctl start tuned

2. Turn the tuned service off:
tuned-adm off

3. Ensure that there are no active profiles:
tuned-adm list

The output should contain the following line:
No current active profile

4. Shutdown and disable the tuned service:
systemctl stop tuned
systemctl disable tuned

5. Reboot nodes

Install and Configure External Databases

Cloudera Manager uses various databases and datastores to store information about the Cloudera Manager configuration, as well as information such as the health of the system or task progress. For quick, simple installations, Cloudera Manager can install and configure an embedded PostgreSQL database as part of the Cloudera Manager installation process. In addition, some CDH services use databases and are automatically configured to use a default database. If you plan to use the embedded and default databases provided during the Cloudera Manager installation

Required Databases

- **Oozie Server** - Contains Oozie workflow, coordinator, and bundle data. Can grow very large.
- **Sqoop Server** - Contains entities such as the connector, driver, links and jobs. Relatively small.
- **Activity Monitor** - Contains information about past activities. In large clusters, this database can grow large. Configuring an Activity Monitor database is only necessary if a MapReduce service is deployed.
- **Reports Manager** - Tracks disk utilization and processing activities over time. Medium-sized.
- **Hive Metastore Server** - Contains Hive metadata. Relatively small.
- **Hue Server** - Contains user account information, job submissions, and Hive queries. Relatively small.
- **Sentry Server** - Contains authorization metadata. Relatively small.
- **Cloudera Navigator Audit Server** - Contains auditing information. In large clusters, this database can grow large.
- **Cloudera Navigator Metadata Server** - Contains authorization, policies, and audit report metadata. Relatively small.

a.) Download and copy MySQL

tar files to home folder using WINSCP.

https://dev.mysql.com/downloads/file/?id=471503

wget https://dev.mysql.com/get/archives/mysql-5.7/mysql-5.7.19-1.el7.x86_64.rpm-bundle.tar

b.) Extract files downloaded.

tar xf mysql-5.7.19-1.el7.x86_64.rpm-bundle.tar

c.) Install components.

rpm -ivh mysql-community-common-5.7.19-1.el7.x86_64.rpm

rpm -ivh mysql-community-libs-5.7.19-1.el7.x86_64.rpm

rpm -ivh mysql-community-libs-compat-5.7.19-1.el7.x86_64.rpm

rpm -ivh mysql-community-devel-5.7.19-1.el7.x86_64.rpm

rpm -ivh mysql-community-client-5.7.19-1.el7.x86_64.rpm

rpm -ivh mysql-community-server-5.7.19-1.el7.x86_64.rpm

d.) Start MySQL server.

$ systemctl start mysqld

e.) Stop the MySQL server if it is running.

$ systemctl stop mysqld

f.) Ensure the MySQL server starts at boot.

$ /sbin/chkconfig mysqld on

g.) Start the MySQL server if it is running.

$ systemctl start mysqld

h.) Get the temporary password for root user.

$ cat /etc/my.cnf

Location of where password stored:

log-error=/var/log/mysqld.log

grep temp /var/log/mysqld.log

Find password after the following text:

A temporary password is generated for root@localhost: qzjLWfHS)0sh

i.) Hardening the MySQL database server by running secure installation and selecting the same options as below.

$ /usr/bin/mysql_secure_installation

[...]

Enter current password for root (enter for none)

OK, successfully used password, moving on...

[...]

Set root password? [Y/n] y

New password: F4tC0ntr0ll3r!

Re-enter new password: F4tC0ntr0ll3r!

Remove anonymous users? [Y/n] Y

[...]

Disallow root login remotely? [Y/n] N

[...]

Remove test database and access to it [Y/n] Y

[...]

Reload privilege tables now? [Y/n] Y

j.) Test new root login works.

mysql -u root -p

k.) Disable the Linux Firewall.

$ systemctl stop firewalld

$ systemctl disable firewalld

$ systemctl status firewalld

CREATING DATABASE FOR CLOUDERA MANAGER

a.) **Creating Databases for Reports Manager, Hive Metastore Server, Sentry Server, Cloudera Navigator Audit Server, and Cloudera Navigator Metadata Server**.

create database rmon DEFAULT CHARACTER SET utf8;

b.) Create a database users and passwords.

grant all on rmon.* TO 'rmon'@'%' IDENTIFIED BY !';

INSTALLING THE MYSQL JDBC DRIVER

a.) Download and copy the MySQL JDBC driver to home folder using WINSCP.
http://www.mysql.com/downloads/connector/j/5.1.html

wget https://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-5.1.46.tar.gz

b.) Extract the JDBC driver JAR file from the downloaded file.

tar zxvf mysql-connector-java-5.1.46.tar.gz

c.) Create the target directory.

$ mkdir -p /usr/share/java/

d.) Copy the JDBC driver, renamed, to the relevant host.

$ cp mysql-connector-java-5.1.46-bin.jar /usr/share/java/mysql-connector-java.jar

Establish Your Cloudera Manager Repository Strategy

RHEL-compatible

1. Save the appropriate Cloudera Manager repo file (`cloudera-manager.repo`) for your system.
   See the **Repo File** column in the Repositories table for the URL.
2. Copy the repo file to the `/etc/yum.repos.d/` directory.

```
[cloudera-manager]
# Packages for Cloudera Manager, Version 5, on RedHat or CentOS 7 x86_64
name=Cloudera Manager
baseurl=https://archive.cloudera.com/cm5/redhat/7/x86_64/cm/5.10.1/
gpgkey =https://archive.cloudera.com/cm5/redhat/7/x86_64/cm/RPM-GPG-KEY-cloudera
gpgcheck = 1
```

Install Cloudera Manager Server Software

Step 2: Install Java

oracle-j2sdk1.7

```
vi .bashrc
export JAVA_HOME=/usr/java/jdk1.7.0_80
export PATH=$JAVA_HOME/bin:$PATH
```

Step 3: Install the Cloudera Manager Server Packages

yum install cloudera-manager-daemons cloudera-manager-server

Preparing a Cloudera Manager Server External Database

Run the `scm_prepare_database.sh` script on the host where the Cloudera Manager Server package is installed:

- Installer or package install
  - /usr/share/cmf/schema/scm_prepare_database.sh database-type [options] database-name username password

/usr/share/cmf/schema/scm_prepare_database.sh mysql -h x.x.x.x clouderascm clouderascm xxxx

Start the Cloudera Manager Server

Start the Cloudera Manager Server

service cloudera-scm-server start

Start and Log into the Cloudera Manager Admin Console

http://Server host:7180

CM log : /var/log/cloudera-scm-server/cloudera-scm-server.log

Choose Cloudera Manager Hosts

Choose the Software Installation Type and Install Software

Add Services

Configure Database Settings

Review Configuration Changes and Start Services

Change the Default Administrator Password

Hive Architecture and its Components

The following image describes the Hive Architecture and the flow in which a query is submitted into Hive and finally processed using the MapReduce framework:

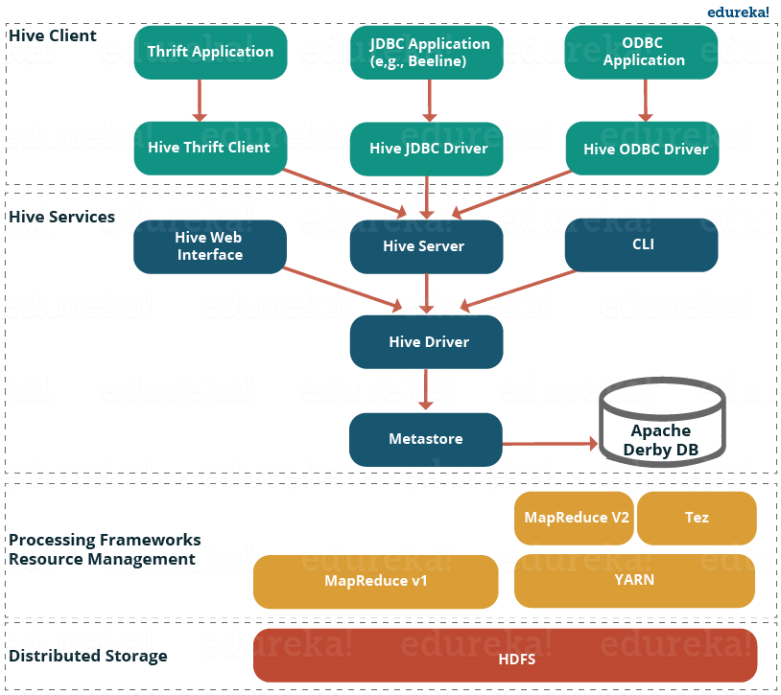


Fig*: Hive Tutorial – Hive Architecture*

As shown in the above image, the 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. We will explore each one of them shortly in this Hive tutorial blog.
- **Processing framework and Resource Management:** Internally, Hive uses Hadoop MapReduce framework as de facto engine to execute the queries. ***Hadoop MapReduce framework*** is a separate topic in itself and therefore, is not discussed here.
- **Distributed Storage:** As Hive is installed on top of Hadoop, it uses the underlying HDFS for the distributed storage. You can refer to the ***HDFS blog*** to learn more about it.

Now, let us explore the first two major components in the Hive Architecture:

1. Hive Clients:

Apache Hive supports different types of client applications for performing queries on the Hive. These clients can be categorized into three types:

- *Thrift Clients:* As Hive server is based on Apache Thrift, it can serve the request from all those programming language that supports Thrift.
- *JDBC Clients:* Hive allows Java applications to connect to it using the JDBC driver which is defined in the class org.apache.hadoop.hive.jdbc.HiveDriver.
- *ODBC Clients:* The Hive ODBC Driver allows applications that support the ODBC protocol to connect to Hive. (Like the JDBC driver, the ODBC driver uses Thrift to communicate with the Hive server.)

2. Hive Services:

Hive provides many services as shown in the image above. Let us have a look at each of them:

- **Hive CLI (Command Line Interface):** This is the default shell provided by the Hive where you can execute your Hive queries and commands directly.
- **Apache Hive Web Interfaces:** Apart from the command line interface, Hive also provides a web based GUI for executing Hive queries and commands.
- **Hive Server:** Hive server is built on Apache Thrift and therefore, is also referred as Thrift Server that allows different clients to submit requests to Hive and retrieve the final result.
- **Apache Hive Driver:** It is responsible for receiving the queries submitted through the CLI, the web UI, Thrift, ODBC or JDBC interfaces by a client. Then, the driver passes the query to the compiler where parsing, type checking and semantic analysis takes place with the help of schema present in the metastore. In the next step, an optimized logical plan is generated in the form of a DAG (Directed Acyclic Graph) of map-reduce tasks and HDFS tasks. Finally, the execution engine executes these tasks in the order of their dependencies, using Hadoop.
- **Metastore:** You can think metastore as a central repository for storing all the Hive metadata information. Hive metadata includes various types of information like structure of tables and the partitions along with the column, column type, serializer and deserializer which is required for Read/Write operation on the data present in HDFS. The metastore comprises of two fundamental units:
    - A service that provides metastore access to other Hive services.
    - Disk storage for the metadata which is separate from HDFS storage.

Now, let us understand the different ways of implementing Hive metastore in the next section of this Hive Tutorial.

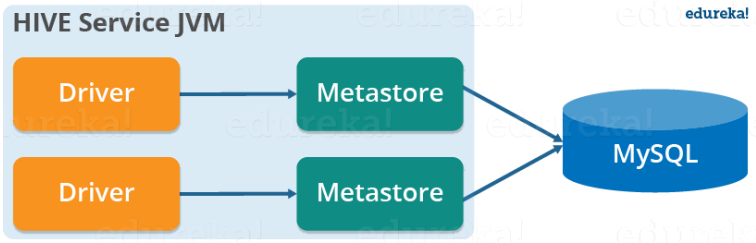
Apache Hive Tutorial: Metastore Configuration

Metastore stores the meta data information using RDBMS and an open source ORM (Object Relational Model) layer called Data Nucleus which converts the object representation into relational schema and vice versa. The reason for choosing RDBMS instead of HDFS is to achieve low latency. We can implement metastore in following three configurations:

1. Embedded Metastore:

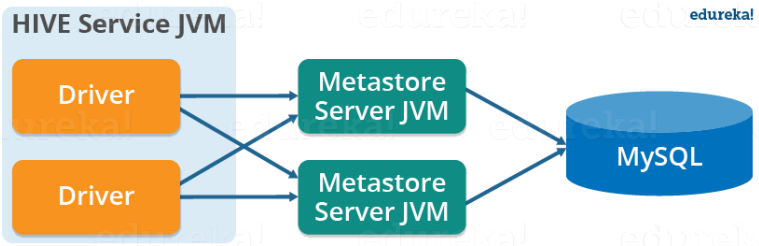
Both the metastore service and the Hive service runs in the same JVM by default using an embedded Derby Database instance where metadata is stored in the local disk. This is called embedded metastore configuration. In this case, only one user can connect to metastore database at a time. If you start a second instance of Hive driver, you will get an error. This is good for unit testing, but not for the practical solutions.

2. Local Metastore:



This configuration allows us to have multiple Hive sessions i.e. Multiple users can use the metastore database at the same time. This is achieved by using any JDBC compliant database like MySQL which runs in a separate JVM or a different machine than that of the Hive service and metastore service which are running in the same JVM as shown above. In general, the most popular choice is to implement a MySQL server as the metastore database.

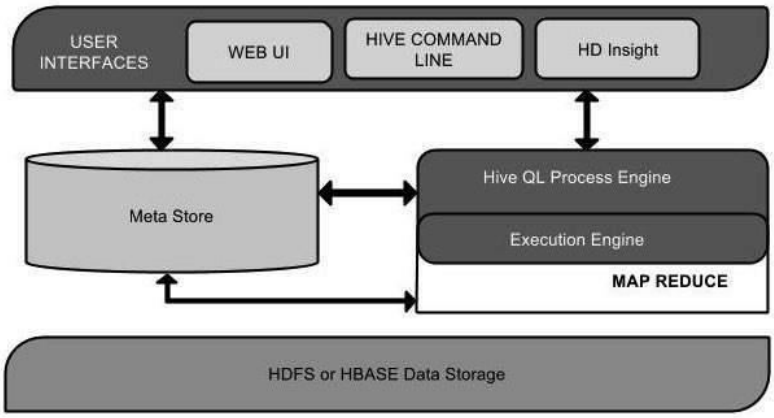
3. Remote Metastore:



In the remote metastore configuration, the metastore service runs on its own separate JVM and not inthe Hive service JVM. Other processes communicate with the metastore server using Thrift Network APIs. You can have one or more metastore servers in this case to provide more availability. The main advantage of using remote metastore is you do not need to share JDBC login credential with each Hive user to access the metastore database.

Architecture of Hive

The following component diagram depicts the architecture of Hive:

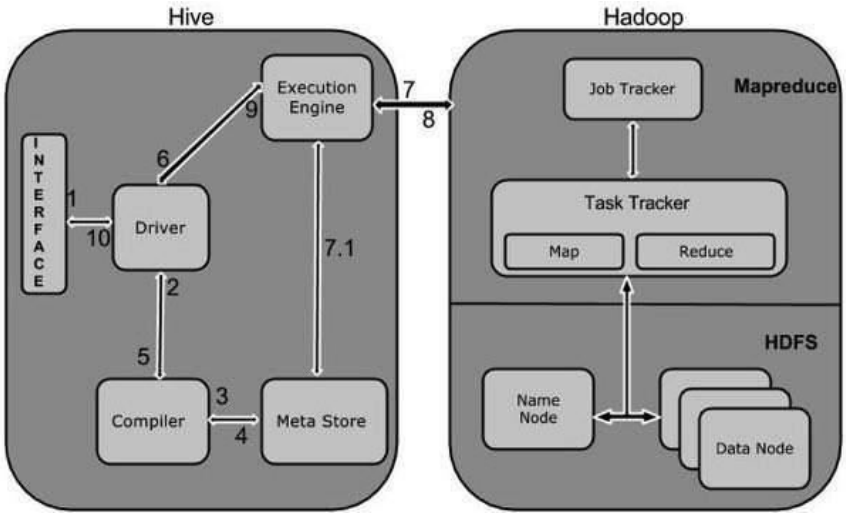


This component diagram contains different units. The following table describes each unit:

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 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 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 HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.



The following table defines how Hive interacts with Hadoop framework:

Step No.	Operation
1	**Execute Query** The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.
2	**Get Plan** The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.
3	**Get Metadata** The compiler sends metadata request to Metastore (any database).
4	**Send Metadata** Metastore sends metadata as a response to the compiler.
5	**Send Plan** The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	**Execute Plan** The driver sends the execute plan to the execution engine.
7	**Execute Job** 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. Here, the query executes MapReduce job.

7.1	**Metadata Ops** Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	**Fetch Result** The execution engine receives the results from Data nodes.
9	**Send Results** The execution engine sends those resultant values to the driver.
10	**Send Results** The driver sends the results to Hive Interfaces.

Data Model

Tables:
Tables in Hive are the same as the tables present in a Relational Database. You can perform filter, project, join and union operations on them.

There are two types of tables in Hive:

1. Managed Table:

Command:

CREATE TABLE <table_name> (column1 data_type, column2 data_type);
LOAD DATA INPATH <HDFS_file_location> INTO table managed_table;

As the name suggests (managed table), Hive is responsible for managing the data of a managed table. In other words, what I meant by saying, "Hive manages the data", is that if you load the data from a file present in HDFS into a Hive *Managed Table*and issue a DROP command on it, the table along with its metadata will be deleted. So, the data belonging to the dropped*managed_table* no longer exist anywhere in HDFS and you can't retrieve it by any means. Basically, you are moving the data when you issue the LOAD command from the HDFS file location to the Hive warehouse directory.

Note: The default path of the warehouse directory is set to/user/hive/warehouse. The data of a Hive table resides in warehouse_directory/table_name (HDFS). You can also specify the path of the warehouse directory in the hive.metastore.warehouse.dir configuration parameter present in the hive-site.xml.

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. In this case, when you issue the LOAD command, Hive moves the data into its warehouse directory. Then, Hive creates the metadata information for the external table. Now, if you issue a DROP command on the *external table*, only metadata information regarding the external table will be deleted. Therefore, you can still retrive the data of that very external table from the warehouse directory using HDFS commands.

Partitions:

Command:

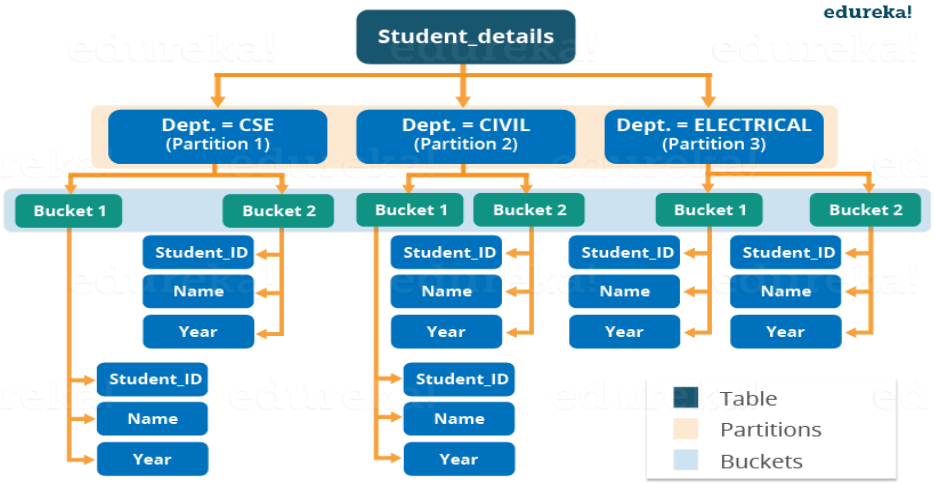
CREATE TABLE table_name (column1 data_type, column2 data_type) PARTITIONED BY (partition1 data_type, partition2 data_type,….);
Hive organizes tables into partitions for grouping similar type of data together based on a column or partition key. Each Table can have one or more partition keys to identify a particular partition. This allows us to have a faster query on slices of the data.

Note: Remember, the most common mistake made while creating partitions is to specify an existing column name as a partition column. While doing so, you will receive an error – "Error in semantic analysis: Column repeated in partitioning columns".

Let us understand partition by taking an example where I have a table student_details containing the student information of some engineering college 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. Physically, a partition is nothing but a sub-directory in the table directory.
Let's say we have data for three departments in our student_details table – CSE, ECE and Civil. Therefore, we will have three partitions in total for each of the departments as shown in the image below. And, for each department we will have all the data regarding that very department residing in a separate sub – directory under the Hive table directory. For example, all the student data regarding CSE departments will be stored in user/hive/warehouse/student_details/dept.=CSE. So, the queries regarding CSE students would only have to look through the data present in the CSE partition. This makes partitioning very useful as it reduces the query latency by scanning only **relevant** partitioned data instead of the whole data set. In fact, in real world implementations, you will be dealing with hundreds of TBs of data. So, imagine scanning this huge amount of data for some query where **95%** data scanned by you was un-relevant to your query.

I would suggest you to go through the blog on ***Hive commands*** where you will find different ways of implementing partitions with an example.



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;

Now, you may divide each partition or the unpartitioned table into Buckets based on the hash function of a column in the table. Actually, each bucket is just a file in the partition directory or the table directory (unpartitioned table). Therefore, if you have chosen to divide the partitions into n buckets, you will have n files in each of your partition directory. For example, you can see the above image where we have bucketed each partition into 2 buckets. So, each partition, say CSE, will have two files where each of them will be storing the CSE student's data.

How Hive distributes the rows into buckets?

Well, Hive determines the bucket number for a row by using the formula: **hash_function (bucketing_column) modulo (num_of_buckets)**. Here, hash_function depends on the column data type. For example, if you are bucketing the table on the basis of some column, let's say user_id, of INT datatype, the hash_function will be – **hash_function (user_id)= integer value of user_id**. And, suppose you have created two buckets, then Hive will determine the rows going to bucket 1 in each partition by calculating: (value of user_id) modulo (2). Therefore, in this case, rows having user_id ending with an even integer digit will reside in a same bucket corresponding to each partition. The hash_function for other data types is a bit complex to calculate and in fact, for a string it is not even humanly recognizable.

Note: If you are using Apache Hive 0.x or 1.x, you have to issue command – set hive.enforce.bucketing = true; from your Hive terminal before performing bucketing. This will allow you to have the correct number of reducer while using cluster by clause for bucketing a column. In case you have not done it, you may find the number of files that has been generated in your table directory are not equal to the number of buckets. As an alternative, you may also set the number of reducer equal to the number of buckets by using set mapred.reduce.task = num_bucket.

Why do we need buckets?
There are two main reasons for performing bucketing to a partition:

- A ***map side join*** requires the data belonging to a unique join key to be present in the same partition. But what about those cases where your partition key differs from join? Therefore, in these cases you can perform a map side join by bucketing the table using the join key.
- Bucketing makes the sampling process more efficient and therefore, allows us to decrease the query time.

What is Impala?

Impala is a MPP (Massive Parallel Processing) SQL query engine for processing huge volumes of data that is stored in Hadoop cluster. It is an open source software which is written in C++ and Java. It provides high performance and low latency compared to other SQL engines for Hadoop.

In other words, Impala is the highest performing SQL engine (giving RDBMS-like experience) which provides the fastest way to access data that is stored in Hadoop Distributed File System.

Why Impala?

Impala combines the SQL support and multi-user performance of a traditional analytic database with the scalability and flexibility of Apache Hadoop, by utilizing standard components such as HDFS, HBase, Metastore, YARN, and Sentry.

- With Impala, users can communicate with HDFS or HBase using SQL queries in a faster way compared to other SQL engines like Hive.
- Impala can read almost all the file formats such as Parquet, Avro, RCFile used by Hadoop.

Impala uses the same metadata, SQL syntax (Hive SQL), ODBC driver, and user interface (Hue Beeswax) as Apache Hive, providing a familiar and unified platform for batch-oriented or real-time queries.

Unlike Apache Hive, **Impala is not based on MapReduce algorithms**. It implements a distributed architecture based on **daemon processes** that are responsible for all the aspects of query execution that run on the same machines.

Thus, it reduces the latency of utilizing MapReduce and this makes Impala faster than Apache Hive.

Advantages of Impala

Here is a list of some noted advantages of Cloudera Impala.

- Using impala, you can process data that is stored in HDFS at lightning-fast speed with traditional SQL knowledge.
- Since the data processing is carried where the data resides (on Hadoop cluster), data transformation and data movement is not required for data stored on Hadoop, while working with Impala.
- Using Impala, you can access the data that is stored in HDFS, HBase, and Amazon s3 without the knowledge of Java (MapReduce jobs). You can access them with a basic idea of SQL queries.
- To write queries in business tools, the data has to be gone through a complicated extract-transform-load (ETL) cycle. But, with Impala, this procedure is shortened. The time-consuming stages of loading & reorganizing is overcome with the new techniques such as ***exploratory data analysis & data discovery*** making the process faster.
- Impala is pioneering the use of the Parquet file format, a columnar storage layout that is optimized for large-scale queries typical in data warehouse scenarios.

Features of Impala

Given below are the features of cloudera Impala −

- Impala is available freely as open source under the Apache license.
- Impala supports in-memory data processing, i.e., it accesses/analyzes data that is stored on Hadoop data nodes without data movement.
- You can access data using Impala using SQL-like queries.
- Impala provides faster access for the data in HDFS when compared to other SQL engines.
- Using Impala, you can store data in storage systems like HDFS, Apache HBase, and Amazon s3.
- You can integrate Impala with business intelligence tools like Tableau, Pentaho, Micro strategy, and Zoom data.
- Impala supports various file formats such as, LZO, Sequence File, Avro, RCFile, and Parquet.
- Impala uses metadata, ODBC driver, and SQL syntax from Apache Hive.

Relational Databases and Impala

Impala uses a Query language that is similar to SQL and HiveQL. The following table describes some of the key dfferences between SQL and Impala Query language.

Impala	Relational databases
Impala uses an SQL like query language that is similar to HiveQL.	Relational databases use SQL language.
In Impala, you cannot update or delete individual records.	In relational databases, it is possible to update or delete individual records.

Impala does not support transactions.	Relational databases support transactions.
Impala does not support indexing.	Relational databases support indexing.
Impala stores and manages large amounts of data (petabytes).	Relational databases handle smaller amounts of data (terabytes) when compared to Impala.

Hive, Hbase, and Impala

Though Cloudera Impala uses the same query language, metastore, and the user interface as Hive, it differs with Hive and HBase in certain aspects. The following table presents a comparative analysis among HBase, Hive, and Impala.

HBase	Hive	Impala
HBase is wide-column store database based on Apache Hadoop. It uses the concepts of BigTable.	Hive is a data warehouse software. Using this, we can access and manage large distributed datasets, built on Hadoop.	Impala is a tool to manage, analyze data that is stored on Hadoop.
The data model of HBase is wide column store.	Hive follows Relational model.	Impala follows Relational model.
HBase is developed using Java language.	Hive is developed using Java language.	Impala is developed using C++.
The data model of HBase is schema-free.	The data model of Hive is Schema-based.	The data model of Impala is Schema-based.
HBase provides Java, RESTful and, Thrift API's.	Hive provides JDBC, ODBC, Thrift API's.	Impala provides JDBC and ODBC API's.
Supports programming languages like C, C#, C++, Groovy, Java PHP, Python, and Scala.	Supports programming languages like C++, Java, PHP, and Python.	Impala supports all languages supporting JDBC/ODBC.
HBase provides support for triggers.	Hive does not provide any support for triggers.	Impala does not provide any support for triggers.

All these three databases −

- Are NOSQL databases.
- Available as open source.
- Support server-side scripting.
- Follow ACID properties like Durability and Concurrency.
- Use **sharding** for **partitioning**.

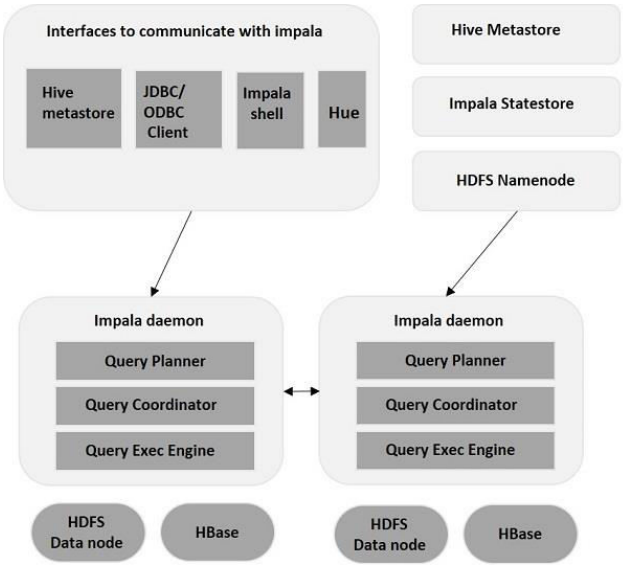
Drawbacks of Impala

Some of the drawbacks of using Impala are as follows −

- Impala does not provide any support for Serialization and Deserialization.
- Impala can only read text files, not custom binary files.
- Whenever new records/files are added to the data directory in HDFS, the table needs to be refreshed.

Impala - Architecture

Impala is an MPP (Massive Parallel Processing) query execution engine that runs on a number of systems in the Hadoop cluster. Unlike traditional storage systems, impala is decoupled from its storage engine. It has three main components namely, Impala daemon *(Impalad)*, Impala Statestore, and Impala metadata or metastore.



The Impala Daemon

The core Impala component is a daemon process that runs on each DataNode of the cluster, physically represented by the `impalad` process. It reads and writes to data files; accepts queries transmitted from the `impala-shell` command, Hue, JDBC, or ODBC; parallelizes the queries and distributes work across the cluster; and transmits intermediate query results back to the central coordinator node.

You can submit a query to the Impala daemon running on any DataNode, and that instance of the daemon serves as the **coordinator node** for that query. The other nodes transmit partial results back to the coordinator, which constructs the final result set for a query. When running experiments with functionality through the `impala-shell` command, you might always connect to the same Impala daemon for convenience. For clusters running production workloads, you might load-balance by submitting each query to a different Impala daemon in round-robin style, using the JDBC or ODBC interfaces.

The Impala daemons are in constant communication with the **statestore**, to confirm which nodes are healthy and can accept new work.

They also receive broadcast messages from the catalogd daemon (introduced in Impala 1.2) whenever any Impala node in the cluster creates, alters, or drops any type of object, or when an `INSERT` or `LOAD DATA` statement is processed through Impala. This background communication minimizes the need for `REFRESH` or `INVALIDATE METADATA` statements that were needed to coordinate metadata across nodes prior to Impala 1.2.

The Impala Statestore

The Impala component known as the **statestore** checks on the health of Impala daemons on all the DataNodes in a cluster, and continuously relays its findings to each of those daemons. It is physically represented by a daemon process named `statestored`; you only need such a process on one host in the cluster. If an Impala daemon goes offline due to hardware failure, network error, software issue, or other reason, the statestore informs all the other Impala daemons so that future queries can avoid making requests to the unreachable node.

Because the statestore's purpose is to help when things go wrong and to broadcast metadata to coordinators, it is not always critical to the normal operation of an Impala cluster. If the statestore is not running or becomes unreachable, the Impala daemons continue running and distributing work among themselves as usual when working with the data known to Impala. The cluster just becomes less robust if other Impala daemons fail, and metadata becomes less consistent as it changes while the statestore is offline. When the statestore comes back online, it re-establishes communication with the Impala daemons and resumes its monitoring and broadcasting functions.

If you issue a DDL statement while the statestore is down, the queries that access the new object the DDL created will fail.

Most considerations for load balancing and high availability apply to the impalad daemon. The statestoredand catalogd daemons do not have special requirements for high availability, because problems with those daemons do not result in data loss. If those daemons become unavailable due to an outage on a particular host, you can stop the Impala service, delete the **Impala StateStore** and **Impala Catalog Server** roles, add the roles on a different host, and restart the Impala service.

The Impala Catalog Service

The Impala component known as the **catalog service** relays the metadata changes from Impala SQL statements to all the Impala daemons in a cluster. It is physically represented by a daemon process named `catalogd`; you only need such a process on one host in the cluster. Because the requests are passed through the statestore daemon, it makes sense to run the statestored and catalogd services on the same host.

The catalog service avoids the need to issue `REFRESH` and `INVALIDATE METADATA` statements when the metadata changes are performed by statements issued through Impala. When you create a table, load data, and so on through Hive, you do need to issue `REFRESH` or `INVALIDATE METADATA` on an Impala node before executing a query there.

Query Processing Interfaces

To process queries, Impala provides three interfaces as listed below.

- **Impala-shell** − After setting up Impala using the Cloudera VM, you can start the Impala shell by typing the command **impala-shell** in the editor. We will discuss more about the Impala shell in coming chapters.
- **Hue interface** − You can process Impala queries using the Hue browser. In the Hue browser, you have Impala query editor where you can type and execute the impala queries. To access this editor, first of all, you need to logging to the Hue browser.
- **ODBC/JDBC drivers** − Just like other databases, Impala provides ODBC/JDBC drivers. Using these drivers, you can connect to impala through programming languages that supports these drivers and build applications that process queries in impala using those programming languages.

Query Execution Procedure

Whenever users pass a query using any of the interfaces provided, this is accepted by one of the Impalads in the cluster. This Impalad is treated as a coordinator for that particular query.

After receiving the query, the query coordinator verifies whether the query is appropriate, using the **Table Schema** from the Hive meta store. Later, it collects the information about the location of the data that is required to execute the query, from HDFS name node and sends this information to other impalads in order to execute the query.

All the other Impala daemons read the specified data block and processes the query. As soon all the daemons complete their tasks, the query coordinator collects the result back and delivers it to the user.

Apache Spark - Introduction

Industries are using Hadoop extensively to analyze their data sets. The reason is that Hadoop framework is based on a simple programming model (MapReduce) and it enables a computing solution that is scalable, flexible, fault-tolerant and cost effective. Here, the main concern is to maintain speed in processing large datasets in terms of waiting time between queries and waiting time to run the program.

Spark was introduced by Apache Software Foundation for speeding up the Hadoop computational computing software process.
As against a common belief, **Spark is not a modified version of Hadoop**and is not, really, dependent on Hadoop because it has its own cluster management. Hadoop is just one of the ways to implement Spark.

Spark uses Hadoop in two ways – one is **storage** and second is **processing**. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

Apache Spark

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its **in-memory cluster computing** that increases the processing speed of an application.

Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming. Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

Evolution of Apache Spark

Spark is one of Hadoop's sub project developed in 2009 in UC Berkeley's AMPLab by Matei Zaharia. It was Open Sourced in 2010 under a BSD license. It was donated to Apache software foundation in 2013, and now Apache Spark has become a top level Apache project from Feb-2014.

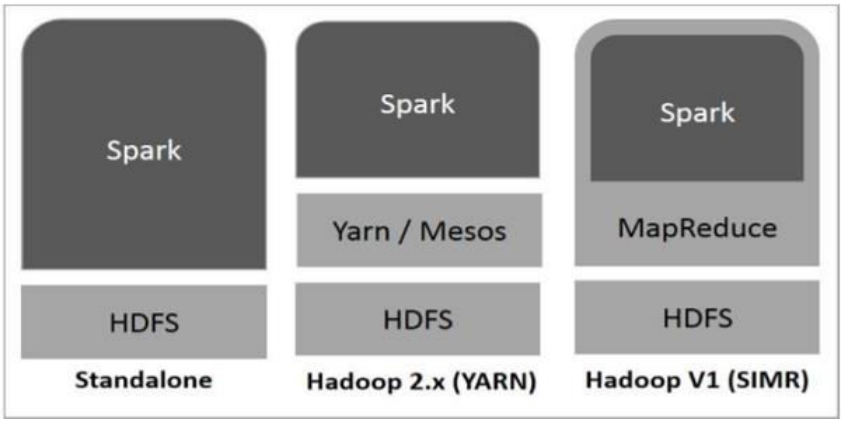
Features of Apache Spark

Apache Spark has following features.

- **Speed** − Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write operations to disk. It stores the intermediate processing data in memory.
- **Supports multiple languages** − Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.
- **Advanced Analytics** − Spark not only supports 'Map' and 'reduce'. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

Spark Built on Hadoop

The following diagram shows three ways of how Spark can be built with Hadoop components.

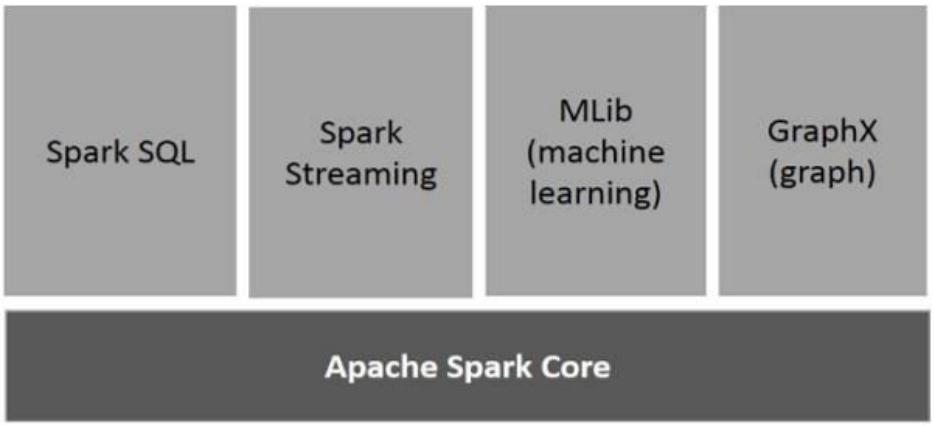


There are three ways of Spark deployment as explained below.

- **Standalone** − Spark Standalone deployment means Spark occupies the place on top of HDFS(Hadoop Distributed File System) and space is allocated for HDFS, explicitly. Here, Spark and MapReduce will run side by side to cover all spark jobs on cluster.
- **Hadoop Yarn** − Hadoop Yarn deployment means, simply, spark runs on Yarn without any pre-installation or root access required. It helps to integrate Spark into Hadoop ecosystem or Hadoop stack. It allows other components to run on top of stack.
- **Spark in MapReduce (SIMR)** − Spark in MapReduce is used to launch spark job in addition to standalone deployment. With SIMR, user can start Spark and uses its shell without any administrative access.

Components of Spark

The following illustration depicts the different components of Spark.



Apache Spark Core

Spark Core is the underlying general execution engine for spark platform that all other functionality is built upon. It provides In-Memory computing and referencing datasets in external storage systems.

Spark SQL

Spark SQL is a component on top of Spark Core that introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data.

Spark Streaming

Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD (Resilient Distributed Datasets) transformations on those mini-batches of data.

MLlib (Machine Learning Library)

MLlib is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture. It is, according to benchmarks, done by the MLlib developers against the Alternating Least Squares (ALS) implementations. Spark MLlib is nine times as fast as the Hadoop disk-based version of **Apache Mahout** (before Mahout gained a Spark interface).

GraphX

GraphX is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API. It also provides an optimized runtime for this abstraction.

Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs − **parallelizing** an existing collection in your driver program, or **referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

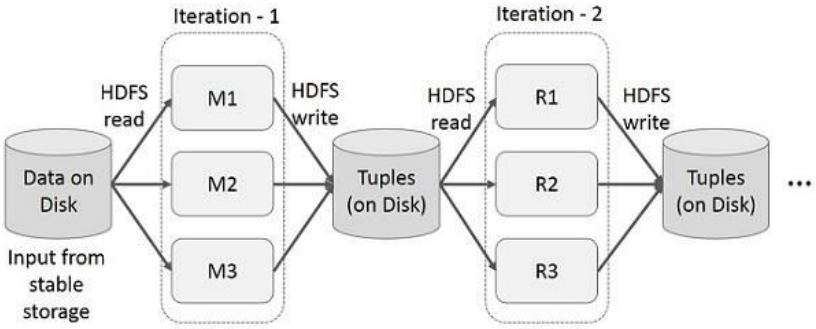
Data Sharing is Slow in MapReduce

MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.
Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex − between two MapReduce jobs) is to write it to an external stable storage system (Ex − HDFS). Although this framework provides numerous abstractions for accessing a cluster's computational resources, users still want more.

Both **Iterative** and **Interactive** applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Iterative Operations on MapReduce

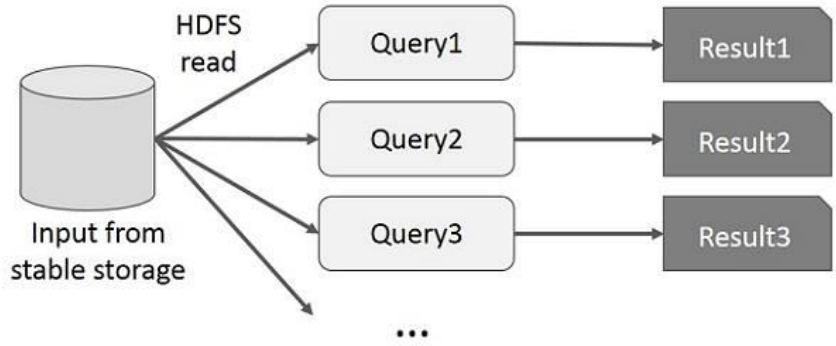
Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce. This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.



Interactive Operations on MapReduce

User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominate application execution time.

The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



Data Sharing using Spark RDD

Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

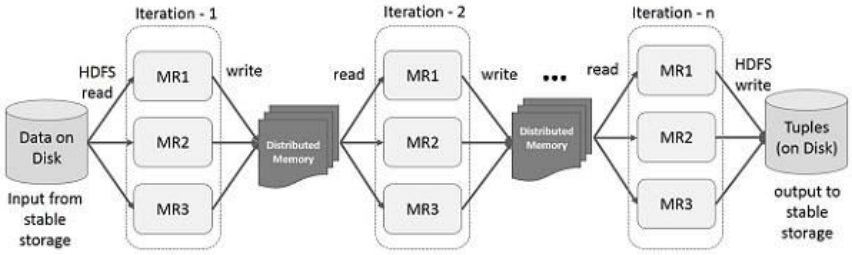
Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is **R**esilient **D**istributed **D**atasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Let us now try to find out how iterative and interactive operations take place in Spark RDD.

Iterative Operations on Spark RDD

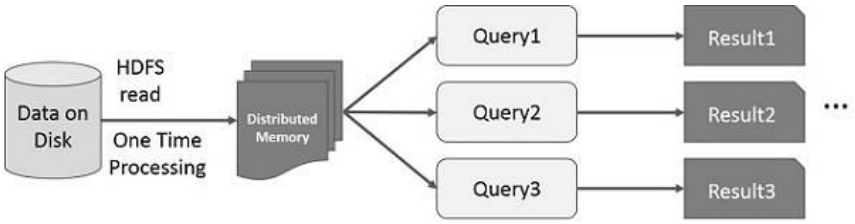
The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

Note − If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.



Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also **persist** an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

Why Spark when Hadoop is already there?
The first of the many questions everyone asks when it comes to Spark is, "***Why Spark when we have Hadoop already?***".
To answer this, we have to look at the concept of batch and real-time processing. *Hadoop* is based on the concept of *batch processing* where the processing happens of blocks of data that have already been stored over a period of time. At the time, Hadoop broke all the expectations with the revolutionary MapReduce framework in 2005. Hadoop MapReduce is the best framework for processing data in batches.
This went on until 2014, till Spark overtook Hadoop. The USP for Spark was that it could *process data* in *real time* and was about 100 times faster than Hadoop MapReduce in batch processing large data sets.

The following figure gives a detailed explanation of the differences between processing in Spark and Hadoop.

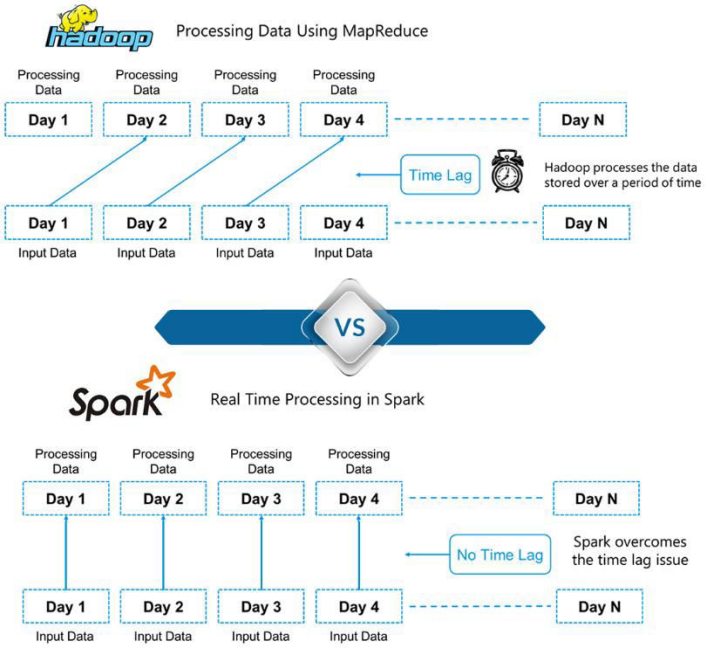


Figure: *Spark Tutorial – Differences between Hadoop and Spark*

Here, we can draw out one of the key differentiators between Hadoop and Spark. Hadoop is based on batch processing of big data. This means that the data is stored over a period of time and is then processed using Hadoop. Whereas in Spark, processing can take place in real-time. This real-time processing power in Spark helps us to solve the use cases of Real Time Analytics we saw in the previous section. Alongside this, Spark is also able to do batch processing 100 times faster than that of Hadoop MapReduce (Processing framework in Apache Hadoop). *Therefore, Apache Spark is the go-to tool for big data processing in the industry.*

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:

1. **Spark Core**

2. **Spark Streaming**
3. **Spark SQL**
4. **GraphX**
5. **MLlib (Machine Learning)**

Spark Core

Spark Core is the base engine for large-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 atop the core allow diverse workloads for streaming, SQL, and machine learning. It is responsible for:

1. Memory management and fault recovery
2. Scheduling, distributing and monitoring jobs on a cluster
3. Interacting with storage systems

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. The fundamental stream unit is DStream which is basically a series of RDDs (Resilient Distributed Datasets) to process the real-time data.

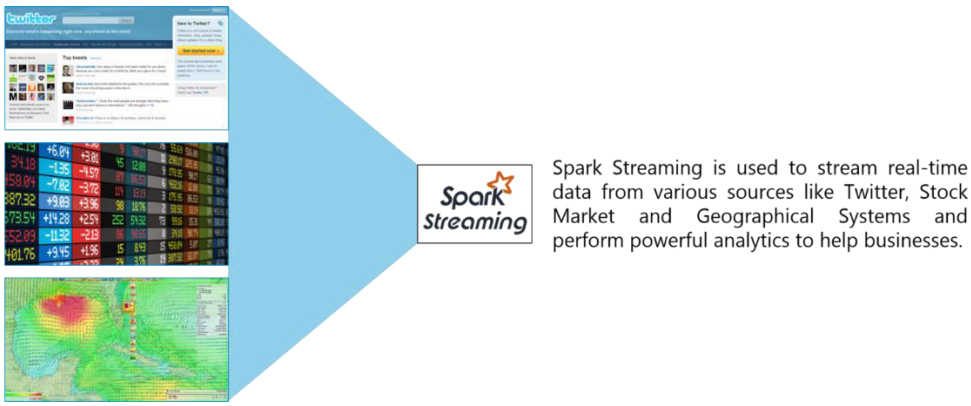


Figure: *Spark Tutorial – Spark Streaming*

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. For those of you familiar with RDBMS, Spark SQL will be an easy transition from your earlier tools where you can extend the boundaries of traditional relational data processing.

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

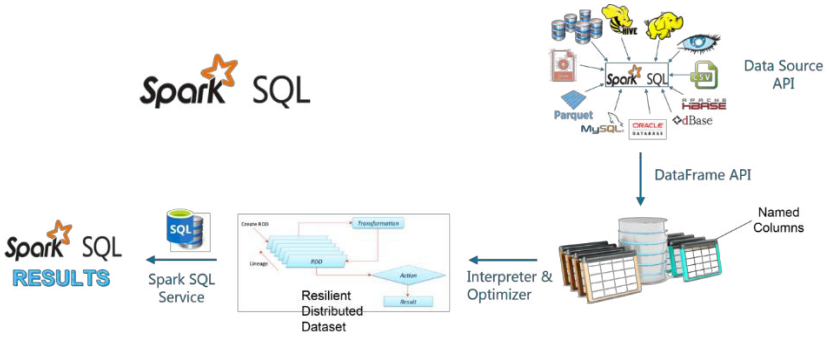


Figure: *The flow diagram represents a Spark SQL process using all the four libraries in sequence*

A complete tutorial on Spark SQL can be found in the given blog: Spark SQL Tutorial Blog

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.

To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and mapReduceTriplets) as well as an optimized variant of the Pregel API. In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.

MILib (Machine Learning)

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

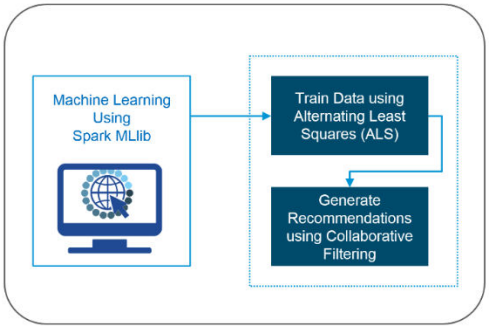


Figure: *Machine Learning Flow Diagram*

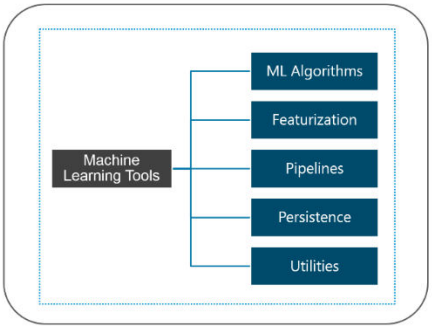
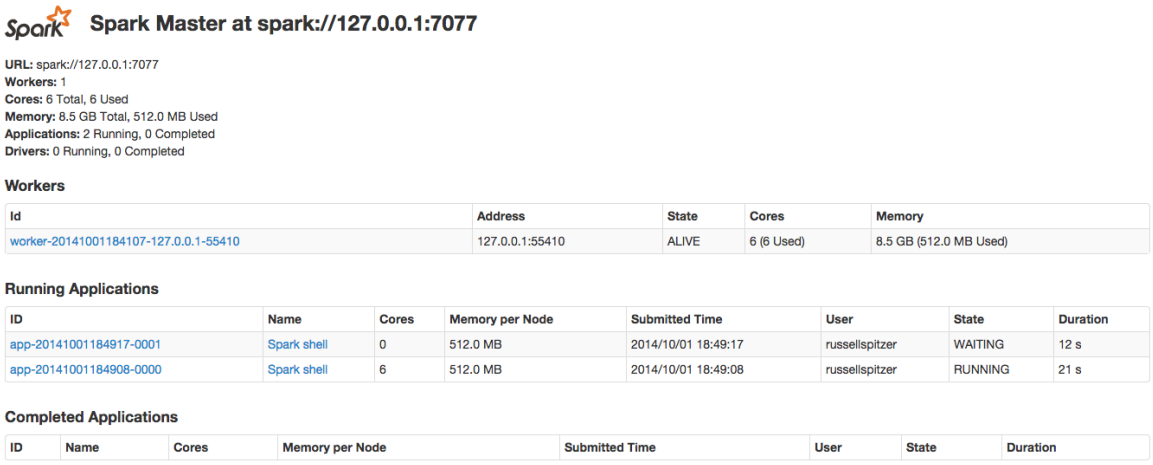


Figure: *Machine Learning Tools*

Common Spark Troubleshooting

This is by far the most common first error that a new Spark user will see when attempting to run a new application. Our new and excited Spark user will attempt to start the shell or run their own application and be met with the following message WARN TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster ui to ensure that workers are registered and have sufficient memory

This message will pop up any time an application is requesting more resources from the cluster than the cluster can currently provide. What resources you might ask? Well Spark is only looking for two things: Cores and Ram. Cores represents the number of open executor slots that your cluster provides for execution. Ram refers to the amount of free Ram required on any worker running your application. Note for both of these resources the maximum value is not your System's max, it is the max as set by the your Spark configuration. To see the current state of your cluster (and it's free resources) check out the UI at **SparkMasterIP:7080** (DSE users can find their SparkMaster URI using *dsetool sparkmaster*.)



Spark Master at spark://127.0.0.1:7077

URL: spark://127.0.0.1:7077
Workers: 1
Cores: 6 Total, 6 Used
Memory: 8.5 GB Total, 512.0 MB Used
Applications: 2 Running, 0 Completed
Drivers: 0 Running, 0 Completed

Workers

Id	Address	State	Cores	Memory
worker-20141001184107-127.0.0.1-55410	127.0.0.1:55410	ALIVE	6 (6 Used)	8.5 GB (512.0 MB Used)

Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20141001184917-0001	Spark shell	0	512.0 MB	2014/10/01 18:49:17	russellspitzer	WAITING	12 s
app-20141001184908-0000	Spark shell	6	512.0 MB	2014/10/01 18:49:08	russellspitzer	RUNNING	21 s

Completed Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

In the above example is a picture of my Spark UI running on my Macbook (localhost:7080). You can see one of the Spark Shell applications is currently waiting. I caused this situation by starting the Spark Shell in 2 different terminals. The first Spark shell has consumed all the available cores in the system leaving the second shell waiting for resources. Until the first spark shell is terminated and its resources are released, all other apps will display the above warning. For more details on how to read the Spark UI check the section below.

The short term solution to this problem is to make sure you aren't requesting more resources from your cluster than exist or to shut down any apps that are unnecessarily using resources. If you need to run multiple Spark apps simultaneously then you'll need to adjust the amount of cores being used by each app.

Long Running Application Metadata Cleanup

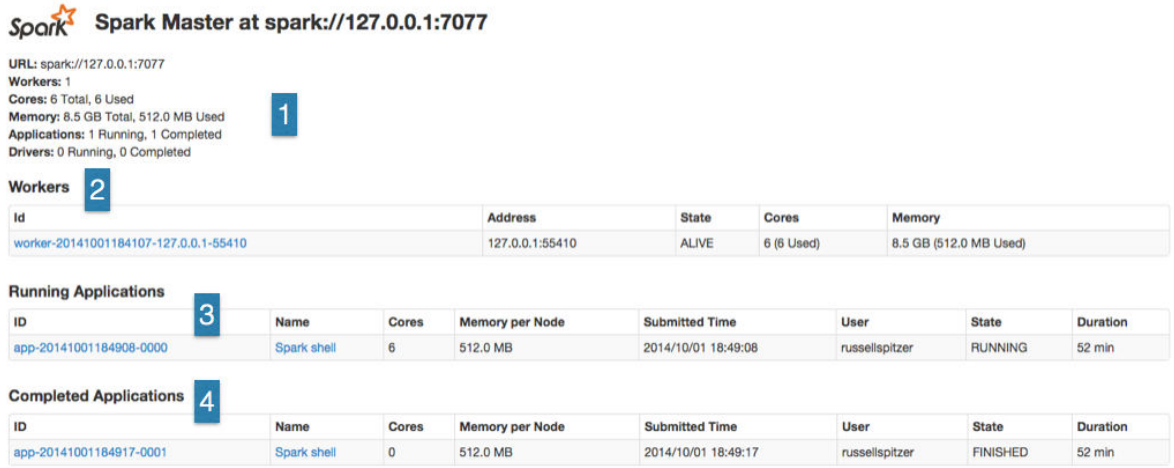
As Spark applications run they create metadata objects which are stored in memory indefinitely by default. For Spark Streaming jobs you are forced to set the variable spark.cleaner.ttl to clean out these objects and prevent an OOM. On other long lived projects you must set this yourself. For those users of Shark Server this is especially important. For DSE users (4.5.x) you can set this property in your **shark-env.sh** file for Shark Server deployments. This will let you have a long running Shark processes without worrying about a sudden OOM. To check if this issue is causing your OOMs look in your heap dumps for a large number of *scheduler.ShuffleMapTasks* and *scheduler.ResultTask objects*.

To set this you would end up modifying your SPARK_JAVA_OPTS variable like this
export SPARK_JAVA_OPTS +="-Dspark.kryoserializer.buffer.mb=10 -Dspark.cleaner.ttl=43200"

A Brief Tour of The Spark UI

Once your job has started and it's not throwing any exceptions you may want to get a picture of what's going on. All of the information about the current state of the application is available on the Spark UI. Here is a brief walkthrough starting with the initial screen>.

If you are running on AWS or GCE you may find it useful to set SPARK_PUBLIC_DNS=PUBLIC_IP for each of the nodes in your cluster. This will cause make the links work correctly and not just connect to the internal provider IP addresses.

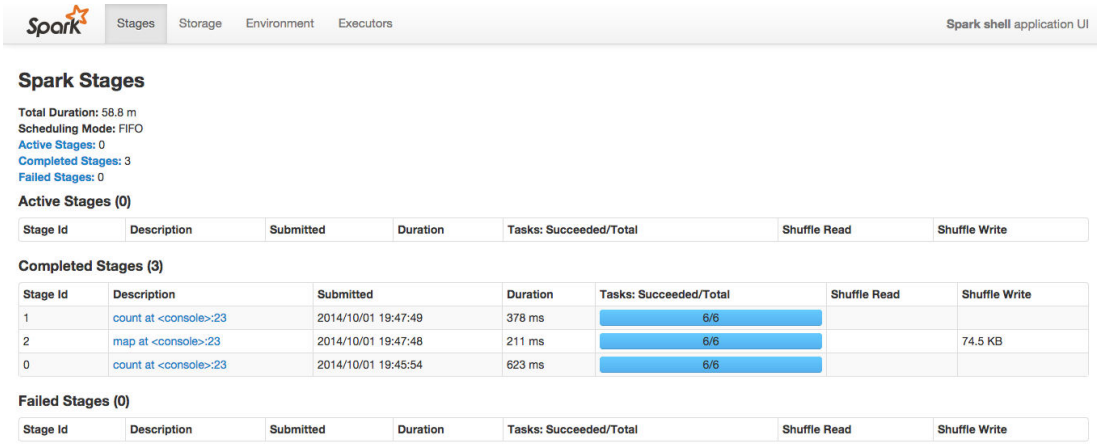


You should see something very similar to this when accessing the UI page for your spark cluster. In the upper left (*1*) you'll see the cluster wide over statistics showing exactly what resources are available. These numbers are aggregates for all of the workers and running jobs in the cluster. Starting at the Workers line (*2*) we'll see what action is actually taking place on our cluster at the moment.

First listed is exactly what workers are currently running and how utilized they are. You can see here on a node by node basis how much memory is free and how many cores are available. This is significant because the upper bound on how much memory an application can use is set at this level. For example a job which requests 8 GB of ram can only run on workers which have at least 8 GB of free ram. If any particular node has hung or is not appearing on the worker list try running **dsetool sparkworker restart** on that node to restart the worker process. (Note: you may see dead workers on this page if you have recently restarted nodes or spark worker processes, this is not an issue the master just hasn't fully confirmed that the old worker is gone.)

Below we that we can see the currently running spark applications (3). Since I'm currently running an instance of the Spark Shell you can see an entry for that listed as running. A line will appear here for every Spark Context object that is created communicating with this master. Since the spark context for the Spark Shell is created on startup and doesn't close until the shell is closed, we should see this listed as long as we are running Spark Shell commands. In the Completed Applications section (4), we can see I've shut down that Spark Shell from earlier that was waiting for resources.

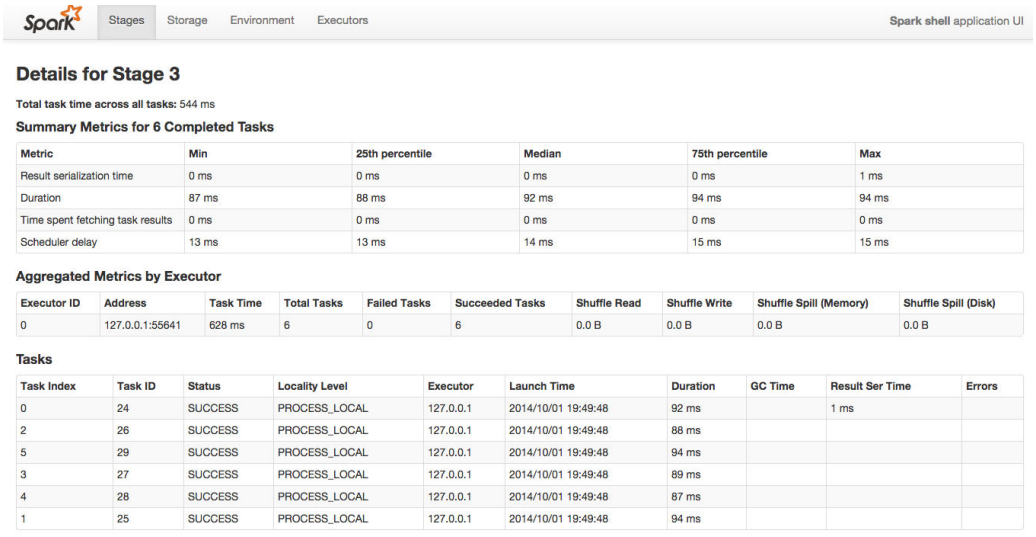
In Spark 1.0+ you can enable event logging which will enable you to see the application detail for past applications but I haven't for this example. This means we can only look into the state of currently running applications. To peek into the app we can click on the applications name ("Spark shell" directly to the right of the 3) and we'll be taken to the App Detail page.



Here we can see the various Stages that this application has completed. For this particular example I've run the following commands.

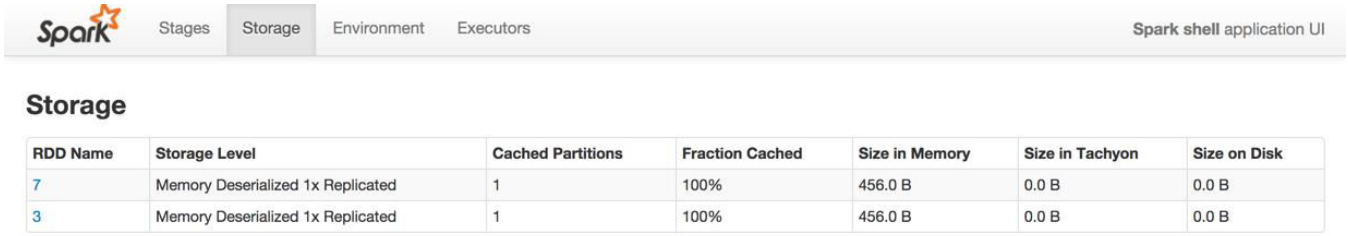
```
sc.parallelize(1 until 10000).count
sc.parallelize(1 until 10000).map( x => (x%30,x)).groupByKey().count
```

You can see each Spark Transformation (map) and Action (count) is reported as a separate stage. We can see that in this case, each of the stages has already completed successfully. We can see for each stage exactly how many Tasks it was broken into and how many of them are currently complete. The number of Tasks here is the maximum level of parallelism that can be accomplished for that stage. The Tasks will be handed out to available executors, so if there are only 2 tasks but 4 executors (cores) then that stage can only ever be run on 2 cores at the same time. Adjusting how many tasks are created is dependent on the underlying RDD and the nature of the transformation you are running, be sure to check your RDD's api to determine how many partitions/tasks will be created. To actually see the details on how a particular stage was accomplished click on the "Description" field for that stage to go to the Stage Detail page.



This page gives us the nitty gritty of how our stage was actually accomplished. At the bottom we see a list of every task, on what machine/or core it was run and how long it took. This is a key place to look to find tasks that failed, on what node they failed and get information about where bottlenecks are. The Summary at the top of the page shows us the summary of all of the entries of the bottom of the page, making it easy to see if there are outliers that may have been running slow for some reason. Use this page to debug performance issues with your tasks.

Let's take a quick detour to the "Storage Tab" at the top of this screen. This will take us to a screen which looks like this.



Here we can see all the RDD's currently stored and by clicking on the "RDD Name" link we can see exactly on which nodes data is being stored. This is helpful when trying to understand exactly whether or not your RDD's are in memory or on disk. For this example I ran two quick RDD operations from the Spark Shell

HBase Data Model

As we know, HBase is a column-oriented NoSQL database. Although it looks similar to a relational database which contains rows and columns, but it is not a relational database. Relational databases are row oriented while HBase is column-oriented. So, let us first understand the difference between Column-oriented and Row-oriented databases:

Row-oriented vs column-oriented Databases:

- Row-oriented databases store table records in a sequence of rows. Whereas column-oriented databases store table records in a sequence of columns, i.e. the entries in a column are stored in contiguous locations on disks.

To better understand it, let us take an example and consider the table below.

Customer ID	Name	Address	Product ID	Product Name
1	Paul Walker	US	231	Gallardo
2	Vin Diesel	Brazil	520	Mustang

If this table is stored in a row-oriented database. It will store the records as shown below:

1, Paul Walker, US, 231, Gallardo,

2, Vin Diesel, Brazil, 520, Mustang

In row-oriented databases data is stored on the basis of rows or tuples as you can see above.

While the column-oriented databases store this data as:

1,2, Paul Walker, Vin Diesel, US, Brazil, 231, 520, Gallardo, Mustang

In a column-oriented databases, all the column values are stored together like first column values will be stored together, then the second column values will be stored together and data in other columns are stored in a similar manner.

- When the amount of data is very huge, like in terms of petabytes or exabytes, we use column-oriented approach, because the data of a single column is stored together and can be accessed faster.
- While row-oriented approach comparatively handles less number of rows and columns efficiently, as row-oriented database stores data is a structured format.
- When we need to process and analyze a large set of semi-structured or unstructured data, we use column oriented approach. Such as applications dealing with **Online Analytical Processing** like data mining, data warehousing, applications including analytics, etc.
- Whereas, **Online Transactional Processing** such as banking and finance domains which handle structured data and require transactional properties (ACID properties) use row-oriented approach.

HBase tables has following components, shown in the image below:

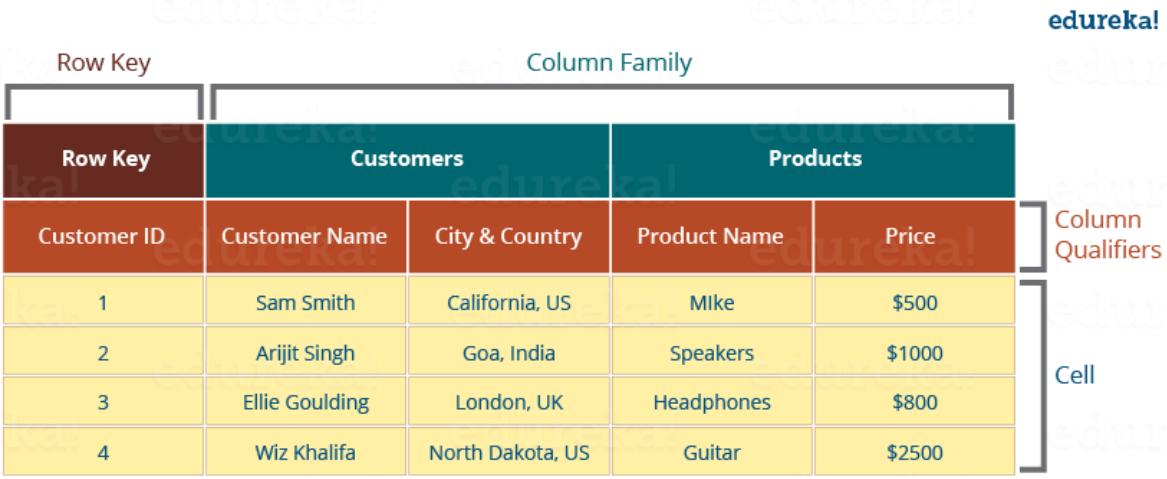


Figure: HBase Table

- **Tables**: Data is stored in a table format in HBase. But here tables are in column-oriented format.
- **Row Key**: Row keys are used to search records which make searches fast. You would be curious to know how? I will explain it in the architecture part moving ahead in this blog.
- **Column Families**: Various columns are combined in a column family. These column families are stored together which makes the searching process faster because data belonging to same column family can be accessed together in a single seek.
- **Column Qualifiers**: Each column's name is known as its column qualifier.
- **Cell**: Data is stored in cells. The data is dumped into cells which are specifically identified by rowkey and column qualifiers.
- **Timestamp**: Timestamp is a combination of date and time. Whenever data is stored, it is stored with its timestamp. This makes easy to search for a particular version of data.

In a more simple and understanding way, we can say HBase consists of:

- Set of tables
- Each table with column families and rows

- Row key acts as a Primary key in HBase.
- Any access to HBase tables uses this Primary Key
- Each column qualifier present in HBase denotes attribute corresponding to the object which resides in the cell.

Now that you know about HBase Data Model, let us see how this data model falls in line with HBase Architecture and makes it suitable for large storage and faster processing.

HBase Architecture: Components of HBase Architecture

HBase has three major components i.e., **HMaster Server**, **HBase Region Server, Regions** and **Zookeeper**.

The below figure explains the hierarchy of the HBase Architecture. We will talk about each one of them individually.

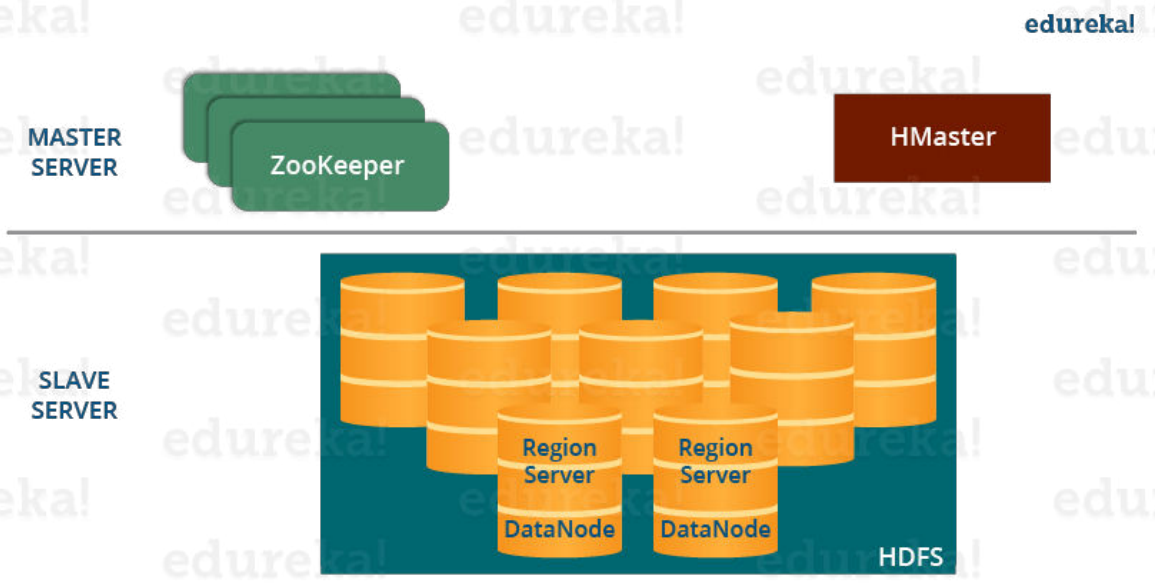


Figure: HMaster & Region Servers

Now before going to the HMaster, we will understand Regions as all these Servers (HMaster, Region Server, Zookeeper) are placed to coordinate and manage Regions and perform various operations inside the Regions. So you would be curious to know what are regions and why are they so important?

HBase Architecture: Region

A region contains all the rows between the start key and the end key assigned to that region. HBase tables can be divided into a number of regions in such a way that

all the columns of a column family is stored in one region. Each region contains the rows in a sorted order.

Many regions are assigned to a **Region Server**, which is responsible for handling, managing, executing reads and writes operations on that set of regions.

So, concluding in a simpler way:

- A table can be divided into a number of regions. A Region is a sorted range of rows storing data between a start key and an end key.
- A Region has a default size of 256MB which can be configured according to the need.
- A Group of regions is served to the clients by a Region Server.
- A Region Server can serve approximately 1000 regions to the client.

Now starting from the top of the hierarchy, I would first like to explain you about HMaster Server which acts similarly as a NameNode in ***HDFS***. Then, moving down in the hierarchy, I will take you through ZooKeeper and Region Server.

HBase Architecture: HMaster

As in the below image, you can see the HMaster handles a collection of Region Server which resides on DataNode. Let us understand how HMaster does that.

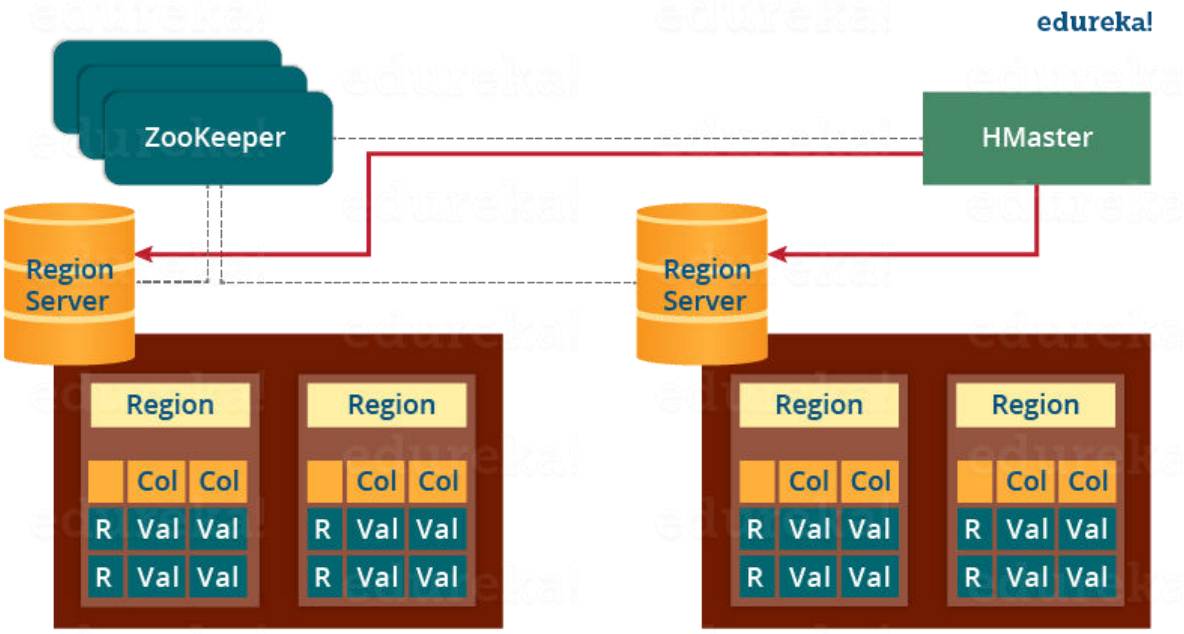


Figure: Components of HBase

- HBase HMaster performs DDL operations (create and delete tables) and assigns regions to the Region servers as you can see in the above image.

- It coordinates and manages the Region Server (similar as NameNode manages DataNode in HDFS).
- It assigns regions to the Region Servers on startup and re-assigns regions to Region Servers during recovery and load balancing.
- It monitors all the Region Server's instances in the cluster (with the help of Zookeeper) and performs recovery activities whenever any Region Server is down.
- It provides an interface for creating, deleting and updating tables.

HBase has a distributed and huge environment where HMaster alone is not sufficient to manage everything. So, you would be wondering what helps HMaster to manage this huge environment? That's where ZooKeeper comes into the picture. After we understood how HMaster manages HBase environment, we will understand how Zookeeper helps HMaster in managing the environment.

HBase Architecture: ZooKeeper – The Coordinator

This below image explains the ZooKeeper's coordination mechanism.

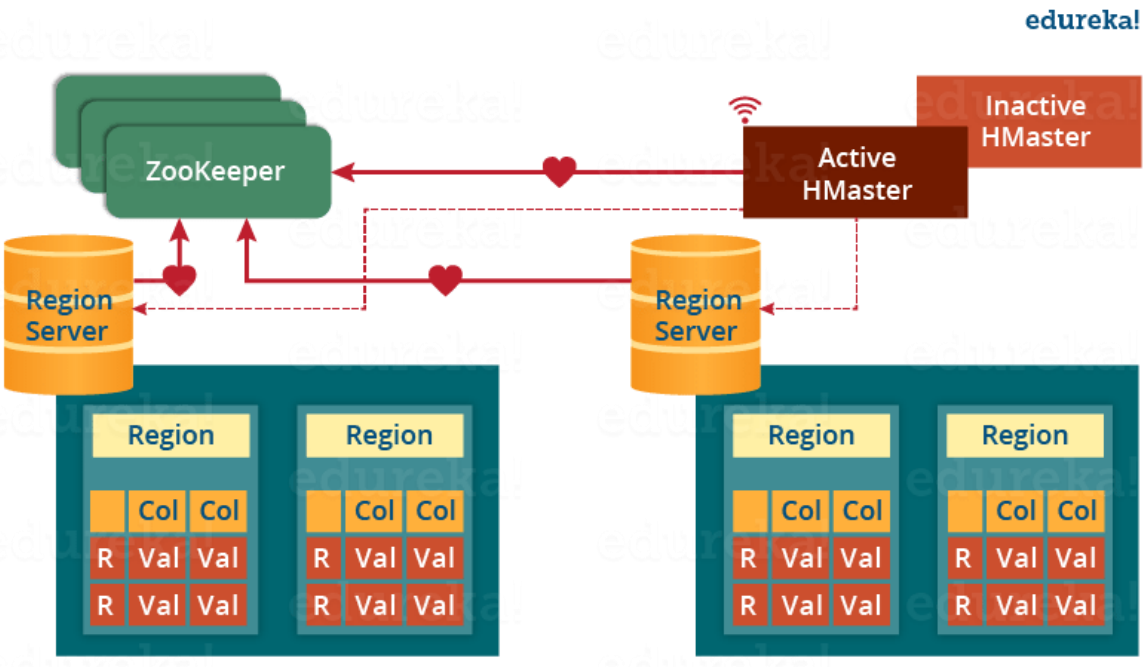


Figure: ZooKeeper as Coordination Service

- Zookeeper acts like a coordinator inside HBase distributed environment. It helps in maintaining server state inside the cluster by communicating through sessions.
- Every Region Server along with HMaster Server sends continuous heartbeat at regular interval to Zookeeper and it checks which server is alive and available as mentioned in above image. It also provides server failure notifications so that, recovery measures can be executed.

- Referring from the above image you can see, there is an inactive server, which acts as a backup for active server. If the active server fails, it comes for the rescue.
- The active HMaster sends heartbeats to the Zookeeper while the inactive HMaster listens for the notification send by active HMaster. If the active HMaster fails to send a heartbeat the session is deleted and the inactive HMaster becomes active.
- While if a Region Server fails to send a heartbeat, the session is expired and all listeners are notified about it. Then HMaster performs suitable recovery actions which we will discuss later in this blog.
- Zookeeper also maintains the .META Server's path, which helps any client in searching for any region. The Client first has to check with .META Server in which Region Server a region belongs, and it gets the path of that Region Server.

As I talked about .META Server, let me first explain to you what is .META server? So, you can easily relate the work of ZooKeeper and .META Server together. Later, when I will explain you the HBase search mechanism in this blog, I will explain how these two work in collaboration.

HBase Architecture: Meta Table

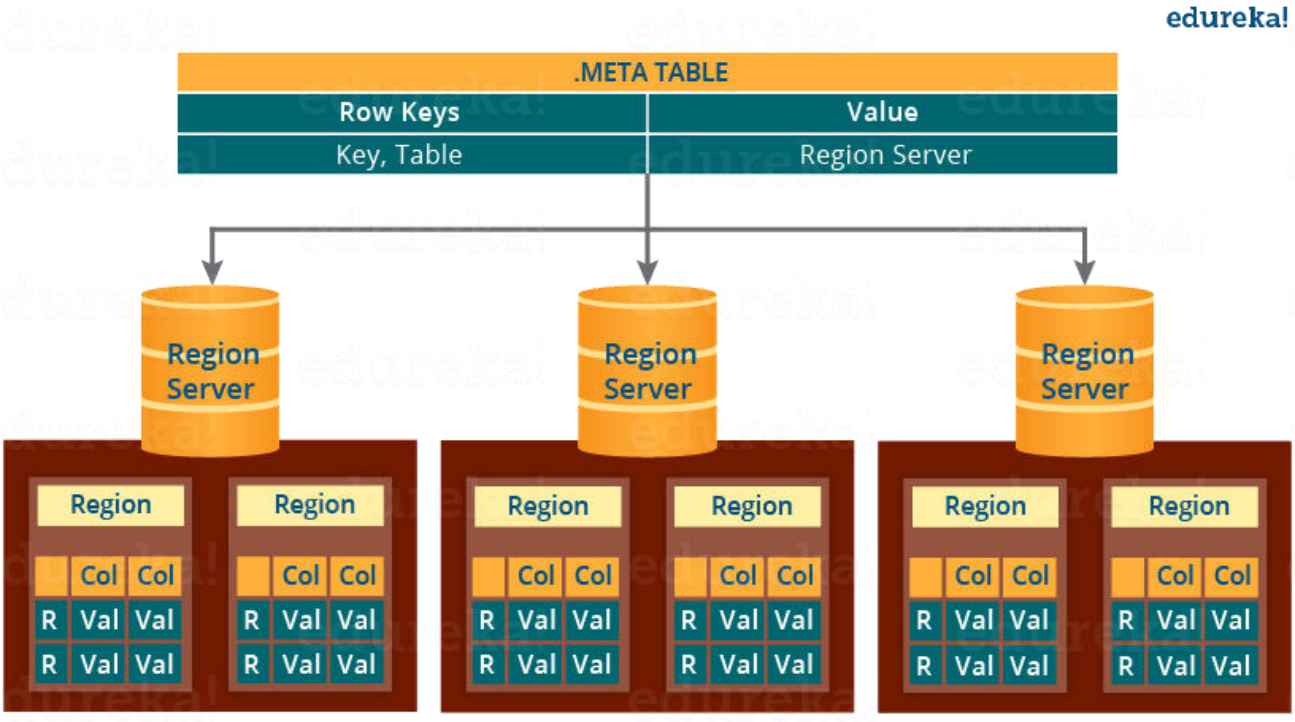


Figure: META Table

- The META table is a special HBase catalog table. It maintains a list of all the Regions Servers in the HBase storage system, as you can see in the above image.

- Looking at the figure you can see, **.META** file maintains the table in form of keys and values. Key represents the start key of the region and its id whereas the value contains the path of the Region Server.

As I already discussed, Region Server and its functions while I was explaining you Regions hence, now we are moving down the hierarchy and I will focus on the Region Server's component and their functions. Later I will discuss the mechanism of searching, reading, writing and understand how all these components work together.

HBase Architecture: Components of Region Server

This below image shows the components of a Region Server. Now, I will discuss them separately.

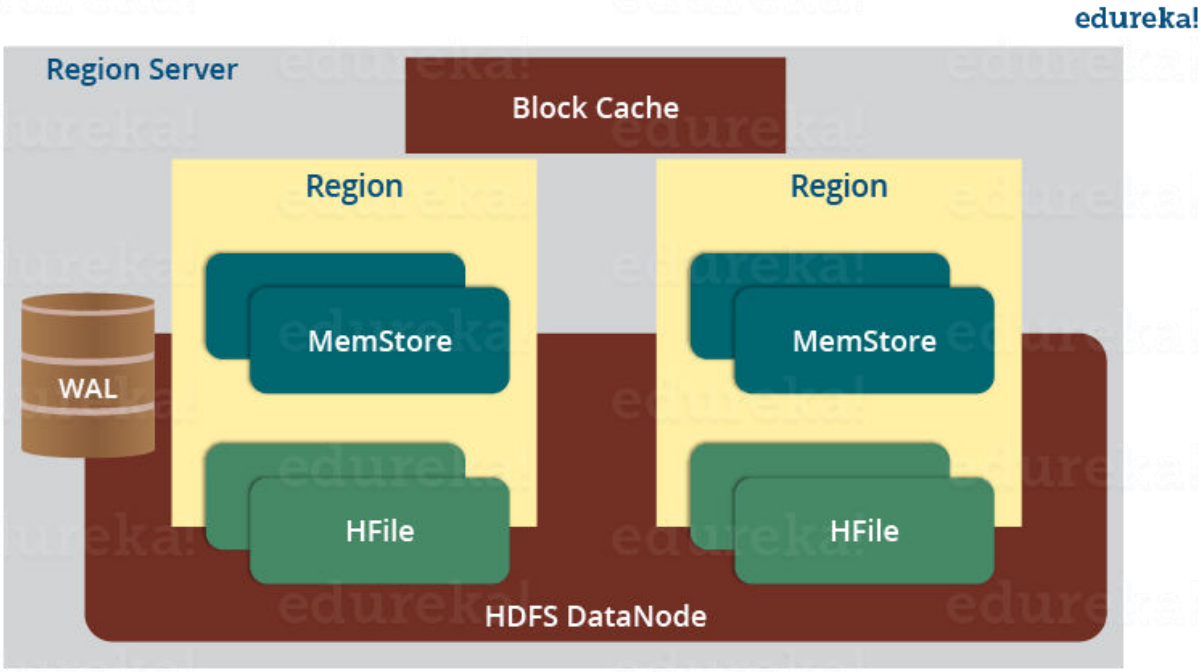


Figure: Region Server Components

A Region Server maintains various regions running on the top of ***HDFS***. Components of a Region Server are:

- **WAL:** As you can conclude from the above image, Write Ahead Log (WAL) is a file attached to every Region Server inside the distributed environment. The WAL stores the new data that hasn't been persisted or committed to the permanent storage. It is used in case of failure to recover the data sets.
- **Block Cache:** From the above image, it is clearly visible that Block Cache resides in the top of Region Server. It stores the frequently read data in the

memory. If the data in BlockCache is least recently used, then that data is removed from BlockCache.

- **MemStore:** It is the write cache. It stores all the incoming data before committing it to the disk or permanent memory. There is one MemStore for each column family in a region. As you can see in the image, there are multiple MemStores for a region because each region contains multiple column families. The data is sorted in lexicographical order before committing it to the disk.
- **HFile:** From the above figure you can see HFile is stored on HDFS. Thus it stores the actual cells on the disk. MemStore commits the data to HFile when the size of MemStore exceeds.

Now that we know major and minor components of HBase Architecture, I will explain the mechanism and their collaborative effort in this. Whether it's reading or writing, first we need to search from where to read or where to write a file. So, let's understand this search process, as this is one of the mechanisms which makes HBase very popular.

HBase Architecture: How Search Initializes in HBase?

As you know, Zookeeper stores the META table location. Whenever a client approaches with a read or writes requests to HBase following operation occurs:

1. The client retrieves the location of the META table from the ZooKeeper.
2. The client then requests for the location of the Region Server of corresponding row key from the META table to access it. The client caches this information with the location of the META Table.
3. Then it will get the row location by requesting from the corresponding Region Server.

For future references, the client uses its cache to retrieve the location of META table and previously read row key's Region Server. Then the client will not refer to the META table, until and unless there is a miss because the region is shifted or moved. Then it will again request to the META server and update the cache.

As every time, clients does not waste time in retrieving the location of Region Server from META Server, thus, this saves time and makes the search process faster. Now, let me tell you how writing takes place in HBase. What are the components involved in it and how are they involved?

HBase Architecture: HBase Write Mechanism

This below image explains the write mechanism in HBase.

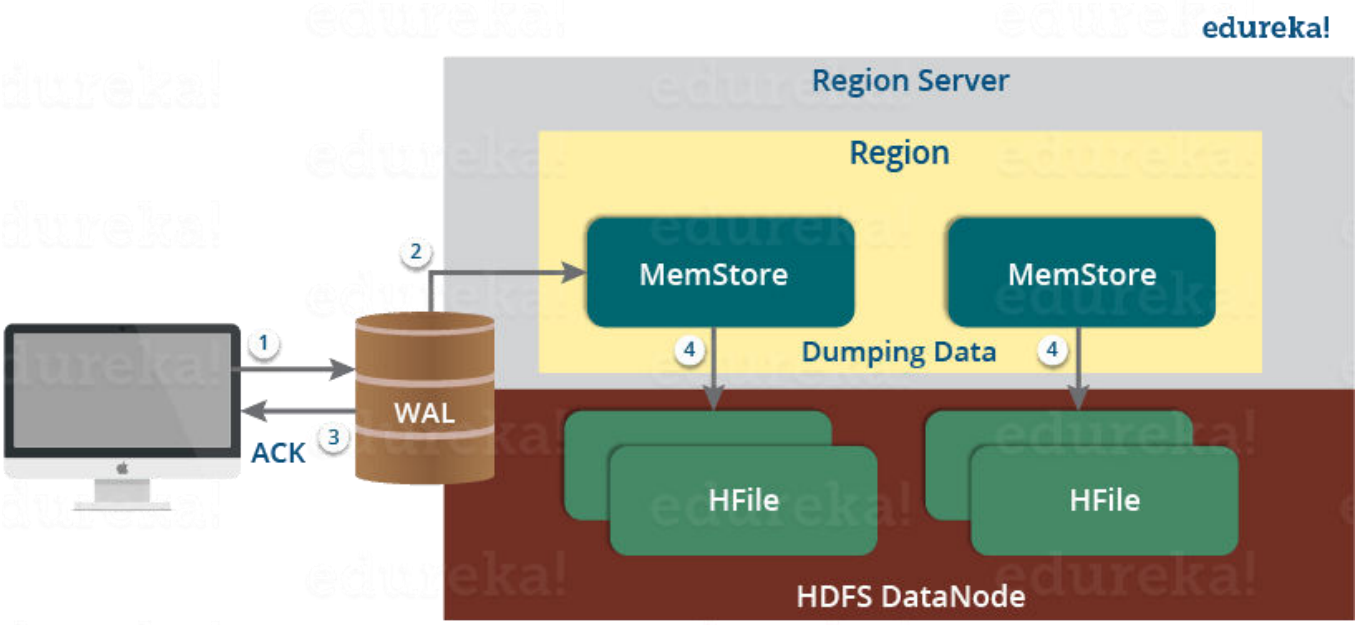


Figure: Write Mechanism in HBase

The write mechanism goes through the following process sequentially (refer to the above image):

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.

Now let us take a deep dive and understand how MemStore contributes in the writing process and what are its functions?

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 knows, 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.

As I discussed several times, that HFile is the main persistent storage in an HBase architecture. At last, all the data is committed to HFile which is the permanent storage of HBase. Hence, let us look at the properties of HFile which makes it faster for search while reading and writing.

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.

After knowing the write mechanism and the role of various components in making write and search faster. I will be explaining to you how the reading mechanism works inside an HBase architecture? Then we will move to the mechanisms which increases HBase performance like compaction, region split and recovery.

HBase Architecture: Read Mechanism

As discussed in our search mechanism, first the client retrieves the location of the Region Server from .META Server if the client does not have it in its cache memory. Then it goes through the sequential steps as follows:

- For reading the data, the scanner first looks for the Row cell in Block cache. Here all the recently read key value pairs are stored.
- If Scanner fails to find the required result, it moves to the MemStore, as we know this is the write cache memory. There, it searches for the most recently written files, which has not been dumped yet in HFile.
- At last, it will use bloom filters and block cache to load the data from HFile.

So far, I have discussed search, read and write mechanism of HBase. Now we will look at the HBase mechanism which makes search, read and write quick in HBase. First, we will understand *Compaction*, which is one of those mechanisms.

HBase Architecture: Compaction

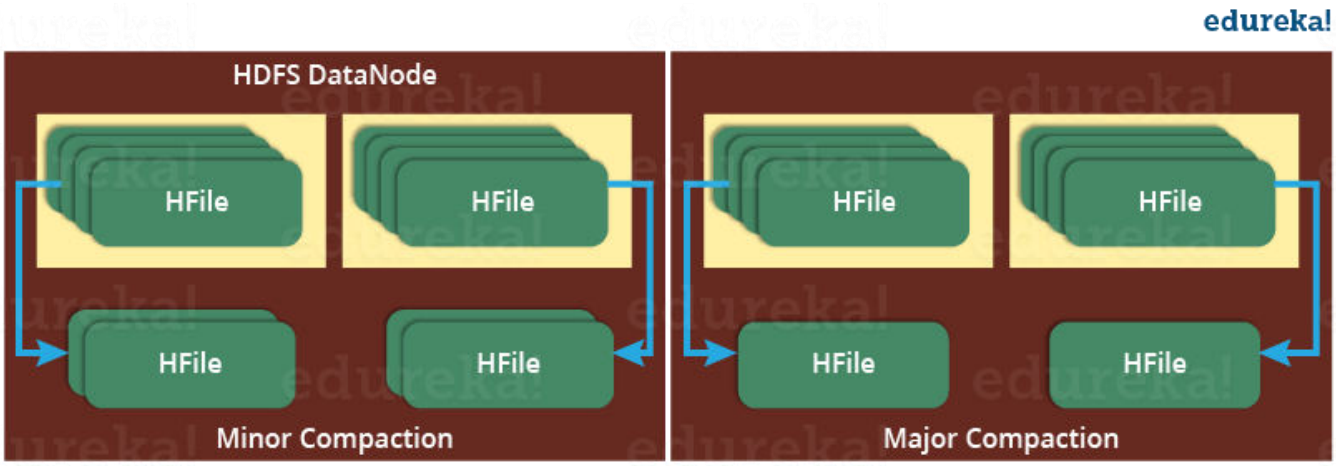


Figure: Compaction in HBase

HBase combines HFiles to reduce the storage and reduce the number of disk seeks needed for a read. This process is called**compaction**. Compaction chooses some HFiles from a region and combines them. There are two types of compaction as you can see in the above image.

1. **Minor Compaction**: HBase automatically picks smaller HFiles and recommits them to bigger HFiles as shown in the above image. This is called Minor Compaction. It performs merge sort for committing smaller HFiles to bigger HFiles. This helps in storage space optimization.
2. **Major Compaction:** As illustrated in the above image, in Major compaction, HBase merges and recommits the smaller HFiles of a region to a new HFile. In this process, the same column families are placed together in the new HFile. It drops deleted and expired cell in this process. It increases read performance.

But during this process, input-output disks and network traffic might get congested. This is known as **write amplification**. So, it is generally scheduled during low peak load timings.

Now another performance optimization process which I will discuss is *Region Split*. This is very important for load balancing.

HBase Architecture: Region Split

The below figure illustrates the Region Split mechanism.

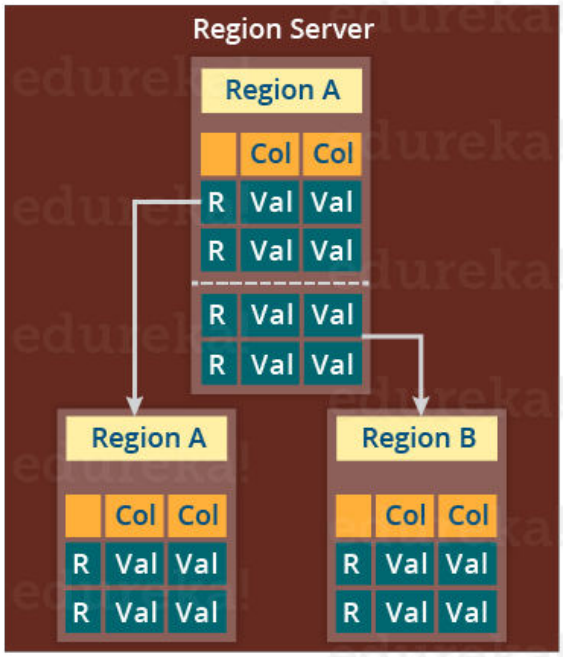


Figure: Region Split in HBase

Whenever a region becomes large, it is divided into two child regions, as shown in the above figure. Each region represents exactly a half of the parent region. Then this split is reported to the HMaster. This is handled by the same Region Server until the HMaster allocates them to a new Region Server for load balancing.

Moving down the line, last but the not least, I will explain you how does HBase recover data after a failure. As we know that**Failure Recovery** is a very important feature of HBase, thus let us know how HBase recovers data after a failure.

HBase Architecture: HBase Crash and Data Recovery

- Whenever a Region Server fails, ZooKeeper notifies to the HMaster about the failure.
- Then HMaster distributes and allocates the regions of crashed Region Server to many active Region Servers. To recover the data of the MemStore of the failed Region Server, the HMaster distributes the WAL to all the Region Servers.
- Each Region Server re-executes the WAL to build the MemStore for that failed region's column family.
- The data is written in chronological order (in a timely order) in WAL. Therefore, Re-executing that WAL means making all the change that were made and stored in the MemStore file.
- So, after all the Region Servers executes the WAL, the MemStore data for all column family is recovered.

I hope this blog would have helped you in understating the HBase Data Model & HBase Architecture. Hope you enjoyed it. Now you can relate to the features of HBase (which I explained in my previous ***HBase Tutorial*** blog) with HBase

Architecture and understand how it works internally. Now that you know the theoretical part of HBase, you should move to the practical part.

Sqoop Introduction

Generally, applications interact with relational database using RDBMS, and thus this makes relational databases one of the most important sources that generates Big Data. Such data is stored in RDB Servers in the relational structure. Here, Apache Sqoop plays an important role in ***Hadoop ecosystem***, providing feasible interaction between relational database server and HDFS.

So, Apache Sqoop is a tool in ***Hadoop ecosystem*** which is designed to transfer data between ***HDFS*** (Hadoop storage) and relational database servers like mysql, Oracle RDB, SQLite, Teradata, Netezza, Postgres etc. Apache Sqoop imports data from relational databases to HDFS, and exports data from HDFS to relational databases. It efficiently transfers bulk data between Hadoop and external datastores such as enterprise data warehouses, relational databases, etc.

*This is how Sqoop got its name – "**SQ**L to Had**oop** & Hadoop to SQL".*

Additionally, Sqoop is used to import data from external datastores into Hadoop ecosystem's tools like ***Hive*** & ***HBase***.
Now, as we know what is Apache Sqoop. So, let us advance in our Apache Sqoop tutorial and understand why Sqoop is used extensively by organisations.

Why Sqoop?

For Hadoop developer, the actual game starts after the data is being loaded in HDFS. They play around this data in order to gain various insights hidden in the data stored in HDFS.

So, for this analysis the data residing in the relational database management systems need to be transferred to HDFS. The task of writing ***MapReduce*** code for importing and exporting data from relational database to HDFS is uninteresting & tedious. This is where Apache Sqoop comes to rescue and removes their pain. It automates the process of importing & exporting the data.

Sqoop makes the life of developers easy by providing CLI for importing and exporting data. They just have to provide basic information like database authentication, source, destination, operations etc. It takes care of remaining part.

Sqoop internally converts the command into MapReduce tasks, which are then executed over HDFS. It uses YARN framework to import and export the data, which provides fault tolerance on top of parallelism.

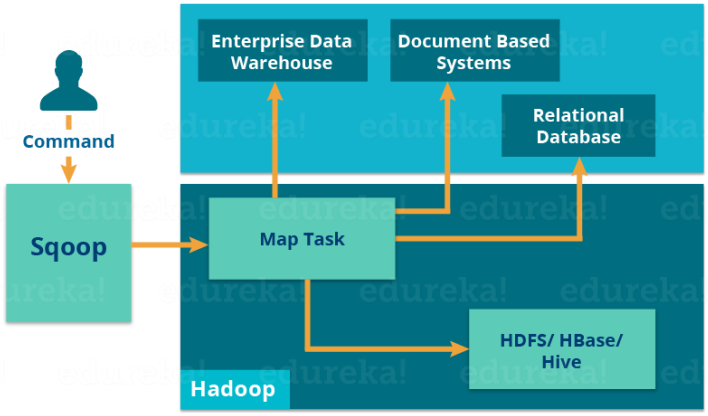
Advancing ahead in this Sqoop Tutorial blog, we will understand the the key features of Sqoop and then we will move on to the Apache Sqoop architecture.

Key Features of Sqoop
Sqoop provides many salient features like:

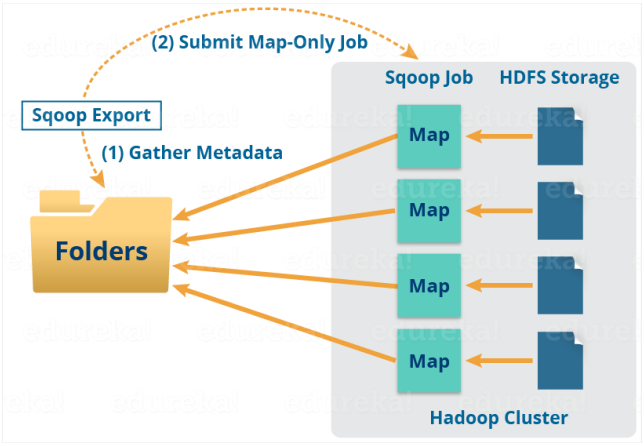
1. ***Full Load***: Apache Sqoop can load the whole table by a single command. You can also load all the tables from a database using a single command.
2. ***Incremental Load***: Apache Sqoop also provides the facility of incremental load where you can load parts of table whenever it is updated.
3. ***Parallel import/export***: Sqoop uses YARN framework to import and export the data, which provides fault tolerance on top of parallelism.
4. ***Import results of SQL query***: You can also import the result returned from an SQL query in HDFS.
5. ***Compression***: You can compress your data by using deflate(gzip) algorithm with –compress argument, or by specifying –compression-codec argument. You can also load compressed table in ***Apache Hive***.
6. ***Connectors for all major RDBMS Databases***: Apache Sqoop provides connectors for multiple RDBMS databases, covering almost the entire circumference.
7. ***Kerberos Security Integration***: Kerberos is a computer network authentication protocol which works on the basis of 'tickets' to allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner. Sqoop supports Kerberos authentication.
8. ***Load data directly into HIVE/HBase***: You can load data directly into ***Apache Hive*** for analysis and also dump your data in HBase, which is a NoSQL database.
9. ***Support for Accumulo***: You can also instruct Sqoop to import the table in Accumulo rather than a directory in HDFS.

The architecture is one which is empowering Apache Sqoop with these benefits. Now, as we know the features of Apache Sqoop, lets move ahead and understand Apache Sqoop's architecture & working.

Apache Sqoop Tutorial: Sqoop Architecture & Working
Let us understand how Apache Sqoop works using the below diagram:



The import tool imports individual tables from RDBMS to HDFS. Each row in a table is treated as a record in HDFS.
When we submit Sqoop command, our main task gets divided into sub tasks which is handled by individual Map Task internally. Map Task is the sub task, which imports part of data to the Hadoop Ecosystem. Collectively, all Map tasks imports the whole data.



Export also works in a similar manner.

The export tool exports a set of files from HDFS back to an RDBMS. The files given as input to Sqoop contain records, which are called as rows in table.

When we submit our Job, it is mapped into Map Tasks which brings the chunk of data from HDFS. These chunks are exported to a structured data destination. Combining all these exported chunks of data, we receive the whole data at the destination, which in most of the cases is an RDBMS (MYSQL/Oracle/SQL Server).

Reduce phase is required in case of aggregations. But, Apache Sqoop just imports and exports the data; it does not perform any aggregations. Map job launch multiple mappers depending on the number defined by user. For Sqoop import, each mapper task will be assigned with a part of data to be imported. Sqoop distributes the input data among the mappers equally to get high performance. Then each mapper creates connection with the database using JDBC and fetches the part of data assigned by Sqoop and writes it into HDFS or Hive or HBase based on the arguments provided in the CLI.

Now that we understand the architecture and working of Apache Sqoop, lets understand the difference between Apache Flume and Apache Sqoop.

Apache Sqoop Tutorial: Flume vs Sqoop

The major difference between Flume and Sqoop is that:

- Flume only ingests unstructured data or semi-structured data into HDFS.
- While Sqoop can import as well as export structured data from RDBMS or Enterprise data warehouses to HDFS or vice versa.

Now, advancing in our Apache Sqoop Tutorial it is the high time to go through Apache Sqoop commands.

Apache Sqoop Tutorial: Sqoop Commands

1. Specify the source connection information.

First, you must specify the:

- database URI (`db.foo.com` in the following example)
- database name (`bar`)
- connection protocol (`jdbc:mysql:`)

For this example, use the following command:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES
```

If the source database requires credentials, such as a username and password, you can enter the password on the command line or specify a file where the password is stored.

For example:

- Enter the password on the command line:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --username <username> -P
Enter password: (hidden)
```

- Specify a file where the password is stored:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --username <username> --password-file
${user.home}/.password
```

2. **Specify the data and the parallelism for import**:

a. Specify the data simply.

Sqoop provides flexibility to specify exactly the data you want to import from the source system:

- Import an **entire table**:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES
```

- Import a **subset of the columns** from a table:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --columns
"employee_id,first_name,last_name,job_title"
```

- Import only the **latest records** by specifying them with a WHERE clause and then that they be appended to an existing table:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES  --where "start_date > '2010-01-01'"
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --where "id > 100000" --target-dir
/incremental_dataset --append
```

You can also use a free-form SQL statement.

b. Specify parallelism.

There are three options for specifying *write parallelism* (number of map tasks):

i. Explicitly set the number of mappers using `--num-mappers`. Sqoop evenly splits the primary key range of the source table:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --num-mappers 8
```

In this scenario, the source table must have a primary key.

ii. Provide an alternate split key using `--split-by`. This evenly splits the data using the alternate split key instead of a primary key:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --split-by dept_id
```

This method is useful if primary keys are not evenly distributed.

iii. When there is not split key or primary key, the data import must be sequential. Specify a single mapper by using `--num-mappers 1` or `--autoreset-to-one-mapper`.

b. Specify the data using a query.

Instead of specifying a particular table or columns, you can specify the date with a query. You can use one of the following options:

i. Explicitly specify a *split-by column* using `--split-by` and put `$ CONDITIONS` that Sqoop replaces with range conditions based on the split-by key. This method requires a target directory:

ii.
```
sqoop import --query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id)
WHERE $CONDITIONS' --split-by a.id --target-dir /user/foo/joinresults
```

iii. Use sequential import if you cannot specify a split-by column:

iv.
```
sqoop import --query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id)
WHERE $CONDITIONS' -m 1 --target-dir /user/foo/joinresults
```

To try a sample query without importing data, use the `eval` option to print the results to the command prompt:

```
sqoop eval --connect jdbc:mysql://db.foo.com/bar --query "SELECT * FROM employees LIMIT 10"
```

c. **Specify the destination for the data: HDFS or Hive**.
Here is an example of specifying the HDFS target directory:

```
sqoop import --query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id)

WHERE $CONDITIONS' --split-by a.id --target-dir /user/foo/joinresults
```

If you can add text data into your Hive table, you can specify that the data be directly added to Hive. Using `--hive-import` is the primary method to add text data directly to Hive:

```
sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES --hive-import
```

This method creates a metastore schema after storing the text data in HDFS.
If you have already moved data into HDFS and want to add a schema, use the `create-hive-table` Sqoop command:

```
sqoop create-hive-table (generic-args) (create-hive-table-args)
```

Additional options for importing data into Hive with Sqoop:

Table 2.12. Sqoop Command Options for Importing Data into Hive

Sqoop Command Option	Description
`--hive-home <directory>`	Overrides `$HIVE_HOME`.
`--hive-import`	Imports tables into Hive using Hive's default delimiters if none are explicitly set.
`--hive-overwrite`	Overwrites existing data in the Hive table.
`--create-hive-table`	Creates a hive table during the operation. If this option is set and the Hive table already exists, the job will fail. Set to `false` by default.
`--hive-table <table_name>`	Specifies the table name to use when importing data into Hive.
`--hive-drop-import-delims`	Drops the delimiters `\n`, `\r`, and `\01` from string fields when importing data into Hive.
`--hive-delims-replacement`	Replaces the delimiters `\n`, `\r`, and `\01` from strings fields with a user-defined string when importing data into Hive.
`--hive-partition-key`	Specifies the name of the Hive field on which a sharded database is partitioned.
`--hive-partition-value <value>`	A string value that specifies the partition key for data imported into Hive.
`--map-column-hive <map>`	Overrides the default mapping from SQL type to Hive type for configured columns.

Kafka

In Big Data, an enormous volume of data is used. Regarding data, we have two main challenges.The first challenge is how to collect large volume of data and the second challenge is to analyze the collected data. To overcome those challenges, you must need a messaging system.
Kafka is designed for distributed high throughput systems.

Kafka tends to work very well as a replacement for a more traditional message broker. In comparison to other messaging systems, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications.

What is a Messaging System?

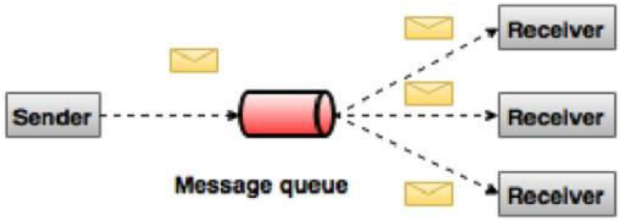
A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it. Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging system. Two types of messaging patterns are available − one is point to point and the other is publish-subscribe (pub-sub) messaging system. Most of the messaging patterns follow **pub-sub**.

Point to Point Messaging System

In a point-to-point system, messages are persisted in a queue. One or more consumers can consume the messages in the queue, but a particular message can be consumed by a maximum of one consumer only. Once a consumer reads a message in the queue, it disappears from that queue. The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time. The following diagram depicts the structure.

Publish-Subscribe Messaging System

In the publish-subscribe system, messages are persisted in a topic. Unlike point-to-point system, consumers can subscribe to one or more topic and consume all the messages in that topic. In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers. A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



What is Kafka?

Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.

Benefits

Following are a few benefits of Kafka –

- **Reliability** − Kafka is distributed, partitioned, replicated and fault tolerance.
- **Scalability** − Kafka messaging system scales easily without down time..
- **Durability** − Kafka uses Distributed commit log which means messages persists on disk as fast as possible, hence it is durable..

- **Performance** − Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even many TB of messages are stored.

Kafka is very fast and guarantees zero downtime and zero data loss.

Use Cases

Kafka can be used in many Use Cases. Some of them are listed below –

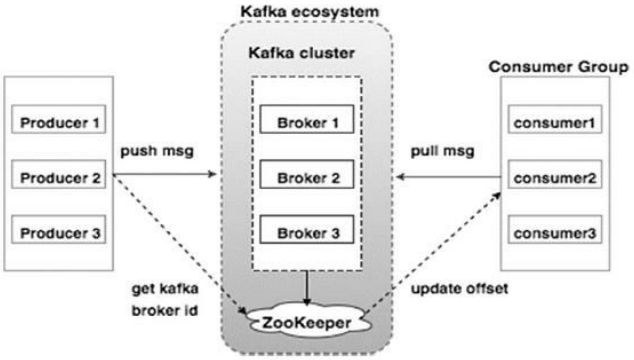
- **Metrics** − Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.
- **Log Aggregation Solution** − Kafka can be used across an organization to collect logs from multiple services and make them available in a standard format to multiple con-sumers.
- **Stream Processing** − Popular frameworks such as Storm and Spark Streaming read data from a topic, processes it, and write processed data to a new topic where it becomes available for users and applications. Kafka's strong durability is also very useful in the context of stream processing.

Need for Kafka

Kafka is a unified platform for handling all the real-time data feeds. Kafka supports low latency message delivery and gives guarantee for fault tolerance in the presence of machine failures. It has the ability to handle a large number of diverse consumers. Kafka is very fast, performs 2 million writes/sec. Kafka persists all data to the disk, which essentially means that all the writes go to the page cache of the OS (RAM). This makes it very efficient to transfer data from page cache to a network socket.

Apache Kafka - Cluster Architecture

Take a look at the following illustration. It shows the cluster diagram of Kafka.



The following table describes each of the components shown in the above diagram.

S.No	Components and Description
1	**Broker** Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each bro-ker can handle TB of messages without performance impact. Kafka broker leader election can be done by ZooKeeper.
2	**ZooKeeper**

	ZooKeeper is used for managing and coordinating Kafka broker. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then pro-ducer and consumer takes decision and starts coordinating their task with some other broker.
3	**Producers** Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn't wait for acknowledgements from the broker and sends messages as fast as the broker can handle.
4	**Consumers** Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages. The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by ZooKeeper.

As of now, we discussed the core concepts of Kafka. Let us now throw some light on the workflow of Kafka.
Kafka is simply a collection of topics split into one or more partitions. A Kafka partition is a linearly ordered sequence of messages, where each message is identified by their index (called as offset). All the data in a Kafka cluster is the disjointed union of partitions. Incoming messages are written at the end of a partition and messages are sequentially read by consumers. Durability is provided by replicating messages to different brokers.

Kafka provides both pub-sub and queue based messaging system in a fast, reliable, persisted, fault-tolerance and zero downtime manner. In both cases, producers simply send the message to a topic and consumer can choose any one type of messaging system depending on their need. Let us follow the steps in the next section to understand how the consumer can choose the messaging system of their choice.

Workflow of Pub-Sub Messaging

Following is the step wise workflow of the Pub-Sub Messaging –

- Producers send message to a topic at regular intervals.
- Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.
- Consumer subscribes to a specific topic.
- Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.
- Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.
- Once Kafka receives the messages from producers, it forwards these messages to the consumers.
- Consumer will receive the message and process it.
- Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.
- Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages.
- This above flow will repeat until the consumer stops the request.
- Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

Workflow of Queue Messaging / Consumer Group

In a queue messaging system instead of a single consumer, a group of consumers having the same Group ID will subscribe to a topic. In simple terms, consumers subscribing to a topic with same Group ID are considered as a single group and the messages are shared among them. Let us check the actual workflow of this system.

- Producers send message to a topic in a regular interval.
- Kafka stores all messages in the partitions configured for that particular topic similar to the earlier scenario.

- A single consumer subscribes to a specific topic, assume Topic-01with Group ID as Group-1.
- Kafka interacts with the consumer in the same way as Pub-Sub Messaging until new consumer subscribes the same topic, Topic-01with the same Group ID as Group-1.
- Once the new consumer arrives, Kafka switches its operation to share mode and shares the data between the two consumers. This sharing will go on until the number of con-sumers reach the number of partition configured for that particular topic.
- Once the number of consumer exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing consumer unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition and once all the partitions are assigned to the existing consumers, the new consumers will have to wait.
- This feature is also called as Consumer Group. In the same way, Kafka will provide the best of both the systems in a very simple and efficient manner.

Role of ZooKeeper

A critical dependency of Apache Kafka is Apache Zookeeper, which is a distributed configuration and synchronization service. Zookeeper serves as the coordination interface between the Kafka brokers and consumers. The Kafka servers share information via a Zookeeper cluster. Kafka stores basic metadata in Zookeeper such as information about topics, brokers, consumer offsets (queue readers) and so on.

Since all the critical information is stored in the Zookeeper and it normally replicates this data across its ensemble, failure of Kafka broker / Zookeeper does not affect the state of the Kafka cluster. Kafka will restore the state, once the Zookeeper restarts. This gives zero downtime for Kafka. The leader election between the Kafka broker is also done by using Zookeeper in the event of leader failure.

What is Apache Oozie?

Apache Oozie is a scheduler system to run and **manage Hadoop jobs** in a distributed environment. It allows to combine multiple complex jobs to be run in a sequential order to achieve a bigger task. Within a sequence of task, two or more jobs can also be programmed to run parallel to each other.

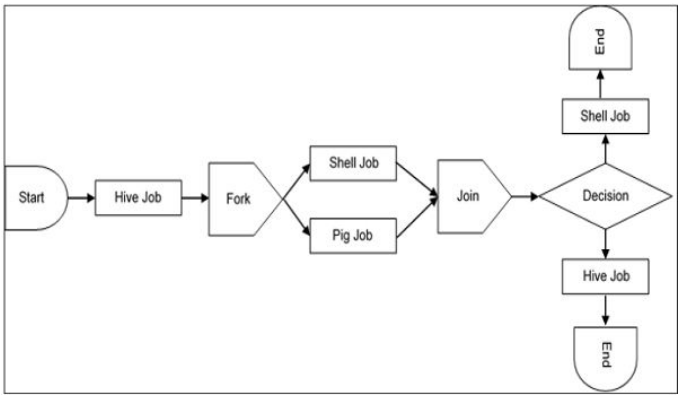
One of the main advantages of Oozie is that it is tightly integrated with Hadoop stack supporting various Hadoop jobs like **Hive, Pig, Sqoop** as well as system-specific jobs like **Java and Shell**.

Oozie is an **Open Source Java Web-Application** available under Apache license 2.0. It is responsible for triggering the workflow actions, which in turn uses the Hadoop execution engine to actually execute the task. Hence, Oozie is able to leverage the existing Hadoop machinery for load balancing, fail-over, etc.

Oozie detects completion of tasks through callback and polling. When Oozie starts a task, it provides a unique **callback HTTP URL** to the task, and notifies that URL when it is complete. If the task fails to invoke the callback URL, Oozie can poll the task for completion.
Following three types of jobs are common in Oozie –

- **Oozie Workflow Jobs** − These are represented as Directed Acyclic Graphs (DAGs) to specify a sequence of actions to be executed.
- **Oozie Coordinator Jobs** − These consist of workflow jobs triggered by time and data availability.
- **Oozie Bundle** − These can be referred to as a package of multiple coordinator and workflow jobs.

We will look into each of these in detail in the following chapters.
A sample workflow with Controls (Start, Decision, Fork, Join and End) and Actions (Hive, Shell, Pig) will look like the following diagram



Workflow will always start with a Start tag and end with an End tag.

Use-Cases of Apache Oozie

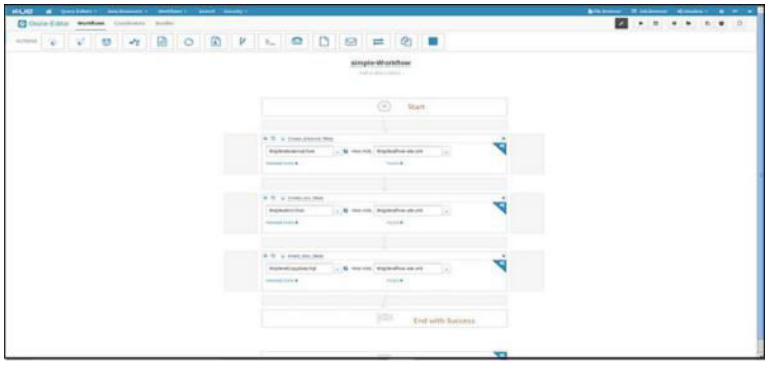
Apache Oozie is used by Hadoop system administrators to run complex log analysis on **HDFS**. Hadoop Developers use Oozie for performing ETL operations on data in a sequential order and saving the output in a specified format (Avro, ORC, etc.) in HDFS.
In an enterprise, Oozie jobs are scheduled as coordinators or bundles.

Oozie Editors

Before we dive into Oozie lets have a quick look at the available editors for Oozie.
Most of the time, you won't need an editor and will write the workflows using any popular text editors (like Notepad++, Sublime or Atom) as we will be doing in this tutorial.
But as a beginner it makes some sense to create a workflow by the drag and drop method using the editor and then see how the workflow gets generated. Also, to map **GUI** with the actual **workflow.xml** created by the editor. This is the only section where we will discuss about Oozie editors and won't use it in our tutorial.
The most popular among Oozie editors is **Hue**.

Hue Editor for Oozie

This editor is very handy to use and is available with almost all Hadoop vendors' solutions.
The following screenshot shows an example workflow created by this editor.



You can drag and drop controls and actions and add your job inside these actions.

Oozie Workflow

Workflow is a sequence of actions arranged in a Direct Acyclic Graph (DAG). The actions are dependent on one another, as the next action can only be executed after the output of current action. A workflow action can be a Pig action, Hive action, MapReduce action, Shell action, Java action etc. There can be decision trees to decide how and on which condition a job should run.
We can create different types of actions based on the job and each type of action can have its own type of tags. The workflow and the scripts or jars should be placed in HDFS path before executing the workflow.

Command: *oozie job –oozie http://localhost:11000/oozie -config job.properties -run*

For checking the status of job, you can go to Oozie web console, i.e. *http://host_name:11000*. By clicking on the job you will see the status of the job.

In scenarios, where we want to run multiple jobs parallelly, we can use *Fork*. Whenever we use fork, we have to use Join as an end node to fork. For each fork there should be a join. Join assumes that all the nodes executing parallelly, are a child of a single fork. For example, we can create two tables at the same time parallelly.

If we want to run an action based on the output of decision, we can add decision tags. For example, if we already have the hive table we won't need to create it again. In that situation, we can add a decision tag to not run the create table steps if the table already exists. Decision nodes have a switch tag similar to switch case.

The value of job-tracker, name-node, script and param can be passed directly. But, this becomes hard to manage. This is where a config file (i.e. .property file) comes handy.

Apache Oozie Tutorial: Oozie Coordinator

You can schedule complex workflows as well as workflows that are scheduled regularly using Coordinator. Oozie Coordinators triggers the workflows jobs based on time, data or event predicates. The workflows inside the job coordinator starts when the given condition is satisfied.
Definitions required for the coordinator jobs are:

- **start**− Start datetime for the job.
- **end**− End datetime for the job.
- **timezone**− Timezone of the coordinator application.
- **frequency**− The frequency, in minutes, for executing the jobs.

Some more properties are available for Control Information:

- **timeout**− The maximum time, in minutes, for which an action will wait to satisfy the additional conditions, before getting discarded. 0 indicates that if all the input events are not satisfied at the time of action materialization, the action should timeout immediately. -1 indicates no timeout, the action will wait forever. The default value is -1.
- **concurrency**− The maximum number of actions for a job that can run parallely. The default value is 1.
- **execution**– It specifies the execution order if multiple instances of the coordinator job have satisfied their execution criteria. It can be:
    -
    - FIFO (default)
    - LIFO
    - LAST_ONLY

Command: *oozie job –oozie http://localhost:11000/oozie -config <path to coordinator.properties file> -run*

If a configuration property used in the definition is not provided with the job configuration while submitting the coordinator job, the job submission will fail.

Apache Oozie Tutorial: Oozie Bundle

Oozie Bundle system allows you to define and execute a set of coordinator applications, often called a data pipeline. In a Oozie bundle, there is no explicit dependency among the coordinator applications. However, you could use the data dependency of coordinator applications to create an implicit data application pipeline. You can start/stop/suspend/resume/rerun the bundle. It gives a better and easy operational control.

Kick-off-time − The time when a bundle should start and submit coordinator applications.

How does OOZIE work?

Oozie runs as a service in the cluster and clients submit workflow definitions for immediate or later processing.

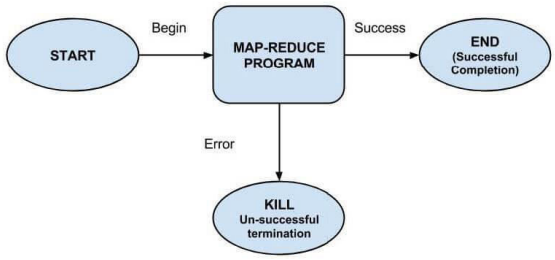
Oozie workflow consists of **action nodes** and **control-flow nodes**.

An **action node** represents a workflow task, e.g., moving files into HDFS, running a MapReduce, Pig or Hive jobs, importing data using Sqoop or running a shell script of a program written in Java.

A **control-flow node** controls the workflow execution between actions by allowing constructs like conditional logic wherein different branches may be followed depending on the result of earlier action node.

Start Node, **End Node** and **Error Node** fall under this category of nodes.
Start Node, designates start of the workflow job.
End Node, signals end of the job.
Error Node, designates an occurrence of error and corresponding error message to be printed.
At the end of execution of workflow, HTTP callback is used by Oozie to update client with the workflow status. Entry-to or exit-from an action node may also trigger callback.

Example Workflow Diagram



Cluster Management

Add Hosts Wizard

1. Click the **Hosts** tab.
2. Click the **Add New Hosts** button.
3. Follow the instructions in the wizard to install the Oracle JDK and Cloudera Manager Agent packages and start the Agent.
4. In the **Specify hosts for your CDH Cluster installation** page, you can search for new hosts to add under the **New Hosts** tab. However, if you have hosts that are already known to Cloudera Manager but have no roles assigned, (for example, a host that was previously in your cluster but was then removed) these will appear under the **Currently Managed Hosts** tab.
5. You will have an opportunity to add (and start) role instances to your newly-added hosts using a host template.
    a. You can select an existing host template, or create a new one.
    b. To create a new host template, click the **+ Create...** button. This will open the **Create New Host Template** pop-up. See Host Templates for details on how you select the role groups that define the roles that should run on a host. When you have created the template, it will appear in the list of host templates from which you can choose.
    c. Select the host template you want to use.
    d. By default Cloudera Manager will automatically start the roles specified in the host template on your newly added hosts. To prevent this, uncheck the option to start the newly-created roles.
6. When the wizard is finished, you can verify the Agent is connecting properly with the Cloudera Manager Server by clicking the **Hosts** tab and checking the health status for the new host. If the Health Status is **Good** and the value for the Last Heartbeat is recent, then the Agent is connecting properly with the Cloudera Manager Server.

If you did not specify a host template during the Add Hosts wizard, then no roles will be present on your new hosts until you add them. You can do this by adding individual roles under the **Instances** tab for a specific service, or by using a host template. See Role Instances for information about adding roles for a specific service. See Host Templates to create a host template that specifies a set of roles (from different services) that should run on a host.

Deleting Hosts

Minimum Required Role: Full Administrator

You can remove a host from a cluster in two ways:

- Delete the host entirely from Cloudera Manager.
- Remove a host from a cluster, but leave it available to other clusters managed by Cloudera Manager.

Both methods decommission the hosts, delete roles, and remove managed service software, but preserve data directories.

Deleting a Host from Cloudera Manager

1. In the Cloudera Manager Admin Console, click the **Hosts** tab.
2. Select the hosts to delete.
3. Select **Actions for Selected** > **Decommission**.
4. Stop the Agent on the host. For instructions, see **Starting, Stopping, and Restarting Cloudera Manager Agents.**
5. In the Cloudera Manager Admin Console, click the **Hosts** tab.
6. Reselect the hosts you selected in **step 2.**
7. Select **Actions for Selected** > **Delete**.

Removing a Host From a Cluster

This procedure leaves the host managed by Cloudera Manager and preserves the Cloudera Management Service roles (such as the Events Server, Activity Monitor, and so on).

1. In the Cloudera Manager Admin Console, click the **Hosts** tab.
2. Select the hosts to delete.
3. Select **Actions for Selected** > **Remove From Cluster**. The Remove Hosts From Cluster dialog box displays.
4. Leave the selections to decommission roles and skip removing the Cloudera Management Service roles. Click **Confirm** to proceed with removing the selected hosts.

Decommissioning Hosts

You cannot decommission a DataNode or a host with a DataNode if the number of DataNodes equals the replication factor (which by default is three) of any file stored in HDFS. For example, if the replication factor of any file is three, and you have three DataNodes, you cannot decommission a DataNode or a host with a DataNode. If you attempt to decommission a DataNode or a host with a DataNode in such situations, the DataNode will be decommissioned, but the decommission process will not complete. You will have to abort the decommission and recommission the DataNode.
To decommission hosts:

1. If the host has a DataNode, perform the steps in Tuning HDFS Prior to Decommissioning DataNodes.
2. Click the **Hosts** tab.
3. Select the checkboxes next to one or more hosts.
4. Select **Actions for Selected > Hosts Decommission**.
   A confirmation pop-up informs you of the roles that will be decommissioned or stopped on the hosts you have selected.
5. Click **Confirm**. A Decommission Command pop-up displays that shows each step or decommission command as it is run, service by service. In the Details area, click to see the subcommands that are run for decommissioning a given service. Depending on the service, the steps may include adding the host to an "exclusions list" and refreshing the NameNode, JobTracker, or NodeManager; stopping the Balancer (if it is running); and moving data blocks or regions. Roles that do not have specific decommission actions are stopped.
   You can abort the decommission process by clicking the **Abort** button, but you must recommission and restart each role that has been decommissioned.
   The Commission State facet in the Filters lists displays Decommissioning while decommissioning is in progress, and Decommissioned when the decommissioning process has finished. When the process is complete, a is added in front of Decommission Command.

You cannot start roles on a decommissioned host.

Recommissioning Hosts

Minimum Required Role: Operator (also provided by **Configurator**, **Cluster Administrator**, **Full Administrator**)

Only hosts that are decommissioned using Cloudera Manager can be recommissioned.

1. Click the **Hosts** tab.
2. Select one or more hosts to recommission.
3. Select **Actions for Selected > Recommission** and **Confirm**. A Recommission Command pop-up displays that shows each step or recommission command as it is run. When the process is complete, a is added in front of Recommission Command. The host and roles are marked as commissioned, but the roles themselves are not restarted.

HDFS Balancers

HDFS data might not always be distributed uniformly across DataNodes. One common reason is addition of new DataNodes to an existing cluster. HDFS provides a balancer utility that analyzes block placement and balances data across the DataNodes. The balancer moves blocks until the cluster is deemed to be balanced, which means that the utilization of every DataNode (ratio of used space on the node to total capacity of the node) differs from the utilization of the cluster (ratio of used space on the cluster to total capacity of the cluster) by no more than a given threshold percentage. The balancer does not balance between individual volumes on a single DataNode.

Configuring the Balancer Threshold

The Balancer has a default threshold of 10%, which ensures that disk usage on each DataNode differs from the overall usage in the cluster by no more than 10%. For example, if overall usage across all the DataNodes in the cluster is 40% of the cluster's total disk-storage capacity, the script ensures that DataNode disk usage is between 30% and 50% of the DataNode disk-storage capacity. To change the threshold:

1. Go to the HDFS service.
2. Click the **Configuration** tab.
3. Select **Scope > Balancer**.
4. Select **Category > Main**.
5. Set the **Rebalancing Threshold** property.
6. Click **Save Changes** to commit the changes.

Running the Balancer

1. Go to the HDFS service.
2. Ensure the service has a Balancer role.
3. Select **Actions > Rebalance**.
4. Click **Rebalance** to confirm. If you see a **Finished** status, the Balancer ran successfully.

Configuring and Running the HDFS Balancer Using the Command Line

The HDFS balancer re-balances data across the DataNodes, moving blocks from overutilized to underutilized nodes. As the system administrator, you can run the balancer from the command-line as necessary -- for example, after adding new DataNodes to the cluster.

Points to note:

- The balancer requires the capabilities of an HDFS superuser (for example, the `hdfs` user) to run.
- The balancer does not balance between individual volumes on a single DataNode.
- You can run the balancer without parameters, as follows:

```
sudo -u hdfs hdfs balancer
```

You can run the script with a different threshold; for example:

```
sudo -u hdfs hdfs balancer -threshold 5
```

Kerberos

Strongly authenticating and establishing a user's identity is the basis for secure access in Hadoop. Users need to be able to reliably "identify" themselves and then have that identity propagated throughout the Hadoop cluster. Once this is done, those users can access resources (such as files or directories) or interact with the cluster (like running MapReduce jobs). Besides users, Hadoop cluster resources themselves (such as Hosts and Services) need to authenticate with each other to avoid potential malicious systems or daemon's "posing as" trusted components of the cluster to gain access to data.

Hadoop uses Kerberos as the basis for strong authentication and identity propagation for both user and services. Kerberos is a third party authentication mechanism, in which users and services rely on a third party - the Kerberos server - to authenticate each to the other. The Kerberos server itself is known as the **Key Distribution Center**, or **KDC**. At a high level, it has three parts:

- A database of the users and services (known as **principals**) that it knows about and their respective Kerberos passwords
- An **Authentication Server** (**AS**) which performs the initial authentication and issues a **Ticket Granting Ticket** (**TGT**)
- A **Ticket Granting Server** (**TGS**) that issues subsequent service tickets based on the initial **TGT**

A **user principal** requests authentication from the AS. The AS returns a TGT that is encrypted using the user principal's Kerberos password, which is known only to the user principal and the AS. The user principal decrypts the TGT locally using its Kerberos password, and from that point forward, until the ticket expires, the user principal can use the TGT to get service tickets from the TGS. Service tickets are what allow a principal to access various services.

Because cluster resources (hosts or services) cannot provide a password each time to decrypt the TGT, they use a special file, called a **keytab**, which contains the resource principal's authentication credentials. The set of hosts, users, and services over which the Kerberos server has control is called a **realm**.

Terminology

Term	Description
Key Distribution Center, or KDC	The trusted source for authentication in a Kerberos-enabled environment.
Kerberos KDC Server	The machine, or server, that serves as the Key Distribution Center (KDC).
Kerberos Client	Any machine in the cluster that authenticates against the KDC.
Principal	The unique name of a user or service that authenticates against the KDC.
Keytab	A file that includes one or more principals and their keys.
Realm	The Kerberos network that includes a KDC and a number of Clients.
KDC Admin Account	An administrative account used by Ambari to create principals and generate keytabs in the KDC.

Kerberos Principals

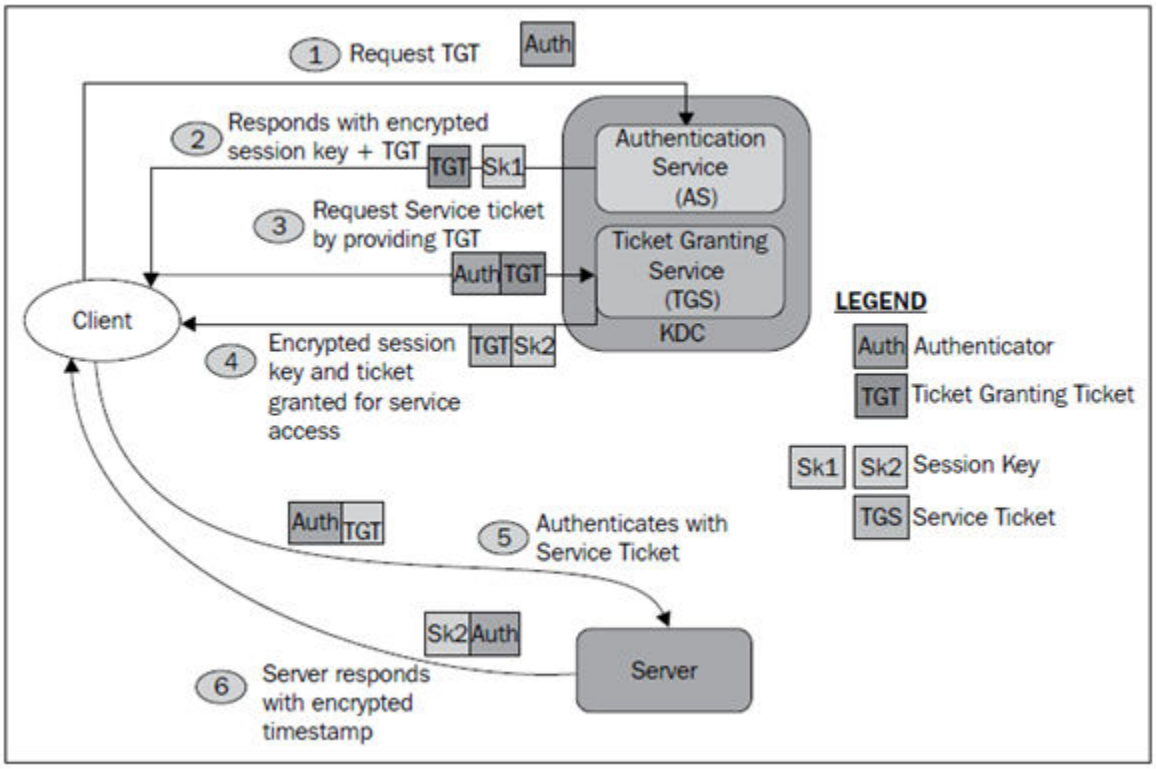
Each service and sub-service in Hadoop must have its own principal. A **principal** name in a given realm consists of a primary name and an instance name, in this case the instance name is the FQDN of the host that runs that service. As services do not log in with a password to acquire their tickets, their principal's authentication credentials are stored in a **keytab** file, which is extracted from the Kerberos database and stored locally in a secured directory with the service principal on the service component host.

Principal and Keytab Naming Conventions

Asset	Convention	Example
Principals	$service_component_name/$FQDN@EXAMPLE.COM	nn/c6401.ambari.apache.org@EXAMPLE.COM
Keytabs	$service_component_abbreviation.service.keytab	/etc/security/keytabs/nn.service.keytab

Notice in the preceding example the primary name for each service principal. These primary names, such as nn or hive for example, represent the NameNode or Hive service, respectively. Each primary name has appended to it the instance name, the FQDN of the host on which it runs. This convention provides a unique principal name for services that run on multiple hosts, like DataNodes and NodeManagers. Adding the host name serves to distinguish, for example, a request from DataNode A from a request from DataNode B. This is important for the following reasons:

- Compromised Kerberos credentials for one DataNode do not automatically lead to compromised Kerberos credentials for all DataNodes.
- If multiple DataNodes have exactly the same principal and are simultaneously connecting to the NameNode, and if the Kerberos authenticator being sent happens to have same timestamps, then the authentication is rejected as a replay request.



There are following things to remember

1. There are three parties involved in this process overall
    a. Client : You, who want to access FileServer (Principal)
    b. KDC (It is made of two components)
        i. Authentication Service
        ii. Ticket Granting Service
    c. FileServer : The actual resource which you want to access

2. In total 3 Secrete keys (1 for Client, 1 for File Server, 1 for KDC itself): Which never ever travels over the network.
    a. Client key resides on client machine as well as KDC
    b. Server Key resides on the Server machine as well as KDC
    c. KDC key resides only on KDC machine

	Client Machine	File Server Machine	KDC Machine
Client Key	Yes		Yes
Server Key		Yes	Yes
KDC Key			Yes

3. Total 2 Session keys, will be generated during the process and valid only for 8 hours session. (They will travel over the network and data is encrypted by these keys when communication happens between client and KDC ,client and File Server).
    a. Client and KDC communication (Encrypted by Session Key 1)
    b. Client and FileServer communication (Encrypted by Session key2)

How overall process works

1. Client/You want to access the FileServer in the network, but you are not authenticated user at first.

2. You will send a request for getting “Ticket Granting Ticket” from the KDC.
   a. While sending the request your message will be encrypted by the Client Secrete key which, is only available with you and KDC.
   b. KDC checks its database whether you are real user or not and find the secrete key to decrypt your message.

3. Now KDC will create TGT (Ticket Granting Ticket for you, this TGT is encrypted by KDC key and owned by KDC only) and a Session Key (S1).
   a. Bundle Both TGT+S1 and encrypt the bundle with user key and send to the client.

4. Now at client side, this bundle will be decrypted using Client Key. However, TGT cannot be decrypted because it is encrypted by KDC Key.

5. Client will have to request the Service Ticket, so it can access the FileServer
   a. Client create a Authenticator object and encrypt it with Session Key1
   b. Bundle the TGT+Authnticator+reuest for FileServer and send to the KDC

6. Now at KDC side, as it checks the bundle and find out that
   a. TGT was not touched/altered in between great
   b. Using session key (S1) decrypt the Authnticator and verify the client, great
   c. And see that user need access to FileServer

7. KDC will create a Service Ticket (Encrypted by Server Key, which is owned by only FileServer and KDC) and another session key S2. The bundle now contains the (Service Ticket + Session Key 2)
   a. This bundle is encrypted by S1 (First Session key) and sent to the client,

8. At client side bundle will be decrypted using first Session key S1
   a. Service Ticket (Can not be decrypted by client as it is owned by FileServer and client does not have it)
   b. Session Key (S2, second one, will be used for communication between client and server)

9. Now Client has the ticket to request file server.
   a. Client Prepare a bundle (Server Ticket+Authenticator+Session Key s2)
   b. Send this bundle to File server

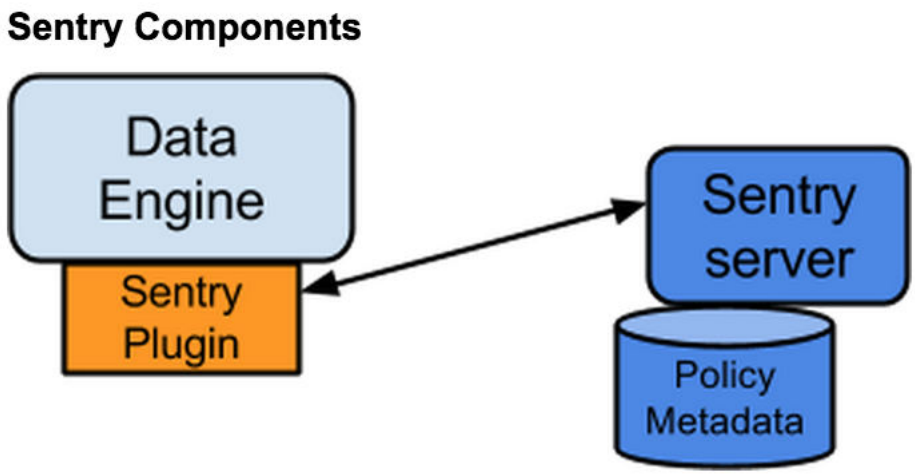
10. At server side bundle will be opened
   a. Server will check service ticket is encrypted by Server key and not impacted/touched.
   b. It will authenticate the user and send the acknowledgement to client.

11. Now client and servers are authenticated and whatever communication happens between client and server will always by encrypted using session key (S2)

Apache Sentry

Apache Sentry on the CDH platform (and Hadoop, in general), enables role-based, fine-grained authorization. Sentry can apply a range of restrictions to various tasks, such as accessing data or creating collections. These restrictions are consistently applied, regardless of the way users attempt to complete actions. For example, restricting access to data in a collection restricts that access whether queries come from the command line, from a browser, Hue, or through the admin console.

Architecture Overview



There are three components involved in the authorization process:

Sentry Server

The Sentry RPC server manages the authorization metadata. It supports interfaces to securely retrieve and manipulate the metadata.

Data Engine

This is a data processing application such as Hive or Impala that needs to authorize access to data or metadata resources. The data engine loads the Sentry plugin and all client requests for accessing resources are intercepted and routed to the Sentry plugin for validation.

Sentry Plugin

The Sentry plugin runs in the data engine. It offers interfaces to manipulate authorization metadata stored in the Sentry server, and includes the authorization policy engine that evaluates access requests using the authorization metadata retrieved from the server.

User Identity and Group Mapping

Sentry relies on underlying authentication systems such as Kerberos or LDAP to identify the user. It also uses the group mapping mechanism configured in Hadoop to ensure that Sentry sees the same group mapping as other components of the Hadoop ecosystem.

Consider users Alice and Bob who belong to an Active Directory (AD) group called finance-department. Bob also belongs to a group called finance-managers. In Sentry, you first create roles and then grant privileges to these roles. For example, you can create a role called Analyst and grant SELECT on tables Customer and Sales to this role.

The next step is to join these authentication entities (users and groups) to authorization entities (roles). This can be done by granting the Analyst role to the finance-department group. Now Bob and Alice who are members of the finance-department group get SELECT privilege to the Customer and Sales tables.

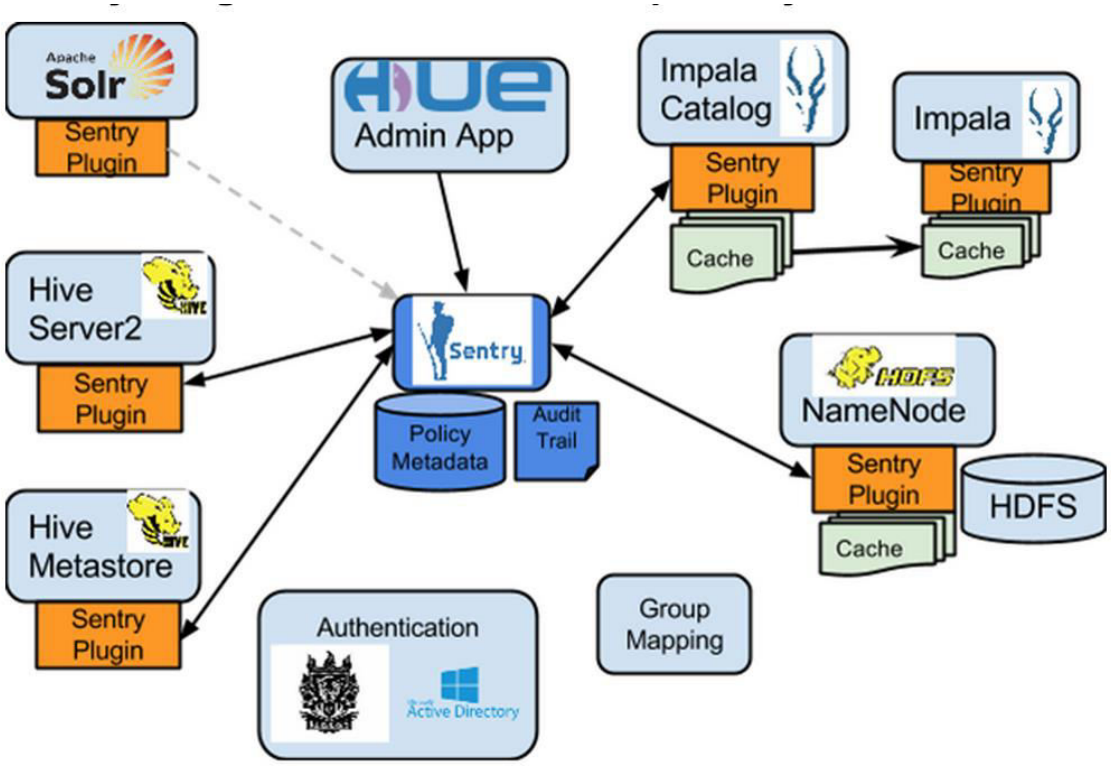
Role-Based Access Control

Role-based access control (RBAC) is a powerful mechanism to manage authorization for a large set of users and data objects in a typical enterprise. New data objects get added or removed, users join, move, or leave organizations all the time. RBAC makes managing this a lot easier. Hence, as an extension of the discussed previously, if Carol joins the Finance Department, all you need to do is add her to the finance-department group in AD. This will give Carol access to data from the Sales and Customer tables.

Unified Authorization

Another important aspect of Sentry is the unified authorization. The access control rules once defined, work across multiple data access tools. For example, being granted the Analyst role in the previous example will allow Bob, Alice, and others in the finance-department group to access table data from SQL engines such as Hive and Impala, as well as via MapReduce, Pig applications or metadata access via HCatalog.

Sentry Integration with the Hadoop Ecosystem



As illustrated above, Apache Sentry works with multiple Hadoop components. At the heart you have the Sentry Server which stores authorization metadata and provides APIs for tools to retrieve and modify this metadata securely.

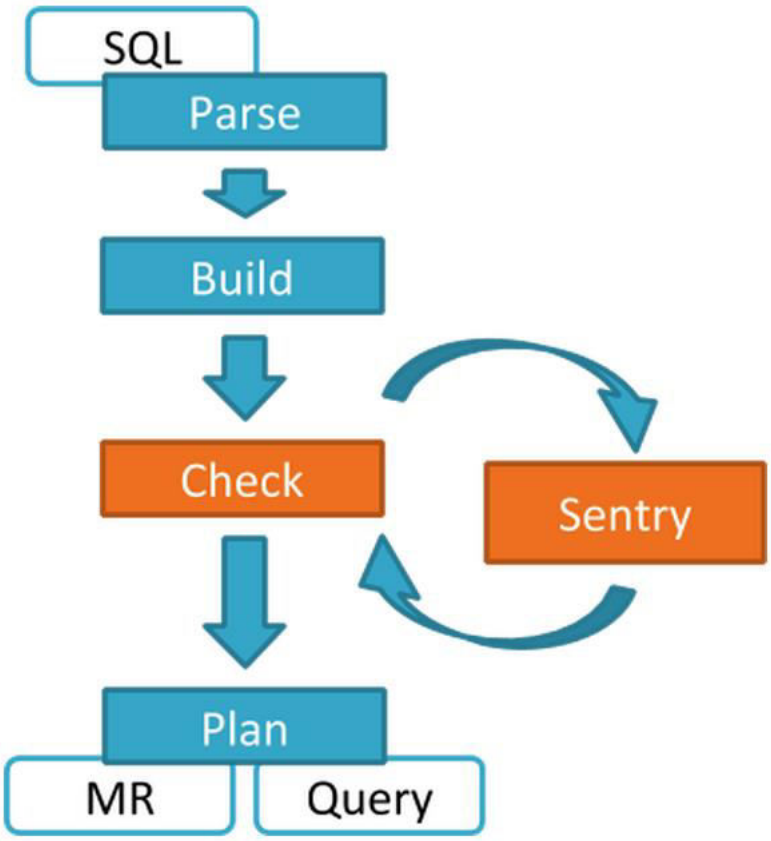
Note that the Sentry server only facilitates the metadata. The actual authorization decision is made by a policy engine which runs in data processing applications such as Hive or Impala. Each component loads the Sentry plugin which includes the service client for dealing with the Sentry service and the policy engine to validate the authorization request.

Hive and Sentry

Consider an example where Hive gets a request to access an object in a certain mode by a client. If Bob submits the following Hive query:

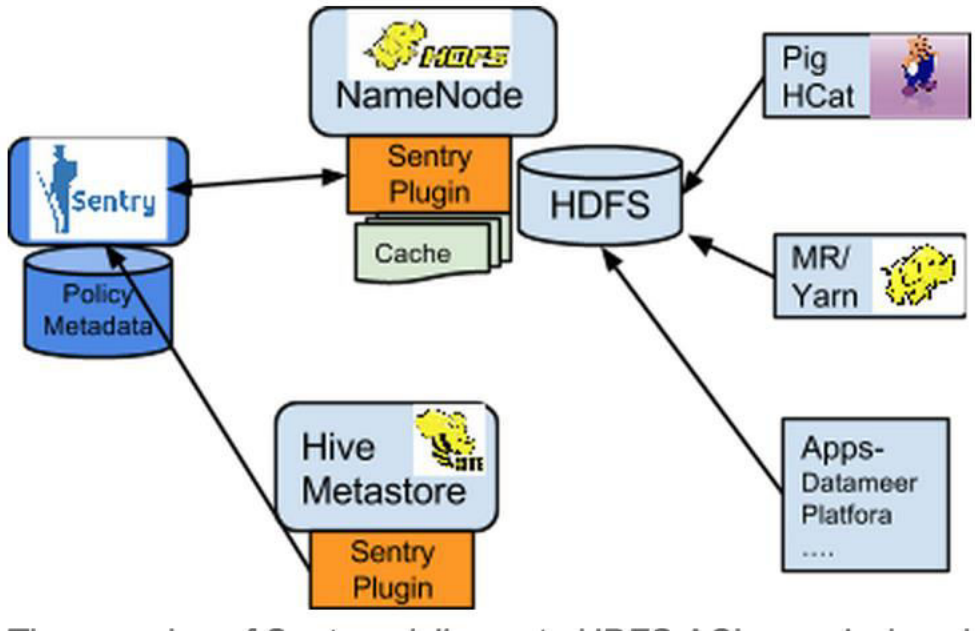
```
select * from production.sales
```

Hive will identify that user Bob is requesting SELECT access to the Sales table. At this point Hive will ask the Sentry plugin to validate Bob's access request. The plugin will retrieve Bob's privileges related to the Sales table and the policy engine will determine if the request is valid.



Sentry-HDFS Synchronization

Sentry-HDFS authorization is focused on Hive warehouse data - that is, any data that is part of a table in Hive or Impala. The real objective of this integration is to expand the same authorization checks to Hive warehouse data being accessed from any other components such as Pig, MapReduce or Spark. At this point, this feature does not replace HDFS ACLs. Tables that are not associated with Sentry will retain their old ACLs.



The mapping of Sentry privileges to HDFS ACL permissions is as follows:

SELECT privilege -> Read access on the file.

INSERT privilege -> Write access on the file.

ALL privilege -> Read and Write access on the file.

The NameNode loads a Sentry plugin that caches Sentry privileges as well Hive metadata. This helps HDFS to keep file permissions and Hive tables privileges in sync. The Sentry plugin periodically polls the Sentry and Metastore to keep the metadata changes in sync.

For example, if Bob runs a Pig job that is reading from the Sales table data files, Pig will try to get the file handle from HDFS. At that point the Sentry plugin on the NameNode will figure out that the file is part of Hive data and overlay Sentry privileges on top of the file ACLs. As a result, HDFS will enforce the same privileges for this Pig client that Hive would apply for a SQL query.

For HDFS-Sentry synchronization to work, you *must* use the Sentry service

ACL

Configuring ACLs on HDFS

Only one property needs to be specified in the hdfs-site.xml file in order to enable ACLs on HDFS:

- **dfs.namenode.acls.enabled**
  Set this property to "true" to enable support for ACLs. ACLs are disabled by default. When ACLs are disabled, the NameNode rejects all attempts to set an ACL.
  Example:

```
<property>
  <name>dfs.namenode.acls.enabled</name>
  <value>true</value>
</property>
```

Using CLI Commands to Create and List ACLs

Two new sub-commands are added to FsShell: `setfacl` and `getfacl`. These commands are modeled after the same Linux shell commands, but fewer flags are implemented. Support for additional flags may be added later if required.

- **setfacl**
  Sets ACLs for files and directories.
  Example:

```
-setfacl [-bkR] {-m|-x} <acl_spec> <path>
-setfacl --set <acl_spec> <path>
```

  Options:

Table 2.1. ACL Options

Option	Description
-b	Remove all entries, but retain the base ACL entries. The entries for User, Group, and Others are retained for compatbility with Permission Bits.
-k	Remove the default ACL.
-R	Apply operations to all files and directories recursively.
-m	Modify the ACL. New entires are added to the ACL, and existing entries are retained.
-x	Remove the specified ACL entires. All other ACL entries are retained.
--set	Fully replace the ACL and discard all existing entries. The acl_spec must include entries for User, Group, and Others for compatibility with Permission Bits.

Option	Description
<acl_spec>	A comma-separated list of ACL entries.
<path>	The path to the file or directory to modify.

Examples:

```
hdfs dfs -setfacl -m user:hadoop:rw- /file

hdfs dfs -setfacl -x user:hadoop /file

hdfs dfs -setfacl -b /file

hdfs dfs -setfacl -k /dir

hdfs dfs -setfacl --set user::rw-,user:hadoop:rw-,group::r--,other::r-- /file

hdfs dfs -setfacl -R -m user:hadoop:r-x /dir

hdfs dfs -setfacl -m default:user:hadoop:r-x /dir
```

Exit Code:
Returns 0 on success and non-zero on error.

- **getfacl**
  Displays the ACLs of files and directories. If a directory has a default ACL, getfacl also displays the default ACL.
  Usage:

```
 -getfacl [-R] <path>
```

  Options:

Table 2.2. getfacl Options

Option	Description
-R	List the ACLs of all files and directories recursively.
<path>	The path to the file or directory to list.

Examples:

```
hdfs dfs -getfacl /file

hdfs dfs -getfacl -R /dir
```

Exit Code:
Returns 0 on success and non-zero on error.

ACL Examples

Before the implementation of Access Control Lists (ACLs), the HDFS permission model was equivalent to traditional UNIX Permission Bits. In this model, permissions for each file or directory are managed by a set of three distinct user classes: Owner, Group, and Others. There are three permissions for each user class: Read, Write, and Execute. Thus, for any file system object, its permissions can be encoded in 3*3=9 bits. When a user attempts to access a file system object, HDFS enforces permissions according to the most specific user class applicable to that user. If the user is the owner, HDFS checks the Owner class permissions. If the user is not the owner, but is a member of the file system object's group, HDFS checks the Group class permissions. Otherwise, HDFS checks the Others class permissions.

This model can sufficiently address a large number of security requirements. For example, consider a sales department that would like a single user -- Bruce, the department manager -- to control all modifications to sales data. Other members of the sales department need to view the data, but must not be allowed to modify it. Everyone else in the company (outside of the sales department) must not be allowed to view the data. This requirement can be implemented by running chmod 640 on the file, with the following outcome:

```
-rw-r-----1 brucesales22K Nov 18 10:55 sales-data
```

Only Bruce can modify the file, only members of the sales group can read the file, and no one else can access the file in any way.

Suppose that new requirements arise. The sales department has grown, and it is no longer feasible for Bruce to control all modifications to the file. The new requirement is that Bruce, Diana, and Clark are allowed to make modifications. Unfortunately, there is no way for Permission Bits to address this requirement, because there can be only one owner and one group, and the group is already used to implement the read-only requirement for the sales team. A typical workaround is to set the file owner to a synthetic user account, such as "salesmgr," and allow Bruce, Diana, and Clark to use the "salesmgr" account via sudo or similar impersonation mechanisms. The drawback with this workaround is that it forces complexity onto end-users, requiring them to use different accounts for different actions.

Now suppose that in addition to the sales staff, all executives in the company need to be able to read the sales data. This is another requirement that cannot be expressed with Permission Bits, because there is only one group, and it is already used by sales. A typical workaround is to set the file's group to a new synthetic group, such as "salesandexecs," and add all users of "sales" and all users of "execs" to that group. The drawback with this workaround is that it requires administrators to create and manage additional users and groups.

Based on the preceding examples, you can see that it can be awkward to use Permission Bits to address permission requirements that differ from the natural organizational hierarchy of users and groups. The advantage of using ACLs is that it enables you to address these requirements more naturally, in that for any file system object, multiple users and multiple groups can have different sets of permissions.

Example 1: Granting Access to Another Named Group

To address one of the issues raised in the preceding section, we will set an ACL that grants Read access to sales data to members of the "execs" group.

- Set the ACL:
  ```
  > hdfs dfs -setfacl -m group:execs:r-- /sales-data
  ```
- Run **getfacl** to check the results:
- ```
  > hdfs dfs -getfacl /sales-data
  ```
- ```
  # file: /sales-data
  ```

```
# owner: bruce
# group: sales
user::rw-
group::r--
group:execs:r--
mask::r--
other::---
```

- If we run the "ls" command, we see that the listed permissions have been appended with a plus symbol (+) to indicate the presence of an ACL. The plus symbol is appended to the permissions of any file or directory that has an ACL.

```
> hdfs dfs -ls /sales-data
Found 1 items
-rw-r-----+ 3 bruce sales 0 2014-03-04 16:31 /sales-data
```

The new ACL entry is added to the existing permissions defined by the Permission Bits. As the file owner, Bruce has full control. Members of either the "sales" group or the "execs" group have Read access. All others do not have access.

Example 2: Using a Default ACL for Automatic Application to New Children

In addition to an ACL enforced during permission checks, there is also the separate concept of a default ACL. A default ACL can only be applied to a directory -- not to a file. Default ACLs have no direct effect on permission checks for existing child files and directories, but instead define the ACL that new child files and directories will receive when they are created.

Suppose we have a "monthly-sales-data" directory that is further subdivided into separate directories for each month. We will set a default ACL to guarantee that members of the "execs" group automatically get access to new subdirectories as they get created each month.

- Set a default ACL on the parent directory:

```
> hdfs dfs -setfacl -m default:group:execs:r-x /monthly-sales-data
```

- Make subdirectories:

```
> hdfs dfs -mkdir /monthly-sales-data/JAN
> hdfs dfs -mkdir /monthly-sales-data/FEB
```

- Verify that HDFS has automatically applied the default ACL to the subdirectories:

```
> hdfs dfs -getfacl -R /monthly-sales-data
# file: /monthly-sales-data
# owner: bruce
# group: sales
user::rwx
group::r-x
other::---
default:user::rwx
default:group::r-x
```

```
default:group:execs:r-x
default:mask::r-x
default:other::---

# file: /monthly-sales-data/FEB
# owner: bruce
# group: sales
user::rwx
group::r-x
group:execs:r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:execs:r-x
default:mask::r-x
default:other::---

# file: /monthly-sales-data/JAN
# owner: bruce
# group: sales
user::rwx
group::r-x
group:execs:r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:execs:r-x
default:mask::r-x
default:other::---
```

Example 3: Blocking Access to a Sub-Tree for a Specific User

Suppose there is a need to immediately block access to an entire sub-tree for a specific user. Applying a named user ACL entry to the root of that sub-tree is the fastest way to accomplish this without accidentally revoking permissions for other users.

- Add an ACL entry to block user Diana's access to "monthly-sales-data":

```
> hdfs dfs -setfacl -m user:diana:--- /monthly-sales-data
```

- Run **getfacl** to check the results:

```
> hdfs dfs -getfacl /monthly-sales-data
# file: /monthly-sales-data
# owner: bruce
# group: sales
user::rwx
user:diana:---
group::r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:execs:r-x
default:mask::r-x
default:other::---
```

It is important to keep in mind the order of evaluation for ACL entries when a user attempts to access a file system object:

- If the user is the file owner, the Owner Permission Bits are enforced.
- Else, if the user has a named user ACL entry, those permissions are enforced.
- Else, if the user is a member of the file's group or any named group in an ACL entry, then the union of permissions for all matching entries are enforced. (The user may be a member of multiple groups.)
- If none of the above are applicable, the Other Permission Bits are enforced.

In this example, the named user ACL entry accomplished our goal because the user is not the file owner and the named user entry takes precedence over all other entries.