Chapter 2
Hadoop, Hive, Spark with examples

Gaetan Robert Lescouflair
Sergio Simonian

2020-09-24

2.1 Introduction

As soon as we have more data than can be fit in one machine or we want to process more than a single machine can handle in a reasonable time, we tend to redesign our systems to work in a distributed manner. While a distributed system can scale horizontally (by adding more machines), it comes with new challenges to tackle. How to optimally distribute the workload across (many different) machines? How to ensure that the system does not interrupt or produce wrong results if a machine fails (fault tolerance) or becomes unavailable (partition tolerance)? Also, a distributed system has to serve multiple users and run several applications concurrently. In that case, how to manage file access rights and resource usage? Moreover, how to make the distributed system appear as a single coherent system to the end-users? This chapter will explore how Hadoop, Hive, and Spark handle these challenges and provide us with a simple way to set up a distributed system featuring distributed storage and parallel processing.

2.2 Hadoop

Apache Hadoop is an open-source, cross-platform framework written in Java for distributed data storage and parallel data processing. Doug Cutting and Mike Cafarella created Hadoop, inspired by two research papers from Google: "The Google File System" (2003) and "MapReduce: Simplified Data Processing on Large Clusters" (2004). It scales up from one machine to large clusters of thousands of machines with different hardware capacities (disk, CPU, RAM, and bandwidth). Hadoop plays an essential role in Big Data distributed computing and storage, ranging from structured to unstructured data. The machines in a Hadoop cluster work together to behave as if they were a single system. Leading providers of Big Data Database Management Systems have implemented the Hadoop platform in their enterprise solutions. For example, Oracle's "Big Data Appliance" [1] , Microsoft's "Polybase" [2] and IBM's "BigInsights" [3] .

[1] `https://docs.oracle.com/en/bigdata/big-data-appliance/5.1/bigug/concepts.html#GUID-8D18CCDF-D5EB-421B-9E5D-13027856EDA0`

[2] `https://docs.microsoft.com/en-us/sql/relational-databases/polybase/get-started-with-polybase`

[3] `https://www.ibm.com/support/knowledgecenter/en/SSPT3X_4.0.0/com.ibm.swg.im.infosphere.biginsights.product.doc/doc/bi_editions.html`

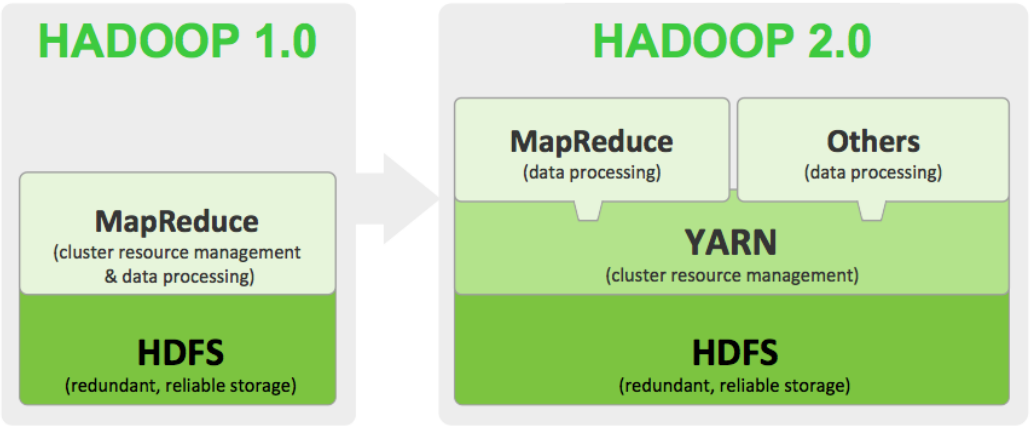

Figure 2.1: Differences between Hadoop 1.0 and Hadoop 2.0 [4]

At the time of this writing, the latest version of Hadoop is 3.3.1. In its first version (Figure 2.1), the essential components are the MapReduce model, which is responsible for the distributed processing and management of cluster resources, and HDFS (Hadoop Distributed File System) for the distributed storage. In the second version of Hadoop (Figure 2.1), the MapReduce model is used only for distributed processing, and YARN (Yet Another Resource Negotiator) has become the cluster resource manager. This change in architecture allows Hadoop to have a whole ecosystem around it, including other Frameworks capable of performing distributed data processing while adding new structures and new ways of making Hadoop function at the application and execution levels. Finally, in version 3, improvements are introduced to reduce storage costs while maintaining fault tolerance and optimizing resource management for even greater scalability.

In the following sections of this chapter, we will look in more detail at how MapReduce, HDFS, YARN, and the ecosystem around Hadoop works, how to install Hadoop, and how to run MapReduce jobs, and finally present Apache Hive and Apache Spark and how to use them with examples.

[4] https://infinitescript.com/wordpress/wp-content/uploads/2014/08/Differences-between-Hadoop-1-and-2.png

2.3 MapReduce

MapReduce is a paradigm designed to simplify parallel data processing on large clusters. Its principle is based on the "divide and conquer" technique - it divides the computation into sub-processes and runs them in parallel on the cluster.

A standard MapReduce program reads data from HDFS, splits it into parts, assigns for each part a key, groups these parts by their keys, and computes a summary for each group.

The four main steps of a MapReduce process:

A MapReduce process consists of several steps. Here are the four main steps in their corresponding order:

- **Split**: Split input data into multiple fragments to form subsets of data according to an index such as a space, comma, semicolon, new line, or any other logical rule.
- **Map**: Map each of the fragments into a new subset where the elements form key-value pairs.
- **Shuffle**: Group all the key-value pairs by their respective keys.
- **Reduce**: Perform a calculation on each group of values and output a possibly smaller set of values.

The general structure of a MapReduce process is in this form:

Map (key 1, value 1) -> list(key 2, value 2)
Reduce (key 2, list(value 2)) -> list(key 3, value 3)

Let us take a closer look at the MapReduce process steps with a word count example (Figure 2.2)

In the following table (Table 2.1), we see the shape of the input and output data for the different steps.

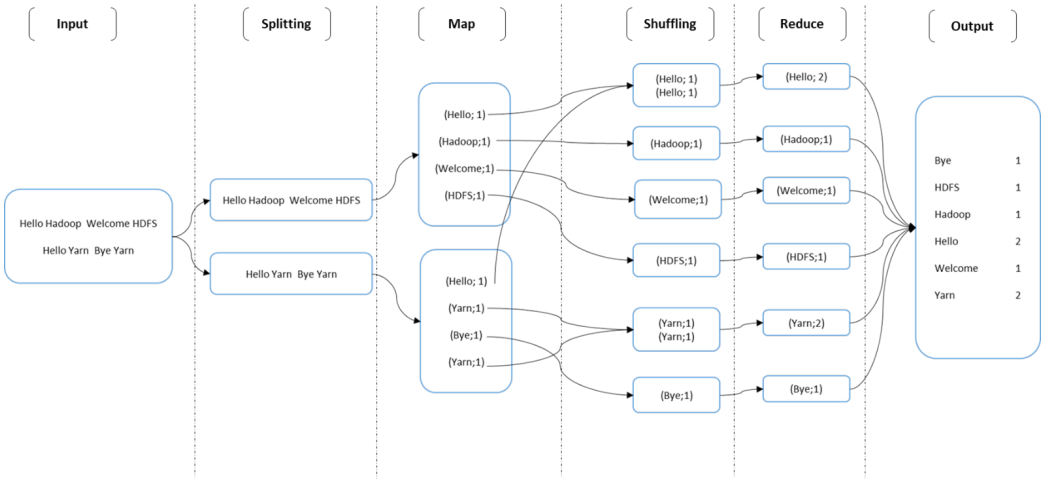

Figure 2.2: MapReduce process steps illustrated with the word counter example

Step	Input	Input type	Output	Output type
Split	Hello Hadoop Welcome HDFS Hello Yarn Bye Yarn	Text file	(1; "Hello Hadoop Welcome HDFS") (2; "Hello Yarn Bye Yarn")	Fragments of the input file in form of key-value pairs
Map	(1; "Hello Hadoop Welcome HDFS") (2; "Hello Yarn Bye Yarn")	Key-value pairs	(Hello;1), (Hadoop;1), (Welcome;1), (HDFS;1), (Hello; 1), (Yarn;1), (Bye;1), (Yarn;1)	Key-value pairs
Shuffle	(Hello;1), (Hadoop;1), (Welcome;1), (HDFS;1), (Hello; 1), (Yarn;1), (Bye;1), (Yarn;1)	Key-value pairs	[(Hello;1)(Hello;1)], [(Hadoop;1)], [(Welcome;1)], [(HDFS;1)], [(Yarn;1)(Yarn;1)], [(Bye;1)]	Groups of key-value pairs by key
Reduce	[(Hello;1)(Hello;1)], [(Hadoop;1)], [(Welcome;1)], [(HDFS;1)], [(Yarn;1)(Yarn;1)], [(Bye;1)]	Groups of key-value pairs by key	(Hello;2), (Hadoop;1), (Welcome;1), (HDFS;1), (Yarn;2), (Bye;1)	Subset of key-value pairs

Table 2.1: MapReduce input/output data in the word count example

Hadoop and MapReduce

In general, Hadoop executes each MapReduce process step in a distributed manner. To see how Hadoop distributes the split step, we will first look at how Hadoop stores its data in the following section.

2.4 HDFS

HDFS stands for Hadoop Distributed File System. As its name indicates, it is a distributed file system used by Hadoop. From a user perspective, it is similar to other filesystems such as Ext4, FAT32, NTFS, and HFS+. However, its internal functioning is very different. Due to its distributed nature, it can store large amounts of data. HDFS divides each file it stores into fixed-size blocks, distributes them over the entire cluster, and replicates each of them (by default three times) across the cluster to assure fault tolerance. In case a node in the cluster becomes unavailable, there will be, for each data block, two other nodes that have a replica of the lost block. All this is handled transparently for the end-user giving him the impression of a regular single-machine filesystem.

HDFS Architecture

HDFS is a master-worker architecture composed of two main daemons: NameNode and DataNode:

- The **NameNode** daemon is the master of the HDFS cluster. It stores metadata about all files and directories present on HDFS (their paths, data block IDs, access rights) and keeps track of all changes done to them. The NameNode persists this information on its local host OS file system in two types of files: *fsimage* and *edit-logs*. The *fsimage* contains the state of the file system at a given time, and the *edit-logs* record every change in the file system metadata since the creation of the last *fsimage*. The NameNode also keeps track of the locations of blocks and replicas on the cluster. All interactions like downloading/uploading/listing/creating/deleting/moving/copying files on HDFS first go through the NameNode. In order to serve clients as quickly as possible, the NameNode daemon keeps all metadata in memory (RAM)

and only persists metadata changes in the edit-logs. In case of a crash, when the NameNode restarts, it loads the last *fsimage* in memory and applies the changes from the *edit-logs* to restore its previous state. However, to keep the file system consistent for all clients, there can only be one active NameNode on the cluster. Unfortunately, this constraint creates a single point of failure. If the NameNode becomes unavailable, the whole HDFS cluster cannot be used by the clients anymore. Fortunately, Hadoop provides a way to assure the NameNode High Availability by running one or multiple Standby NameNodes that keep their state in sync with the active one and can take over its role in case of a failure. Moreover, failover from the Active NameNode to a Standby NameNode can be relatively quick and transparent to the clients.

- The **DataNode** daemon is a worker of the HDFS cluster. It runs on every cluster node, except usually the Namenode, and is responsible for storing and managing data blocks. Each DataNode performs block creation, deletion, and replication upon instruction from the NameNode and serves read and write requests from the file system's clients. It also periodically sends Heartbeats and block reports to the NameNode to confirm that it is alive and healthy. The DataNodes communicate as well with each other to perform data replication.

Reading and Writing files on HDFS

When a filesystem client wants to read a file in HDFS, it has to ask the NameNode where to retrieve the data blocks for the file. The NameNode will respond with a list of DataNodes for each block. Then the client has to contact the DataNodes directly to retrieve the corresponding data blocks. Finally, the filesystem client reconstructs the original file by merging the retrieved data blocks.

Writing files to HDFS is quite similar. The client splits the original file into blocks and asks the NameNode where to store them. The NameNode will respond with a list of DataNodes for each block. Then the client has to contact the DataNodes directly to send the corresponding file data blocks. Next, the DataNodes will replicate the received blocks and send an acknowl-

[5] `https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`

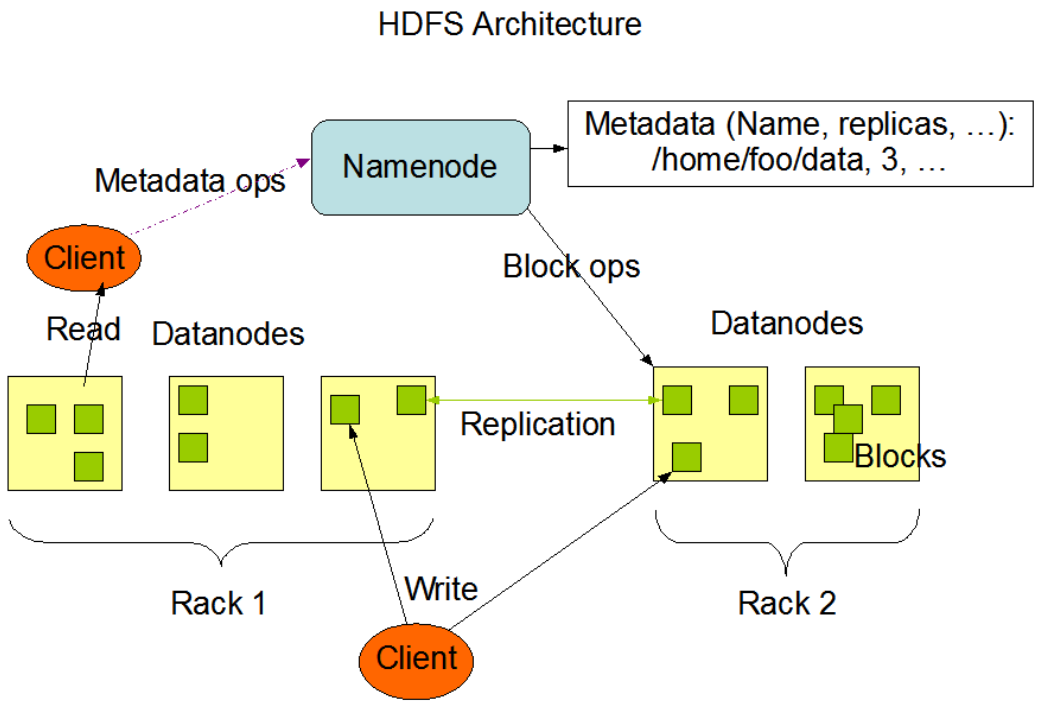

Figure 2.3: HDFS Architecture [5]

edgment to the client. Finally, the client will notify the NameNode about the completion of the write operation.

Fortunately, for the end-user, reading and writing files on HDFS is abstracted by a command-line interface (**hadoop fs** [6]) and a java library (**org.apache.hadoop.fs**[7]).

Using HDFS comand-line interface

Here we will demonstrate the usage of the Hadoop command-line interface with some examples.

To create some nesting directories, we can use the **-mkdir** argument followed by **-p** and the path of the directories.

[6] https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html

[7] https://hadoop.apache.org/docs/r3.3.1/api/org/apache/hadoop/fs/package-summary.html

```
hadoop fs -mkdir -p /user/hdoop/example
```

Next, we will create two files in our local filesystem and upload them to HDFS in our example directory using the **-put** argument.

```
echo "Hello Hadoop Welcome HDFS" > test_file_1.txt
echo "Hello Yarn  Bye Yarn" > test_file_2.txt
hadoop fs -put test_file_1.txt test_file_2.txt /user/hdoop/example
```

To list the content of our directory, we can use the **-ls** argument.

```
hadoop fs -ls /user/hdoop/example
```

To print the content of a file on the standard output, we can use the **-cat** argument.

```
hadoop fs -cat /user/hdoop/example/test_file_1.txt
```

Finally, to download a file back to our local filesystem, we can use the **-get** option.

```
hadoop fs -get /user/hdoop/example/test_file_2.txt ./downloaded_file.txt
```

Note that the **hadoop fs** command has many more arguments providing additional functionality. We have merely shown some of the most common and simple ones. For an exhaustive list of all available arguments, refer to the official Hadoop documentation. [8])

MapReduce and HDFS

In a nutshell, files stored on HDFS are split into fixed-size blocks, replicated, and spread over the Hadoop cluster. This prior setup allows Hadoop to distribute the processing of MapReduce programs on cluster nodes that already possess the required data (data-locality). Let us now revisit the word count MapReduce example. In the first MapReduce step, we want to split our input file by line, and we have mentioned that Hadoop will perform this task in a distributed manner. Hadoop will start by computing the number of split operation tasks to spawn and selects several cluster nodes to perform these tasks based on their current resource availability, configured policies, and other factors with a preference for nodes that possess the required data.

[8] `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html`

By default, the number of split operation tasks is equal to the number of data blocks of the input file. However, to accomplish this, Hadoop needs a way to coordinate its tasks which is the subject of the next section.

2.5 YARN

YARN is Hadoop's resource manager that distributes tasks to all the machines in the Hadoop cluster and tracks the status of the running tasks.

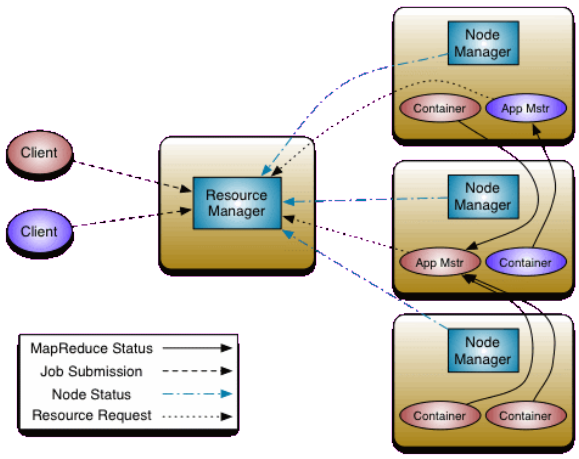

Figure 2.4: YARN architecture [9]

YARN is also a master-worker architecture composed of two main daemons: ResourceManager (master) and NodeManager (worker). (See Figure 2.4):

- The **ResourceManager** keeps track and manages available resources in the cluster. It also receives application submissions from clients.
- The **NodeManager** runs on each node of the cluster, except usually the ResourceManager node, and is responsible for providing execution containers. The containers consist of execution environments with limited resources (like RAM or CPU) and run tasks (for example, MAP, REDUCE, ApplicationMaster, ...).

[9] Source : `https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html`

****
When a YARN receives an application submission from a client, it starts its execution on a NodeManager cluster node. **********

2.6 Hadoop Ecosystem

Hadoop is a framework that includes a range of tools and technologies around it which form the Hadoop ecosystem (see Figure 2.5). Before starting to work with Hadoop, it is vital to understand its environment. Each tool can play a substantial role in different parts of a Big Data Project. HDFS, YARN, and MapReduce are the foundation of the Hadoop ecosystem. Most tools of the Hadoop ecosystem are open-source projects from the Apache Software Foundation. However, there are proprietary solutions too. All the tools in the ecosystem have the ingestion, storage, analysis of data, and maintenance of the system as their primary purpose.

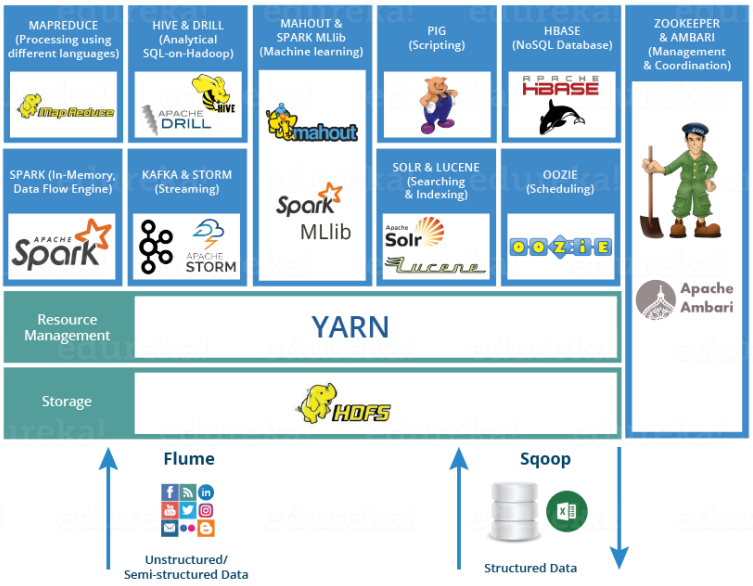

Figure 2.5: Hadoop ecosystem [10]

[10]Source : https://cdn.edureka.co/blog/wp-content/uploads/2016/10/HADOOP-ECOSYSTEM-Edureka.png

The number of tools around Hadoop is constantly increasing. In this section, we will take a look at the most commonly used tools currently on the market in the field of Big Data.

2.6.1 Hive

Apache Hive [11] is a Data Warehousing Framework that allows you to read, write and manage large volumes of data in a distributed environment from an SQL-like interface.

2.6.2 Spark

Apache Spark [12] is a framework for performing data analysis using in-memory data processing in a distributed environment.

2.6.3 Sqoop

Apache Sqoop [13] is an ETL (Extract, Transform, Load) tool designed to perform data transfers between Hadoop and structured data (relational database, CSV file, ...) on large volumes efficiently.

2.6.4 Hbase

Apache HBase [14] is a Hadoop database with the ability to manage read and write access to large volumes of data in a random and real-time manner. HBase is a database capable of maintaining large tables that can contain millions of columns.

2.6.5 Pig

Apache Pig [15] is a platform for performing data analysis on large volumes of data. It offers a high-level language, named Pig Latin, with command

[11] See https://hive.apache.org/
[12] See https://spark.apache.org/
[13] Voir http://sqoop.apache.org/
[14] Voir https://hbase.apache.org/
[15] Voir https://pig.apache.org/

structures similar to SQL. When compiled, it produces MAP and REDUCE job sequences already capable of being parallelized on Hadoop.

2.6.6 Zookeeper

Apache Zookeeper [16] is a centralized service manager in a distributed environment. It allows to maintain configuration information and provides distributed information synchronization and enumeration and grouping services.

2.6.7 Ambari

Apache Ambri [17] is a management tool that provides services to simplify new service provisioning and configuration, management, and monitoring in Hadoop clusters.

2.6.8 Oozie

Apache Oozie [18] is an event scheduling and triggering system in Hadoop. It can be considered as a clock or alarm service internal to Hadoop. It can execute a set of events one after the other or trigger events based on the availability of information. The events launched can be map-reduce, Pig, Hive, Sqoop, Java program tasks, and many others.

2.6.9 Apache Solr and Lucene

Apache Solr and Apache Lucene [19] are two services that are used for search and indexing in the Hadoop environment. They are suitable for implementing information systems that require full-text search. Lucene is a core component, and Solr is built around it, adding even more functionality.

---

[16] See https://zookeeper.apache.org/
[17] See https://ambari.apache.org/
[18] See http://oozie.apache.org/
[19] See https://solr.apache.org/

2.6.10 Kafka

Apache Kalka [20] is a distributed messaging system for publishing, subscribing, and recording data stream exchanges. It allows the creation of a data distribution pipeline between systems or applications.

2.6.11 Storm

Apache Storm [21] is a data stream processing system for real-time analytics use cases, machine learning, continuous operations monitoring.

2.6.12 Flume

Apache Flume [22] is a distributed service for collecting, aggregating, and transferring large volumes of semi-structured or unstructured data from online streams in HDFS.

2.6.13 Drill

Apache Drill [23] is a schema-free SQL query engine for Hadoop, NoSQL, and Cloud Storage. It supports a variety of NoSQL databases and is capable of performing join queries between multiple data sources.

2.6.14 Mahout

Apache Mahout [24] provides an environment for the development of Machine Learning applications at scale.

2.6.15 Impala

Apache Impala [25] and Presto [26] are SQL query engines designed for Big Data. They are capable of processing Petabytes of data very quickly. For

[20] See https://kafka.apache.org/
[21] See https://storm.apache.org/
[22] See https://flume.apache.org/
[23] See https://drill.apache.org/
[24] See https://mahout.apache.org/
[25] See https://impala.apache.org/
[26] Voir https://prestodb.io/

more information on Impala, see « Impala : A Modern, Open-Source SQL Engine for Hadoop » [27] For Presto, see « Presto: Interacting with petabytes of data at Facebook » [28]

2.7 Hadoop 3.3.1 cluster installation on Linux Ubuntu 20.04.1 LTS

This section shows the installation and configuration of an Apache Hadoop version 3.3.1 cluster with YARN. As shown in the following diagram 2.6, the installation will be on three (3) machines with one Master node (hdmaster) and two Worker nodes (hdworker1 and hdworker2). However, because this example setup is a pretty small cluster and the Hadoop master services will not consume much processing power on the Master node, we will also use the Master node as a Worker node.

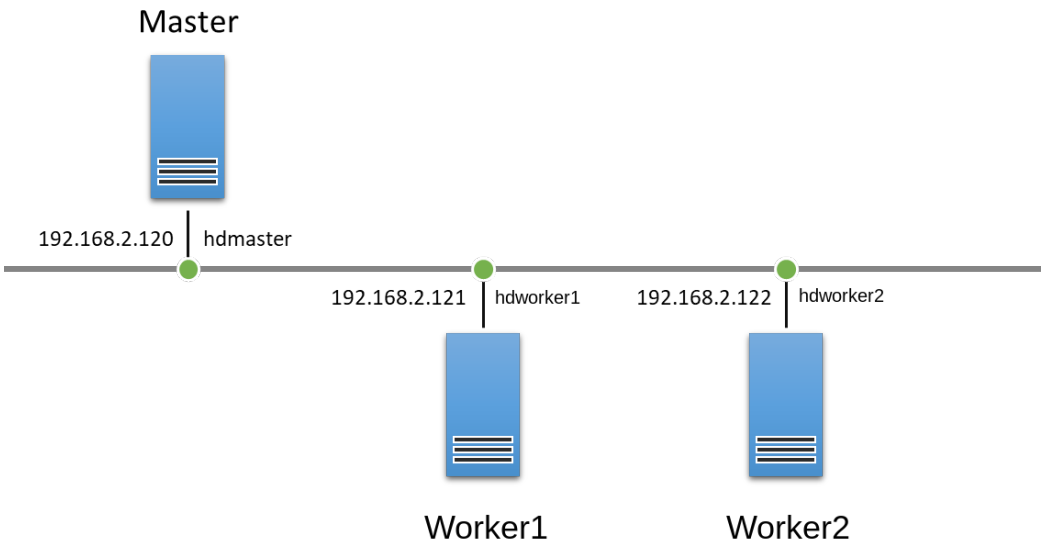

Figure 2.6: Example Cluster Schema

[27] M. Kornacker et al., « Impala: A Modern, Open-Source SQL Engine for Hadoop. », in CIDR, 2015, vol. 1, p. 9.

[28] "Presto: Interacting with petabytes of data at Facebook." [Online]. https://www.facebook.com/notes/facebookengineering/presto-interacting-with-petabytes-of-data-atfacebook/10151786197628920.

The machines used for this example installation are interconnected through a network switch, run Linux Ubuntu 20.04.1 LTS operating system, have about 4G of RAM, 200G free space on their hard drives, and Intel i3 processors.

2.7.1 Prerequisites

In this section, we will see the initial setup for the operation of Apache Hadoop and some best practices. Since Hadoop is a Java-based platform, in order for it to work, it needs the Java Virtual Machine (JVM) to run. Hadoop 3.3.1 can run on Java 8 or 11. We will install Java 8 because many Hadoop ecosystem components only support Java versions up to 8. Another important aspect is that Hadoop uses SSH (Secure Shell) to connect to the cluster nodes. Moreover, to provide better isolation and security between Hadoop services, it is recommended to create dedicated users.

2.7.1.1 Install Java version 8

Firstly, we will install Java 8 on all cluster nodes.

In the terminal:

Update and upgrade packages:

```
sudo apt update
sudo apt upgrade
```

Install Java 8 OpenJDK Development Kit:

```
sudo apt install openjdk-8-jdk
```

Check the Java version:

```
java -version
```

The output should look similar to this:

```
openjdk version "1.8.0_275"
OpenJDK Runtime Environment (build 1.8.0_275-8u275-b01-0ubuntu1~20.04-b01)
OpenJDK 64-Bit Server VM (build 25.275-b01, mixed mode)
```

2.7.1.2 Hostname configuration

It is important to determine the hostnames and IP addresses associated with each machine from the start. Our master node is named "hdmaster", and

our worker nodes are "hdworker1" and "hdworker2". Make sure that each node has a static IP address so that it does not change over time. This can be done from the network configuration file, which can be found at /etc/network/interfaces. Add in the **/etc/hosts** file the **IP addresses and hostnames** corresponding to each node. We are using the **vim** text editor for this task. However, any other text editor could do it as well.

```
sudo vim /etc/hosts
```

```
# IPs and Hostnames for Hadoop configuration
192.168.2.120 hdmaster
192.168.2.121 hdworker1
192.168.2.122 hdworker2
```

2.7.1.3 Create a Hadoop user for HDFS and MapReduce access

We will create a non-root Hadoop user and a group named **hdoop** on each cluster node. However, using separate users for each Hadoop service is preferable because it provides better isolation and security.

```
sudo adduser hdoop
```

2.7.1.4 SSH installation

In the terminal:

```
sudo apt install ssh
```

Next, we will set up Passwordless SSH access for the Hadoop user. Generate the SSH key pair with passphrase in the master node:

```
su - hdoop
ssh-keygen -t rsa -b 4096 -m pem
```

Then copy the SSH key from Master to the Workers and localhost to initiate SSH access without a password.

```
ssh-copy-id -i $HOME/.ssh/id_rsa.pub hdoop@hdworker1
ssh-copy-id -i $HOME/.ssh/id_rsa.pub hdoop@hdworker2
ssh-copy-id -i $HOME/.ssh/id_rsa.pub hdoop@localhost
```

Finally load the password for they SSH key in memory with ssh-agent

```
ssh-agent $SHELL
ssh-add
```

To check that the Hadoop User has gained passwordless access for the localhost and the worker nodes, we will attempt to connect to each node.

```
ssh hdoop@localhost
exit
ssh hdoop@hdworker1
exit
ssh hdoop@hdworker2
exit
```

2.7.2 Hadoop installation

The binary version of Apache Hadoop can be downloaded from the official website (https://hadoop.apache.org/). We choose the "/usr/local/" directory for the installation.

```
cd /usr/local/
```

Download the Hadoop archive file:

```
sudo wget https://miroir.univ-lorraine.fr/apache/hadoop/common/hadoop-3.3.1/hadoop-3.3.
```

Unarchive the newly downloaded file (here hadoop-3.3.1.tar.gz)

```
sudo tar xzf hadoop-3.3.1.tar.gz
```

Give the hdoop user the ownership of the directory:

```
sudo chown hdoop:hdoop -R /usr/local/hadoop-3.3.1
```

We will also create an alias for our installation directory, which could be useful when upgrading the Hadoop version in the future:

```
sudo ln -s hadoop-3.3.1 hadoop
```

Setting up Hadoop environment variables

There are various environment variables to configure the Hadoop installation. Some notable environment variables are:

- The **JAVA_HOME** variable informs Hadoop where to find the Java installation.
- The **HADOOP_HOME** variable holds the absolute path to the Hadoop installation and the **HADOOP_CONF_DIR** variable points to the directory containing the Hadoop configuration files. Hadoop ecosystem tools often require these variables to be set in order to find Hadoop libraries and configurations.
- The **HADOOP_OPTS** variable specifies JVM (Java Virtual Machine) options to use when starting Hadoop services.

We will define these environment variables for all cluster nodes in the **.profile** file at the home directory of the *hdoop* user. In this way, the shell will define our environment variables on each subsequential login. First, switch to the hdoop user:

```
su - hdoop
```

Add Hadoop environment variables to the end of the hdoop user profile file.

```
vim /home/hdoop/.profile
```

```
## BEGIN -- HADOOP ENVIRONMENT VARIABLES
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib/native"
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
export PATH=$PATH:$JAVA_HOME/bin
## END -- HADOOP ENVIRONMENT VARIABLES
```

Make the change of the profile file active immediately:

```
source /home/hdoop/.profile
```

Update the Hadoop environment configuration file

```
vim /usr/local/hadoop-3.3.1/etc/hadoop/hadoop-env.sh
```

Find the line defining containing "export JAVA_HOME=" and update it to:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Check if the Hadoop command is now defined:

```
hadoop version
```

Hadoop is in its default configuration at this stage, which means it is in "Stand Alone" mode.

2.7.3 Configuring Hadoop in fully-distributed mode

To change Hadoop's default configuration, we can use site-specific configuration files located by default at **$HADOOP_HOME/etc/hadoop**. Adding parameters to for example **mapred-site.xml**, **yarn-site.xml**, **capacity-scheduler.xml**, and other files in this directory means that Hadoop must consider these new properties listed instead of the default values.

2.7.3.1 Edit the "workers" file

The Hadoop master daemons need to know the hostnames or IP addresses of the worker nodes to communicate with and manage them. To provide this information to the Hadoop master daemons, we will edit the "workers" file. In the master node:

```
vim /usr/local/hadoop/etc/hadoop/workers
```

Insert all workers hostnames or IP addresses (one per line)

```
hdworker1
hdworker2
localhost
```

Note that by adding **localhost** to this file, we also use our Master node as a Worker node.

2.7.3.2 Edit the "core-site.xml" file

The **core-site.xml** file enables us to overwrite Hadoop's default configuration properties from **core-default.xml**. Hadoop's official website provides more details about configurable values in the core-site.xml and the set of

default values. [29] We will set the default file system property to HDFS and point it to our master node for our installation.

In the master node and the worker nodes :

```
vim /usr/local/hadoop/etc/hadoop/core-site.xml
```

Insert this property between the opening (<configuration>) and closing (</configuration>) tags.

```
<property>
        <name>fs.default.name</name>
        <value>hdfs://hdmaster:9000</value>
</property>
```

2.7.3.3 Creation of Hadoop's data directories

Optionally, we can define the directories where HDFS DataNodes will store their local data blocks and where the NameNode will store its *edit-logs* and *fsimage*. In our example we will use the "/usr/local/tmp_hadoop/namenode" and "/usr/local/tmp_hadoop/datanode" directories. First, we will create the directories, and then in the following sections, we will present the corresponding Hadoop configuration. In the master node (NameNode):

```
sudo mkdir -p /usr/local/tmp_hadoop/hdfs/namenode
sudo mkdir -p /usr/local/tmp_hadoop/hdfs/datanode
sudo chown -R hdoop:hdoop /usr/local/tmp_hadoop/
```

In the DataNodes nodes :

```
sudo mkdir -p /usr/local/tmp_hadoop/hdfs/datanode
sudo chown -R hdoop:hdoop /usr/local/tmp_hadoop/
```

2.7.3.4 Edit the "hdfs-site.xml" file

The **hdfs-site.xml** file enables us to overwrite the default configuration for the HDFS client from **hdfs-default.xml**. In our example, we will configure the block replication factor to 3 and indicate that the NameNode should store its local files (fsimage/edit-logs) in the data directory we created in the previous section ("Creation of Hadoop's data directories"). For more

[29] See https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-common/core-default.xml

details about what is configurable in the **hdfs-site.xml** see Hadoop's official website. [30]

In the master node and the worker nodes :

```
vim /usr/local/hadoop/etc/hadoop/hdfs-site.xml
```

Insert these properties between the opening (<configuration>) and closing (</configuration>) tags.

```
<property>
        <name>dfs.replication</name>
        <value>3</value>
 </property>
<property>
        <name>dfs.namenode.name.dir</name>
        <value>/usr/local/tmp_hadoop/hdfs/namenode</value>
</property>
<property>
        <name>dfs.datanode.data.dir</name>
        <value>/usr/local/tmp_hadoop/hdfs/datanode</value>
</property>
```

2.7.3.5 Edit the "yarn-site.xml" file

The **yarn-site.xml** file enables us to overwrite the default configuration for YARN from **yarn-default.xml**. In our example, we will configure the hostnames and ports used by the Resource Manager and Node Managers. We will also configure the auxiliary shuffle service and reduce the minimum container memory allocation value. Furthermore, we will enable log aggregation to store container logs on HDFS. For more details about what is configurable in the **yarn-site.xml** see Hadoop's official website. [31]

In the master node and the slave nodes :

```
vim /usr/local/hadoop/etc/hadoop/yarn-site.xml
```

Insert these properties between the opening (<configuration>) and closing (</configuration>) tags.

[30] See https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml

[31] https://hadoop.apache.org/docs/r3.3.1/hadoop-yarn/hadoop-yarn-common/yarn-default.xml

```
<property>
        <name>yarn.resourcemanager.hostname</name>
        <value>hdmaster</value>
</property>
<property>
        <name>yarn.resourcemanager.address</name>
        <value>hdmaster:8032</value>
</property>
<property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value>hdmaster:8030</value>
</property>
<property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>hdmaster:8031</value>
</property>
<property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
</property>
<property>
          <name>yarn.scheduler.minimum-allocation-mb</nam>
          <value>256</value>
</property>
<property>
          <name>yarn.log-aggregation-enable</name>
          <value>true</value>
</property>
```

2.7.3.6 Edit the "mapred-site.xml" file

The **mapred-site.xml** file enables us to overwrite Hadoop's default configuration properties from **mapred-default.xml**. Hadoop's official website provides more details about what is configurable in the **mapred-site.xml** along with the set default values. [32] We will indicate that we want to use YARN as the runtime framework for executing our MapReduce jobs for our installation. We will also indicate where to search for related jar files and packages for our MapReduce applications. In the master node and the slave nodes :

[32] http://hadoop.apache.org/docs/r3.3.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml

```
vim /usr/local/hadoop/etc/hadoop/mapred-site.xml
```

Insert these properties between the opening (<configuration>) and closing (</configuration>) tags.

```
<property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>
</property>
<property>
        <name>yarn.app.mapreduce.am.env</name
        <value>HADOOP_MAPRED_HOME=$HADOOP_MAPRED_HOME</value>
</property>
<property>
        <name>mapreduce.map.env</name>
        <value>HADOOP_MAPRED_HOME=$HADOOP_MAPRED_HOME</value>
</property>
<property>
        <name>mapreduce.reduce.env</name>
        <value>HADOOP_MAPRED_HOME=$HADOOP_MAPRED_HOME</value>
</property>
```

2.7.3.7 Format the NameNode

In the master node, to start HDFS for the first time, it is required to format the NameNode.

```
hdfs namenode -format
```

2.7.4 Starting the Hadoop daemons on the cluster

2.7.4.1 Starting the HDFS daemons

To start HDFS, Hadoop provides a shell script named **start-dfs.sh**. In the master node:

```
start-dfs.sh
```

Run this command to check that the HDFS daemons started.

```
jps
```

You should get something similar to this

```
40161 NameNode
40708 Jps
40549 SecondaryNameNode
```

In the slave nodes :

When the NameNode daemon launches, it connects to the worker nodes through SSH.

To check that the worker nodes are properly started, we can connect to them with SSH and run this command:

```
jps
```

```
8561 Jps
7753 DataNode
```

2.7.4.2 Starting the YARN daemon

Similar to HDFS, Hadoop provides a script to start YARN named **start-yarn.sh** In the master node: Run this command to start YARN

```
start-yarn.sh
```

Check that the YARN ResourceManager has started

```
jps
```

The output should be similar to this

```
8128 ResourceManager
8561 Jps
7604 NameNode
7964 SecondaryNameNode
```

In the worker nodes:

Here too, the YARN ResourceManager will connect to the worker nodes and start the NodeManager daemon. No further actions are required.

To check the NodeManager has started, connect to the slave nodes and run this command:

```
jps
```

The output should be similar to this

```
8561 Jps
8249 NodeManager
7753 DataNode
```

To stop the Hadoop daemons, run the **stop-yarn.sh** and **stop-dfs.sh** scripts as the hadoop user (hdoop in our example).

Configuring YARN and MapReduce for optimal resource management mbox

It is crucial to know where to find the ideal balance for the system to manage and optimize shared resources and memory usage. One configuration may work well for one application and not for another. In many cases, processes running multiple applications fail because the memory size of ApplicationMaster and other containers exceeds the available capacity. Requiring resource-heavy containers may involve the application being accepted for execution and the process being left in a queue or abruptly stopped during execution.

The following tables describe the default and current values of some relevant configuration properties in their respective files.

mapred-site.xml

Property name	Default value	Current value	description
mapreduce.map.memory.mb	1204	256	
mapreduce.reduce.memory.mb	3072	256	
mapreduce.map.java.opts	-Xm900m	-Xmx205m	
mapreduce.reduce.java.opts	-Xm2560m	-Xmx205m	
yarn.app.mapreduce.am.resource.mb	1536	768	
yarn.app.mapreduce.am.command-opts	-Xm1024m	-Xmx615m	

Table 2.2:

yarn-site.xml

Property name	Default value	Current value	description
yarn.nodemanager.resource.memory-mb		2048	
yarn.scheduler.minimum-allocation-mb	1024	256	
yarn.scheduler.maximum-allocation-mb	8192	1408	
yarn.scheduler.minimum-allocation-vcores	1	1	
yarn.scheduler.maximum-allocation-vcores	32	4	
yarn.scheduler.increment-allocation-mb		128	
yarn.nodemanager.vmem-check-enabled	true	false	
yarn.nodemanager.pmem-check-enabled	true	true	

Table 2.3:

Hadoop MapReduce example program

This section will showcase how to compile and run a MapReduce program for Hadoop. We will use a typical introductory WordCount example. In the example, we will use the **test_file_1.txt** and **test_file_2.txt** files stored in the **/user/hdoop/example** HDFS directory, which we created earlier.

First, we will create a directory in our local filesystem to put our .java program.

```
mkdir    ~/word_count
cd   ~/word_count/
```

Next we will save the following Java code in a file named **WordCount.java** in our **word_count** directory.

WordCount.java [33]

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
```

[33] Source: https://hadoop.apache.org/docs/r3.3.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

```
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString())
          ↪ ;
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException
                          ↪  {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
```

```
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

To compile this program, Java needs to know where to find the present Hadoop libraries. One way to specify the locations of our libraries for the Java compiler is to add their paths to the **CLASSPATH** environment variable.

```
export CLASSPATH="/usr/local/hadoop/share/hadoop/common/hadoop-
    ↪ common-3.3.1.jar:/usr/local/hadoop/share/hadoop/mapreduce/
    ↪ hadoop-mapreduce-client-common-3.3.1.jar:/usr/local/hadoop
    ↪ /share/hadoop/common/lib/commons-cli-1.2.jar:/usr/local/
    ↪ hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-core
    ↪ -3.3.1.jar"
```

Them, we can compile our WordCount.java file with the following command:

```
javac WordCount.java
```

Next, package our newly compiled program in a JAR archive named **wc.jar**

```
jar -cf wc.jar WordCount*.class
```

Finally, we will launch our MapReduce program by indicating as input the HDFS directory (/user/hdoop/example) where the **test_file_1.txt** and **test_file_2.txt** files are located and, as output, the directory (/user/hdoop/output_example) where the results will be saved.

```
hadoop jar wc.jar WordCount /user/hdoop/example /user/hdoop/output_example
```

To verify the results of our MapReduce program, we will list the contents of the **output_example** directory.

```
hadoop fs -ls  /user/hdoop/output_example
```

```
Found 2 items
-rw-r--r--   3 hdoop supergroup          0 2021-09-29 07:59 /user/hdoop/output_e
-rw-r--r--   3 hdoop supergroup         47 2021-09-29 07:59 /user/hdoop/output_e
```

The generated "_SUCCESS" file indicates that the program execution was successful. The results are in the file **part-r-00000**. The number of result files names **part-r-XXXXX** depends on the number of reducer tasks involved in the MapReduce process, where **XXXXX** is a counter starting at **00000**.

To print the result file content to standart output we can use the following command:

```
hadoop fs -cat /user/hdoop/output_example/part-r-*
```

```
Bye             1
HDFS            1
Hadoop          1
Hello           2
Welcome 1
Yarn            2
```

Hadoop's default way to write final key-value pairs is to have one key-value per line where a tab character separates each key from the corresponding value.

2.8 Hive

Apache Hive is a data warehousing framework that provides an SQL-like interface to interact with large amounts of structured and semi-structured data stored in a distributed environment. Originally it was developed by Facebook to make processing data stored on HDFS more accessible for analysts familiar with SQL. The SQL language of Hive is called Hive Query Language (HiveQL or HQL). HiveQL is pretty similar to regular SQL. Though, it does not fully support all SQL-95 features [34] and has some differences in syntax. While Hive might seem to be a database, it is not a database per se. Instead, it provides a perception of a database with tables and rows based on data stored on a filesystem. One of Hive's typical use cases is to map data stored on HDFS into databases and tables and then process it with HiveQL. However, Hive can also interact with many other storage systems and databases like MySQL, MongoDB, Oracle, and HBase. When Hive receives a query, it transforms it into a sequence of MapReduce tasks and submits them to

[34] https://dwgeek.com/what-are-sql-features-missing-in-hive.html/

YARN. This approach enables Hive to execute queries at the scale of the underlining Hadoop cluster.

Hive architecture

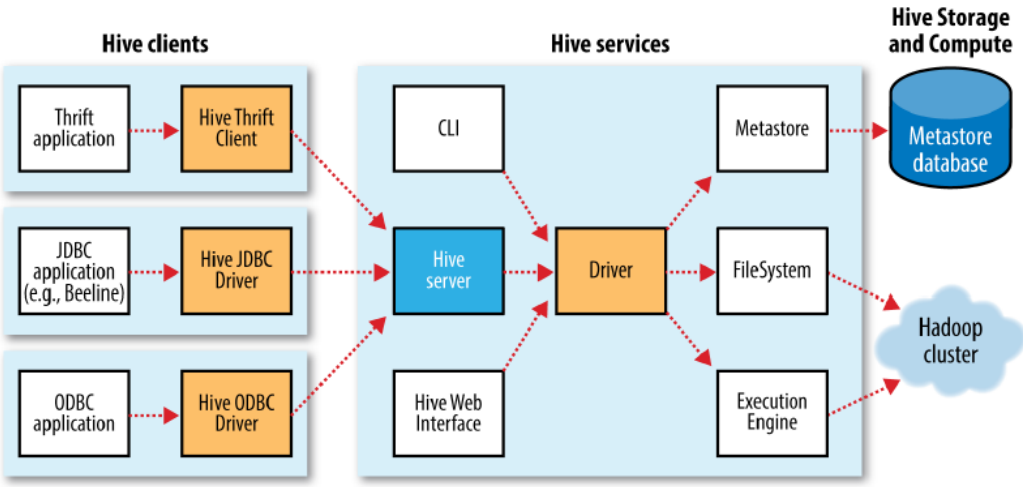

Figure 2.7: Hive architecture [35]

At a very high level, the architecture of Hive has three principal components: **Metastore**, **HiveServer2**, and **Beeline**.

- The **Metastore** is a service that stores metadata about Hives databases and tables in a relational database (RDBMS). By default, the Metastore uses an embedded Apache Derby database which is limited to only one client connection. Therefore it is recommended to configure Hive to use an external database. The user installing Hive on their system can choose any DataNucleus compliant relational database for the Hive Metastore (like MySQL, Oracle, Postgres). There are two ways to run the Metastore service: either in embedded mode or in server mode. In the embedded mode, the Metastore service works as a library inside the HiveServer2 process. In server mode, the Metastore runs as a separate process which might or might not be on the same machine where the HiveServer2 service runs.

[35]Source: T. White, Hadoop: the definitive guide; [storage and analysis at Internet scale], 4. ed., Updated Beijing: O'Reilly, 2015. p.480

- The **HiveServer2** is a server that listens for client query submissions and executes them with the help of the Hive driver library. When the HiveServer receives a query, it first tries to parse it, then generates an execution plan using the metadata present in the **Metastore** and optimizes it. The execution plan consists of a directed acyclic graph of Map/Reduce jobs. Finally, Hive submits the generated Map/Reduce jobs to a configured execution engine and monitors the execution of the jobs until they are complete. Hive supports three execution engines: Map/Reduce (YARN), Tez, and Spark. By default, it uses the Map/Reduce engine. However, for faster (in-memory) query execution, it is recommended to use Tez (for Hadoop version 2) or Spark.

- The **beeline** client is a command-line interface used to connect to the HiveServer2 and submit HiveQL queries.

One of Hive's valuable additional services is **Hcatalog** which works on top of the **Metastore** and provides an API to access Hive databases and tables for other tools such as Pig or Flink.

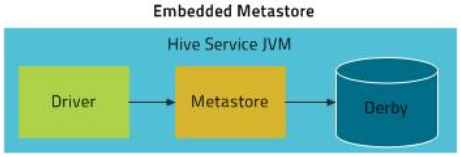

Figure 2.8: Hive Architecture with Embedded Metastore and Database[36]

[36]Source : http://www.cloudera.com/documentation/cdh/5-1-x/CDH5-Installation-Guide/cdh5ig_hive_metastore_configure.html

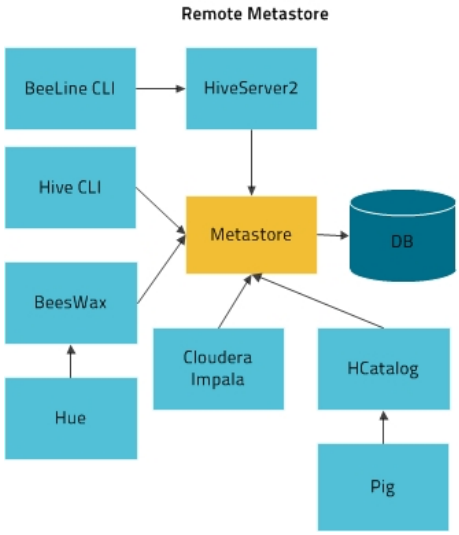

Figure 2.9: Hive architecture with Metastore in server mode and external database

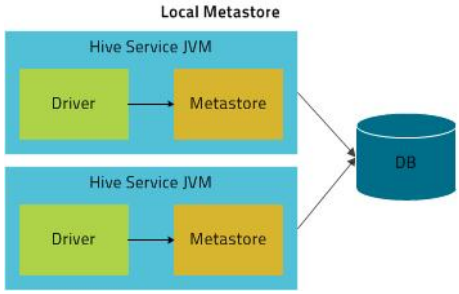

Figure 2.10: Hive architecture with embedded Metastore and external database

2.8.1 Installing Hive 3.1.2 on a Hadoop cluster

This section will demonstrate how to install Hive on an existing (previously configured) Hadoop cluster. As shown in Figure 2.11, we will install Hive on the Master Node. Regarding the Metastore service, we will configure it to run in server mode with an external MySQL database.

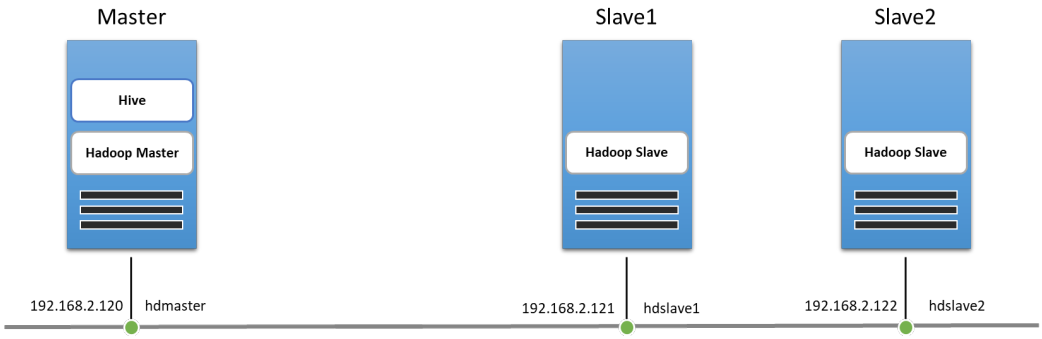

Figure 2.11: Hive installation on the Hadoop cluster

2.8.1.1 Prerequisites

For this Hive configuration, the Hadoop cluster installation must already be in place, with all necessary configurations (see section 2.5).

2.8.1.2 Hive 3.1.2 installation

The installation process for Hive is quite similar to Hadoop. We will download the pre-compiled Hive sources archive from the official website (https://hive.apache.org/) into the "/usr/local" directory using the "wget" shell command. Here we use Hive version 3.1.2.

```
cd /usr/local/
sudo wget https://dlcdn.apache.org/hive/hive-3.1.2/apache-hive-3.1.2-bin.tar.gz
```

As for Hadoop, we will unzip the downloaded archive, give the Hadoop user ownership over the directory and create a handy alias to simplify the access to the directory.

```
sudo tar -zxf apache-hive-3.1.2-bin.tar.gz
sudo chown hdoop:hdoop -R /usr/local/apache-hive-3.1.2-bin
sudo ln -s /usr/local/apache-hive-3.1.2-bin /usr/local/hive
```

Before continuing the actual Hive installation, we will prepare our environment. First, we will install the MySQL database for the Hive Metastore on the Master node. Then, we will prepare relevant HDFS directories required by Hive.

Installing MySQL for Hive Metastore on Ubuntu 20.04.1

To install MySQL on the master node we will execute the following command in the terminal:

```
sudo apt-get install mysql-server
```

After installation, the MySQL service should be up and running. To double check, we can use the following command:

```
sudo systemctl status mysql
```

We should also install the java connector for the MySQL database so the Hive Metastore service would know how to communicate with it. For that, we can refer to the official MySQL website to fetch the **mysql-connector-java** deb package and install it. We will also create a symbolic link of the connector in the /usr/local/hive/lib directory to give Hive direct access.

```
wget https://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java_8.0.26-
sudo dpkg -i ./mysql-connector-java_8.0.26-1ubuntu20.04_all.deb
sudo ln -s /usr/share/java/mysql-connector-java-8.0.26.jar /usr/local/hive/lib/
```

Next, let us secure our MySQL installation:

```
sudo mysql_secure_installation
```

The command will prompt a couple of questions to which we respond like shown below:

```
...
Would you like to setup VALIDATE PASSWORD component?: y
Please enter 0 = LOW, 1 = MEDIUM and 2 = STRONG: 2
Please set the password for root here.
New password: ************
Re-enter new password:  ************
Estimated strength of the password: 100
Do you wish to continue with the password provided?: y
Remove anonymous users?: y
Disallow root login remotely?: y
Remove test database and access to it?: y
Reload privilege tables now?: y
All done!
```

After this, we will update the "bind-address" of the MySQL server in the "mysqld.cnf" file with its local IP address. Setting up the "bind-address"

property enables us to connect to the MySQL server from other machines in the (local) network.

```
sudo vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

```
#find this line and change localhost to your server-ip
bind-address            = 192.168.2.120
```

For this change to take effect, it is required to restart the MySQL service. However, beforehand, we will log in as root to MySQL and create a Hive user with a metastore database.

```
sudo mysql -uroot
```

```
CREATE DATABASE IF NOT EXISTS metastore;
USE metastore;
CREATE USER 'hive'@'' IDENTIFIED BY 'hive!Hive1';REVOKE ALL
    PRIVILEGES, GRANT OPTION FROM 'hive'@'GRANT ALL PRIVILEGES ON
    metastore.* TO 'hive'@'FLUSH PRIVILEGES;quit;
```

Now we will restart the MySQL service:

```
sudo systemctl restart mysql
```

To check that the hive user can connect to the database, we will use the following command:

```
mysql -h hdmaster -u hive -p
Enter password: hive!Hive1
quit
```

Create Hive directories in HDFS There are two directories in HDFS that our Hive installation requires. One is the Hive scratch directory, used for storing temporary data related to execution outputs and plans. By default, it points to "/tmp" in HDFS. The second is the metastore data warehouse directory, used as the default HDFS directory to store Hive's databases and tables data files. For that purpose, Hive uses the "/user/hive/warehouse" HDFS directory by default. We will not change these defaults in our current installation, although we could, by setting new values in the Hive configuration file "hive-site.xml" located in the Hive installation directory at "/usr/local/hive/conf/hive-site.xml". However, we have to create these two directories in advance and set appropriate permissions for them.

```
hadoop fs -mkdir -p /tmp
hadoop fs -mkdir -p /user/hive/warehouse
hadoop fs -chmod g+w /tmp
hadoop fs -chmod g+w /user/hive/warehouse
```

Setting up Hive's environment variables Some of Hive's configurations rely on environment variables. To set these variables at each login, we will switch to the Hadoop user and define them in the ".profile" at the $HOME directory.

```
sudo su hdoop
vim /home/hdoop/.profile
```

```
#Add to the end of this file
# -- HIVE ENVIRONMENT VARIABLE START -- #
export HIVE_HOME=/usr/local/hive
export METASTORE_HOME=$HIVE_HOME
export PATH=$PATH:$HIVE_HOME/bin
# -- HIVE ENVIRONMENT VARIABLE END -- #
```

To make the change in profile file effective immediately, we could use the next command:

```
source /home/hdoop/.profile
```

Configure Hive

Up until now, we have avoided configuring Hive via its main configuration file "/usr/local/hive/conf/hive-site.xml" by relying on the default values. However, at this stage, we need to update it to make the Hive Metastore use our previously installed MySQL database.

By default, there is no "hive-site.xml" file present in the Hive configuration directory (/usr/local/hive/conf). We will have to create it ourselves. Fortunately, Hive provides a template with various configuration properties set to their defaults, which we can use to bootstrap the "hive-site.xml" file and update it to our needs.

```
cd /usr/local/hive/conf
cp hive-default.xml.template hive-site.xml
vim hive-site.xml
```

In the hive-site.xml file, we now need to add the **system:java.io.tmpdir** property to indicate Hive's default temporary directory on HDFS. Furthermore, we will edit the following properties:

- The **javax.jdo.option.ConnectionURL** property to point to our MySQL database.
- The **javax.jdo.option.ConnectionDriverName** property to inidicate the required JDO driver to use for the database connection.
- The **javax.jdo.option.ConnectionUserName** property to specify the username for the MySQL connection.
- The **javax.jdo.option.ConnectionPassword** property to specify the password for the MySQL connection.
- The **hive.metastore.uris** property to run the metastore as an external service.

```
<property>
    <name>system:java.io.tmpdir</name>
    <value>/tmp</value>
</property>
<property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:mysql://hdmaster:3306/metastore</value>
    <description>JDBC connect string for a JDBC metastore</
        ↪ description>
</property>
<property>
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>com.mysql.cj.jdbc.Driver</value>
    <description>Driver class name for a JDBC metastore</
        ↪ description>
</property>
<property>
    <name>javax.jdo.option.ConnectionUserName</name>
    <value>hive</value>
    <description>Username to use against metastore database</
        ↪ description>
</property>
<property>
    <name>javax.jdo.option.ConnectionPassword</name>
    <value>hive!Hive1</value>
```

```
    <description>password to use against metastore database</
        ↪ description>
</property>
<property>
    <name>hive.metastore.uris</name>
    <value>thrift://hdmaster:9083</value>
    <description>Thrift URI for the remote metastore. Used by
        ↪ metastore client to connect to remote metastore.</
        ↪ description>
  </property>
  <property>
    <name>hive.server2.logging.operation.level</name>
    <value>VERBOSE</value>
    <description>none, execution, performance, verbose</
        ↪ description>
  </property>
```

Another valuable configuration of Hive is named **hive.server2.enable.doAs** which controls if Hive should run received queries on behalf of the user that submitted them to Hive or if Hive should run them with the user who started the Hiveserver2. The default of this value is **true**, which means that when a client submits a query to Hive, the user that runs the HiveServer2 service will impersonate itself as the submitting user and execute the query on his behalf. However, a user that accesses HDFS and runs MapReduce jobs on behalf of another user needs special permission to do so on Hadoop. We will give this permission to our **hdoop** user, which will also run the HiveServer2 by adding the following lines to the "core-site.xml" Hadoop configuration file located at **/usr/local/hadoop/etc/hadoop/core-site.xml**.

```
<property>
    <name>hadoop.proxyuser.hdoop.hosts</name>
    <value>hdmaster</value>
</property>
<property>
    <name>hadoop.proxyuser.hdoop.groups</name>
    <value>hdoop</value>
</property>
```

2.8.1.3 Starting Hive services

To connect, create and manipulate data on Hive via the Beeline client, we need first to launch its Metastore and HiveServer2 services. However, we

should ensure that the Hadoop daemons (HDFS and Yarn) are up and running before the launch. Also, before launching the Hive Metastore for the first time, it is required to initialize the database schema. As we have added the Hive bin directory in our path, we can use Hive's **schematool** utility for this task.

```
schematool -initSchema -dbType mysql
```

After the initialization finishes, we can start the Hive Metastore service with the following command in the terminal as the Hadoop user:

```
nohup hive --service metastore > /dev/null &
```

In this command, we use the **nohub** command to prevent the metastore service from stopping when we log out of the system. Also, note that we are discarding the output of **nohub** by redirecting it to **/dev/null** because Hive already handles the persistence of his logs, and by default, it writes them in the **/tmp/username/hive.log** file on the local machine.

Next, we will start the HiveServer2 service in a similar manner:

```
nohup hiveserver2 > /dev/null &
```

At this stage, we can launch the Hive's beeline client to connect and manipulate the Hive database with the **hdoop** user.

```
beeline
```

```
beeline> !connect jdbc:hive2://192.168.2.120:10000/ hdoop ""
```

Finally, to try out some commands on Hive we can write the following in the beeline prompt:

```
beeline> show databases;
beeline> use default;
beeline> show tables;
```

The "show databases;" command returns the list of Hive databases. « use default » selects the database named default as the default database. « show tables » returns a list of tables in the currently selected database.

2.8.1.4 Simple Hive usage example

The data for this example comes from the "MovieLens 100K" Dataset [37], made publicly available by the "GroupLens" research lab. (See https://grouplens.org/datasets/ for more information). We are interested in the **u.data** file, which consists of a tab-separated list of movie ratings made by various users. Each line of the file contains the movie ID, the user ID, the rating, and a timestamp. In this example, we will create a "movielens" database containing a "m_rating" table pointing to this file on HDFS.

To start, we will log in as the Hadoop user, download the dataset on our local filesystem, unzip it and load it to HDFS:

```
su - hdoop
wget  http://files.grouplens.org/datasets/movielens/ml-100k.zip
unzip  ml-100k.zip
hadoop fs -put ml-100k
```

Next, we will connect to Hive using the beeline client, and create the **movielens** database:

```
beeline
beeline>!connect jdbc:hive2://192.168.2.120:10000/ hdoop ""
beeline>CREATE DATABASE IF NOT EXISTS movielens;
```

Then, we will switch to the **movielens** database:

```
beeline>USE movielens;
```

After that we will create a table named **rating** with the following HiveQL statement:

```
CREATE EXTERNAL TABLE rating (
        user_id INT,
        movie_id INT,
        rating INT,
        rating_timestamp STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE;
```

[37] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

This statement specifies the fields (user_id, movie_id, rating, timestamp) and their types (INT, STRING) of the Hive table. It also indicates the table-related data source (STORED AS TEXTFILE) and format (ROW FORMAT DELIMITED FIELDS TERMINATED BY '').

With the table schema defined, it is now time to fill it with the data contained in the **u.data** file.

```
LOAD DATA INPATH '/user/hdoop/ml-100k/u.data'
OVERWRITE INTO TABLE rating;
```

Note that the "LOAD DATA INPATH" statement will move the '/user/hdoop/ml-100k/u.data' to the '/user/hive/warehouse/movielens.db/rating/' directory.

We can now use the following query to check whether we have loaded the rows into the table. The query should output the first ten results or the rating table.

```
beeline> SELECT * FROM rating LIMIT 10;
```

It would be interesting to know the average rating for each movie in decreasing order. To achieve that, we can use the following query.

```
beeline> SELECT movie_id, avg(rating) AS avg_rating FROM rating
    ↪ GROUP BY movie_id ORDER BY avg_rating DESC;
```

The result should be similar to this:

```
+-----------+--------------------+
| movie_id  |     avg_rating     |
+-----------+--------------------+
| 1467      | 5.0                |
| 1599      | 5.0                |
| 1201      | 5.0                |
| 1500      | 5.0                |
| 1293      | 5.0                |
| 1189      | 5.0                |
| 814       | 5.0                |
| 1653      | 5.0                |
| 1122      | 5.0                |
| 1536      | 5.0                |
| 1449      | 4.625              |
| 1594      | 4.5                |
| 119       | 4.5                |
| 1398      | 4.5                |
| 1642      | 4.5                |
```

2.9 Spark

Apache Spark is a framework written in Scala to perform data analysis using in-memory data processing in a distributed environment.

Spark and Hadoop form a pair often used by companies for processing and analyzing data stored in HDFS. Spark is best suited for near real-time processing (high processing speed) and advanced analytics. In contrast, Hadoop excels in storing data ranging from structured to unstructured and running batch processes for processing. This pairing takes advantage of the storage capacity of Hadoop and the processing and analysis speed of Spark.

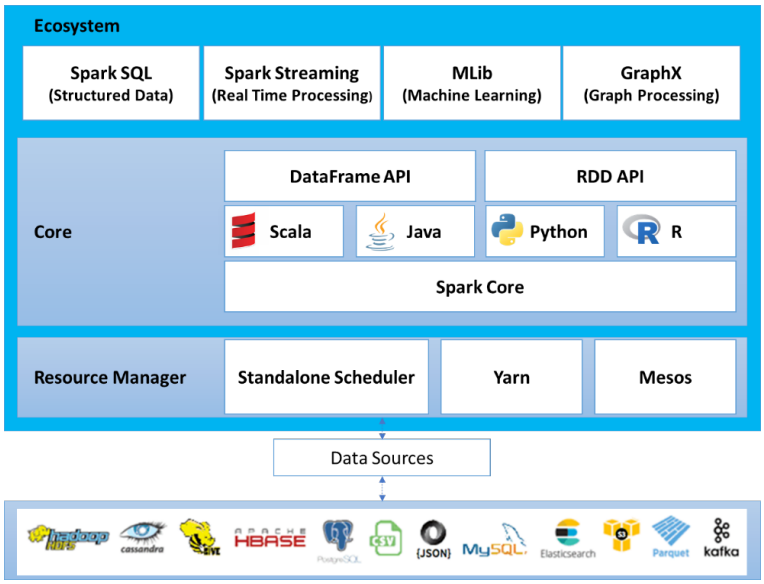

Figure 2.12: Spark Ecosystem

The Spark ecosystem is composed of several Services, APIs, and high-level libraries (see Figure 2.12), which work around the Spark Core.

Spark provides programmers with a set of libraries to build an application that can be run on a Spark cluster and produce code in a language that best suits the programmer. Among the available libraries are MLib for Machine Learning, GraphX for parallel graph data processing, and SparkR for working with the R language in the Spark cluster environment. Other languages added to the list include Java, Scala, Python, and SQL.

Spark offers two APIs, DataFrames and Resilient Distributed Datasets (RDDs). DataFrames (untyped API) and DataSets (typed API) provide

a view of how the user will operate in Spark. To a user, they are tables represented by rows and columns distributed in memory across the cluster. For Spark, they are immutable, slowly evolving tables. Spark handles the processing of structured data with SQL or through Dataframes and DataSets. RDDs, in turn, are the lowest level abstraction of Spark, representing an immutable, partitioned collection of elements used for parallel processing on a cluster. As RDDs are the core abstraction, when compiling the code, DataFrames or DataSets are transformed into RDDs.

Spark SQL is a module for working with structured data. Spark SQL is used to execute SQL queries or through the DataFrame API, which works with Java, Python, Scala, and R. The combination of SQL and DataFrames provides a common way to access various data sources such as Hive, Avro, Parquet, ORC, JSON, CSV, JDBC, and ODBC.

Spark Streaming is a module in Spark for building real-time data processing applications. Streaming provides the ability to design interactive analysis, monitoring, and detection applications. It also allows the reuse of the same batch processing code to perform joins of broadcast data (in real-time acquisition) and historical data or to run queries on broadcast (real-time) data.

In a cluster, Spark requires a resource manager (or cluster manager) to optimize, monitor, control, and assign execution of tasks on the cluster. It supports four types of resource managers in a cluster configuration (Figure 2 - 14). In Standalone or Native Cluster Manager mode, Spark uses its integrated resource manager. In "YARN" mode, Spark uses Hadoops resource manager. In "Mesos" mode, Spark runs under an independent Apache Mesos resource manager. In "Kubernetes" mode, Spark uses the Kubernetes scheduler as its resources manager. Note that Spark also has a "Local" mode in which it executes in a non-distributed manner. In this mode, all its processes are in a single JVM on a single machine, ideal for testing and debugging.

In a distributed Apache Spark architecture (see Figure 2.14), a Spark application runs as a set of processes on the cluster. There are two principal types of Spark processes - one primary process called Driver and multiple spark executor processes. The Driver process is similar to Hadoops Application Master. When we submit a Spark application to the cluster manager, it will start a container to run the Driver process. The Driver assesses the required resources and plans the execution of the application on the cluster.

[38]Source : `https://spark.apache.org/docs/latest/cluster-overview.html`

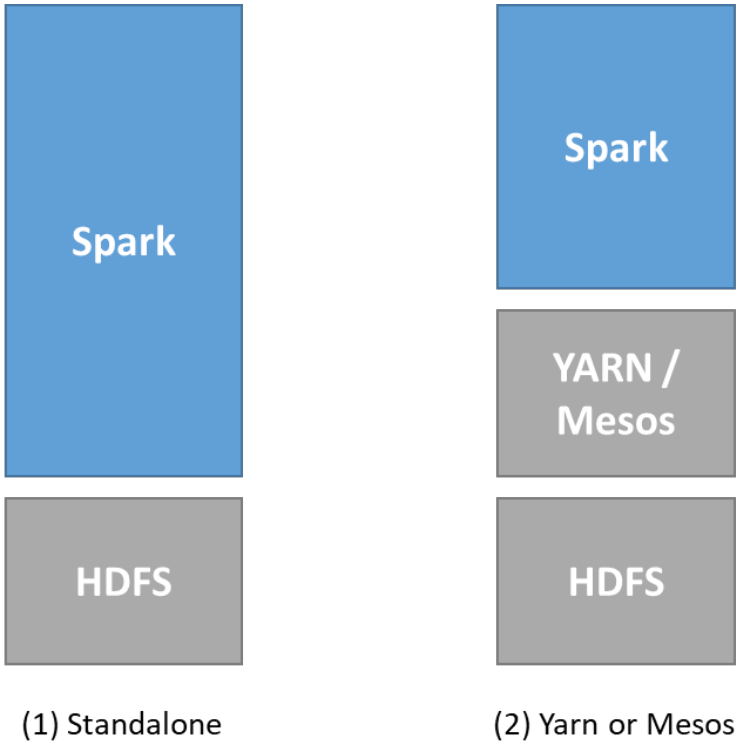

Figure 2.13: Spark deployment modes

Then, it requests the resource manager to launch the required amount of spark executors. Next, it submits application-related tasks to the executors and monitors their execution until they finish.

2.9.1 Installing Apache Spark 3.1.2

This section shows how to install Spark on an existing (previously configured) Hadoop cluster. We will use YARN from our previous Hadoop installation as the Spark resource manager, and all three Hadoop worker nodes will also act as Spark workers. This setup implies that Spark and Hadoop will share the same resources (RAM, CPU, disk). As the resource manager for both, YARN is responsible for balancing the resource allocation on the Worker Nodes.

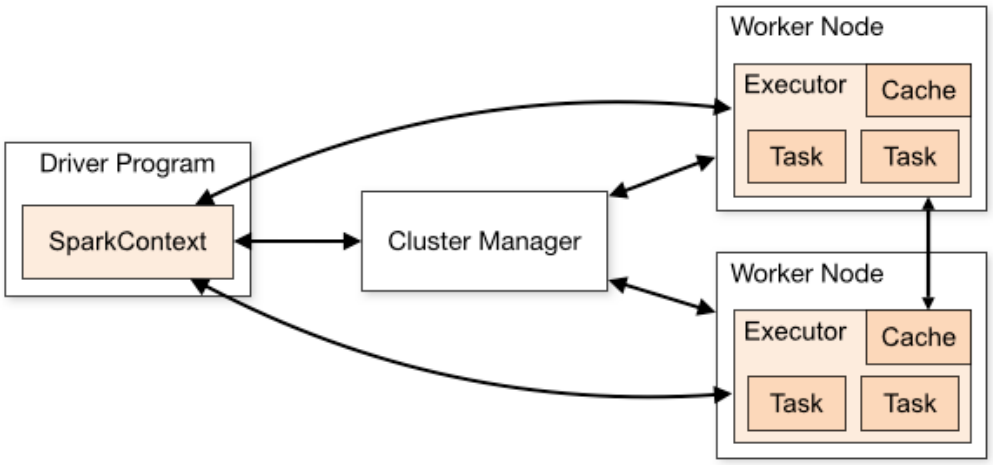

Figure 2.14: Spark cluster Architecture [38]

2.9.1.1 Prerequisites

As we install Spark on YARN, a Hadoop cluster installation must already be in place, with all the necessary setup presented in section 2.5. In our examples, we will use Sparks python API. Therefore we will also have to install Python on our machines.

2.9.1.2 Installing Python 3.9

This step is optional as Ubuntu 20.04.1 ships with Python 3.8 pre-installed. However, in our case, we would prefer to use Python 3.9. To install Python 3.9 on an Ubuntu machine, it is sufficient to run the following command:

```
sudo apt install python3.9
```

We will run this command on all nodes of the cluster.

2.9.1.3 Spark installation

Spark's binaries are available at its official website (https://spark.apache.org/). We will install Spark alongside Hadoop in the "/usr/local" directory on all cluster nodes.

```
cd /usr/local
```

Download Spark binaries:

```
sudo wget https://www.apache.org/dyn/closer.lua/spark/spark
    ↪ -3.1.2/spark-3.1.2-bin-hadoop3.2.tgz
```

Untar Spark: (Note that we also give our Hadoop user ownership of the installation directory, and we also create a handy alias to ease access to the Spark installation directory).

```
sudo tar zxvf spark-3.1.2-bin-hadoop3.2.tgz
sudo chown -R hdoop:hdoop spark-3.1.2-bin-hadoop3.2
sudo ln -s spark-3.1.2-bin-hadoop3.2 spark
```

Setup environment variables

Next, it is required to set some environment variables that Spark uses. For that, we will switch to the Hadoop user and add them to the profile file. We perform this step on all machines in the cluster.

Switch to the Hadoop user:

```
su - hdoop
```

Edit the "/home/hdoop/.profile" and add these Spark environment variables at the end:

```
nano /home/hdb/.bashrc
```

```
# -- Spark ENVIRONMENT VARIABLES START -- #
export SPARK_HOME=/usr/local/spark
export PATH=$PATH:$SPARK_HOME/bin
# -- Spark ENVIRONMENT VARIABLES END -- #
```

To make the change in profile file effective immediately, we could use the next command:

```
source /home/hdoop/.profile
```

To check if the setup was successful, we could try to launch the interactive python spark-shell:

```
pyspark
```

We should obtain a command prompt waiting for our input. To exit the python spark-shell, we could type the following line and press Enter.

```
quit()
```

2.9.1.4 Simple Python Spark example with Yarn

In this section, we will show a Spark word count MapReduce example in Python. In this example, our Spark application will be a client of Hadoop's YARN resource manager. We will use the previously created files in the /user/hdoop/example directory on HDFS (**test_file_1.txt** and **test_file_2.txt**).

Pass this command to launch the Python Spark-Shell as a client of YARN on a cluster node.

```
pyspark --master yarn --deploy-mode client
```

After running the pyspark-shell as a yarn client, type the word counter code shown below in the Python Spark prompt.

```
lines = sc.textFile("hdfs:///user/hdoop/example")
words = lines.flatMap(lambda x: x.split())
tuples = words.map(lambda x: (x, 1))
count_by_word = tuples.reduceByKey(lambda a, b: a + b)
count_by_word.saveAsTextFile("hdfs:///user/hdoop/output_ex_spark"
    ↪ )
```

The first line of the Spark program indicates the directory from where to load the input data. Each line present in the files will become an item of the "lines" RDD. Then, with the flatMap method, we split each item by space and create a new RDD where each item is a word. Next, we use the map method to transform each word into a tuple. The tuple holds the original word as the key and the integer one as the value. Then we group all resulting tuples by their key and sum up the values for each group with the reduceByKey function. Finally, we save our result on HDFS using the saveAsTextFile function.

Spark will store the results in the output directory "/user/hdoop/output_ex_spark" in files named from part-00000 to part-XXXXX. The number of part-XXXXX files depends on the number of partitions the final RDD had.

To display our results we could run the following command in the shell :

```
hadoop fs -cat  /user/hdoop/output_ex_spark/*
```

```
(Bye,1)
(Welcome,1)
(Hello,2)
(Yarn,2)
(HDFS,1)
```

(Hadoop , 1)

2.10 Chapter 2 References

How Hadoop splits input data: https://stackoverflow.com/questions/14291170/how-does-hadoop-process-records-split-across-block-boundaries https://stackoverflow.com/question hadoop-hdfs-file-splitting https://stackoverflow.com/questions/32876408/relation-between-number-of-input-splits-and-number-of-mappers-in-mapreduce-hadoo https://stackoverflow.com/questions/10719191/hadoops-input-splitting-how-does-it-work https://www.netjstech.com/2018/05/input-splits-in-hadoop.html https://www.netjstech.com/2018/04/data-locality-in-hadoop.html

[1]T. White, "Hadoop: the definitive guide" [storage and analysis at Internet scale], 4. ed., Updated. Beijing: O'Reilly, 2015.

[2] Apache Hadoop official Documents [En Ligne] Disponible: https://wiki.apache.org/hado

[1]Manish A. Kukreja, "Apache Hive: Enterprise SQL on Big Data frameworks" Unpublished, 2016.

[1]A. Thusoo et al., "Hive - a petabyte-scale data warehouse using Hadoop" 2010, p. 996-1005.

[1]A. Thusoo et al., "Hive: a warehousing solution over a map-reduce framework" Proceedings of the VLDB Endowment, vol. 2, no 2, p. 1626-1629, août 2009.

[1]E. Capriolo, D. Wampler, et J. Rutherglen, Programming Hive: [data warehouse and query language for Hadoop], 1. ed. Beijing: O'Reilly, 2012.

[1]S. Ryza, U. Laserson, S. Owen, et J. Wills, Advanced analytics with Spark, First edition. Beijing; Sebastopol, CA: O'Reilly, 2015.

[1]B. Chambers et M. Zaharia, Spark, the definitive guide: big data processing made simple. 2017.

[1]J. Dean et S. Ghemawat, "MapReduce: simplified data processing on large clusters" Communications of the ACM, vol. 51, no 1, p. 107-113, Jan. 2008.