Getting Insight from Big Data with Hadoop and Its Ecosystem

Gandhi Manalu
(Big Data Engineer @ Think Big Analytics)

A Workshop at Institut Teknologi Del
Saturday, 15 October 2016
# Table of Contents

A.1 Hadoop Architecture .................................................................................................................. 4
  A.1.1 NameNode ............................................................................................................................... 4
  A.1.2 Secondary NameNode ............................................................................................................ 5
  A.1.3 JobTracker ........................................................................................................................... 5
  A.1.4 TaskTracker .......................................................................................................................... 5

A.2 Hadoop Framework Components ............................................................................................... 5
  A.2.1 Hadoop Common .................................................................................................................... 5
  A.2.2 Hadoop Distributed File System .......................................................................................... 5
  A.2.3 MapReduce ........................................................................................................................... 6
  A.2.4 YARN ...................................................................................................................................... 6

B.0 Software Requirements ............................................................................................................... 8

B.1 Environment Setup ...................................................................................................................... 8

B.2 Administering Hadoop Cluster .................................................................................................. 14

B.3 Distributed Data Storage with HDFS ...................................................................................... 16
  B.3.1 Operational Commands ..................................................................................................... 16
    B.3.1.1 Listing Contents of a Directory ....................................................................................... 17
    B.3.1.2 Creating a Directory in HDFS ......................................................................................... 18
    B.3.1.3 Copy a Local File/Directory to HDFS ........................................................................... 18
    B.3.1.4 Read a File in HDFS ....................................................................................................... 18
    B.3.1.5 Copy a File/Directory within HDFS ............................................................................... 19
    B.3.1.6 Move a File/Directory within HDFS ............................................................................... 19
    B.3.1.7 Copy an HDFS File/Directory to Local ......................................................................... 19
    B.3.1.8 Remove a File/Directory in HDFS .................................................................................. 20
  B.3.2 Administrative Commands .................................................................................................. 20
    B.3.2.1 Checking HDFS Capacity ............................................................................................. 20
    B.3.2.2 Checking Usage of a Directory in HDFS ........................................................................ 21
    B.3.2.3 Setting Quota on a directory in HDFS ............................................................................ 22

B.4 Distributed Data Processing with MapReduce .......................................................................... 22
  B.4.1 MapReduce Program with Java ........................................................................................... 22
  B.4.2 MapReduce Program with Hadoop Streaming and Python .................................................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.5 Querying Data with Apache Hive (and Impala)</td>
<td>26</td>
</tr>
<tr>
<td>B.5.1 Loading Data into Apache Hive</td>
<td>27</td>
</tr>
<tr>
<td>B.5.1.1 Upload Datasets File into HDFS</td>
<td>27</td>
</tr>
<tr>
<td>B.5.1.2 Create Empty Hive Table</td>
<td>29</td>
</tr>
<tr>
<td>B.5.2 Querying Data in Apache Hive</td>
<td>35</td>
</tr>
<tr>
<td>B.5.2.1 Simple Query</td>
<td>35</td>
</tr>
<tr>
<td>B.5.2.2 Join Query</td>
<td>35</td>
</tr>
<tr>
<td>B.5.3 Querying Data in (Apache) Impala</td>
<td>35</td>
</tr>
<tr>
<td>B.5.4 Using Hive to Store and Query XML Data</td>
<td>37</td>
</tr>
<tr>
<td>B.6 Distributed Data Processing with Apache Spark</td>
<td>40</td>
</tr>
<tr>
<td>B.6.1 Creating RDD from a File and Writing RDD to HDFS</td>
<td>41</td>
</tr>
<tr>
<td>B.6.2 Performing Operations</td>
<td>42</td>
</tr>
<tr>
<td>B.7 Manipulating Structured Data Using Apache Spark</td>
<td>43</td>
</tr>
<tr>
<td>B.7.1 DataFrame Operation with JSON File</td>
<td>43</td>
</tr>
<tr>
<td>B.7.2 Specifying a Schema Programmatically</td>
<td>44</td>
</tr>
<tr>
<td>B.7.3 DataFrame Operation with Hive Table</td>
<td>45</td>
</tr>
<tr>
<td>B.8 Introduction to Machine Learning with Apache Spark</td>
<td>45</td>
</tr>
</tbody>
</table>
A.1 Hadoop Architecture
Apache Hadoop is an open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common and should be automatically handled by the framework.

The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part called MapReduce.

A.1.1 NameNode
Namenode is the node which stores the filesystem metadata i.e. which file maps to what block locations and which blocks are stored on which datanode.

The namenode maintains two in-memory tables, one which maps the blocks to datanodes (one block maps to 3 datanodes for a replication value of 3) and a datanode to block number mapping. Whenever a datanode reports a disk corruption of a particular block, the first table gets updated and whenever a datanode is detected to be dead (because of a node/network failure) both the tables get updated.

Two different files are:

1. Fsimage: the snapshot of the filesystem when namenode started
2. Edit logs: the sequence of changes made to the filesystem after namenode started
A.1.2 Secondary NameNode
Secondary NameNode helps NameNode by taking over responsibility of merging editlogs with fsimage from the NameNode.

A.1.2 DataNode
The data node is where the actual data resides.

All datanodes send a heartbeat message to the namenode every 3 seconds to say that they are alive. If the namenode does not receive a heartbeat from a particular data node for 10 minutes, then it considers that data node to be dead/out of service and initiates replication of blocks which were hosted on that data node to be hosted on some other data node.

The data nodes can talk to each other to rebalance data, move and copy data around and keep the replication high.

A.1.3 JobTracker
The primary function of the job tracker is resource management (managing the task trackers), tracking resource availability and task life cycle management (tracking its progress, fault tolerance etc.)

A.1.4 TaskTracker
The task tracker has a simple function of following the orders of the job tracker and updating the job tracker with its progress status periodically.

The task tracker is pre-configured with a number of slots indicating the number of tasks it can accept. When the job tracker tries to schedule a task, it looks for an empty slot in the tasktracker running on the same server which hosts the datanode where the data for that task resides. If not found, it looks for the machine in the same rack. There is no consideration of system load during this allocation.

A.2 Hadoop Framework Components
In general, Apache Hadoop comprises of the four components:

A.2.1 Hadoop Common
Hadoop common is a set of common libraries and utilities used by other Hadoop modules. Apache Foundation has pre-defined set of utilities and libraries that can be used by other modules within the Hadoop ecosystem. For example, if HBase and Hive want to access HDFS they need to make of Java archives (JAR files) that are stored in Hadoop Common.

A.2.2 Hadoop Distributed File System
The default big data storage layer for Apache Hadoop is HDFS. HDFS operates on a Master-Slave architecture model where the NameNode acts as the master node for keeping a track of the storage cluster and the DataNode acts as a slave node summing up to the various systems within a Hadoop cluster.
HDFS component creates several replicas of the data block to be distributed across different clusters for reliable and quick data access.

### A.2.3 MapReduce

MapReduce is a Java-based system created by Google where the actual data from the HDFS store gets processed efficiently. In the Hadoop ecosystem, Hadoop MapReduce is a framework based on YARN architecture. YARN based Hadoop architecture, supports parallel processing of huge data sets and MapReduce provides the framework for easily writing applications on thousands of nodes, considering fault and failure management.

![MapReduce Diagram](image.png)

**Figure 2 MapReduce**

### A.2.4 YARN

YARN is one of the key features in the second-generation Hadoop 2 version of the Apache Software Foundation's open source distributed processing framework. Originally described by Apache as a redesigned resource manager, YARN is now characterized as a large-scale, distributed operating system for big data applications.

YARN is a resource manager that was created by separating the processing engine and resource management capabilities of MapReduce as it was implemented in Hadoop 1. YARN is often called the operating system of Hadoop because it is responsible for managing and monitoring workloads,
maintaining a multi-tenant environment, implementing security controls, and managing high availability features of Hadoop.

Figure 3 YARN
B.0 Software Requirements
In order to be able to run steps and procedures describe in this document, the following list of software are required:

b. Cloudera Distribution for Hadoop 5.5.0 Quickstart virtual machine: [https://downloads.cloudera.com/demo_vm/vmware/cloudera-quickstart-vm-5.5.0-0-vmware.zip](https://downloads.cloudera.com/demo_vm/vmware/cloudera-quickstart-vm-5.5.0-0-vmware.zip)
c. MobaXterm: [http://mobaxterm.mobatek.net/MobaXterm_Setup_9.3.msi](http://mobaxterm.mobatek.net/MobaXterm_Setup_9.3.msi)

The following datasets will be used in this workshop:

a. [https://raw.githubusercontent.com/hortonworks/tutorials/hdp-2.5/driver_data.zip](https://raw.githubusercontent.com/hortonworks/tutorials/hdp-2.5/driver_data.zip)

B.1 Environment Setup
This chapter onwards will address technical aspects of Hadoop platform. Workshop participants will learn about Hadoop platform from the ground up. Starting with setting up and administer the Hadoop environment, up to developing Big Data application using various tools in Hadoop ecosystem.

The Hadoop distribution used in this workshop is Cloudera Distribution for Hadoop (CDH), specifically version 5.5. In this workshop, for ease of operation, a Quickstart will be used as the main working environment. However, most of the subjects in this workshop will be applicable to full-fledged Hadoop cluster or other Hadoop distribution with little to none modification. (Note: Operating system in which the VMware is running will be called “host”, while the operating system of the CDH 5.5 being used will be called “guest”).

This workshop assumes a VMware Player (or Workstation) is already installed. Follow the following steps to setup CDH 5.5 on VMware Player:

1) Extract the CDH 5.5 virtual machine image, cloudera-quickstart-vm-5.5.0-0-vmware.zip. Assume the location of the extracted image is C:\Users\USERNAME\Documents\cloudera-quickstart-vm-5.5.0-0-vmware
2) Start the VMware Player
3) Click **File -> Open** and choose C:\Users\USERNAME\Documents\cloudera-quickstart-vm-5.5.0-0-vmware\cloudera-quickstart-vm-5.5.0-0-vmware.vmx
4) Click ![Start](start_icon.png) to start the CDH 5.5 virtual machine (or click **VM -> Power -> Start Up Guest**)
5) The CDH 5.5 virtual machine will be starting. This will take a while. Notice that the operating system used by CDH 5.5 is a GNU/Linux Centos 6.4 distribution.
6) The following is the final result after CDH 5.5 virtual machine is up and running.

Figure 4 CDH 5.5 Virtual Machine is starting up

Figure 5 CDH 5.5 Virtual Machine is up and running
Go on and play around the CDH 5.5 virtual machine. For starter, you can check the Hadoop version:

1) Click 🖱️ on the CDH 5.5 virtual machine to open the terminal
2) Type “hadoop version” to check Hadoop version on CDH 5.5 virtual machine. You will see something similar with this:

![Figure 6 Checking the Hadoop version](image)

As we can see from Figure 3, the Hadoop version is 2.6.0 on CDH 5.5.0 distribution. You can also play with other GNU/Linux commands you are familiar with. For example: “df -h”, “du”, “date”, etc to name a few.

The default time zone used by the CDH 5.5 virtual machine is PDT time zone. Follow the following steps to change the time zone into WIB time zone (UTC+7:00):

1) Open the terminal
2) Type “sudo rm -f /etc/localtime”
3) Type “sudo ln -s /usr/share/zoneinfo/Asia/Jakarta /etc/localtime”
Quickstart environment is not really convenient to access from the guest operating system; therefore we will configure client tools to access the quickstart from the host operating system. Follow the following steps to configure client tools:

1) Open the terminal
2) Type “ifconfig” to get the IP Address of the CDH 5.5 virtual machine
   You will see something similar with this:
3) Install MobaXTerm by clicking the file MobaXterm_Setup_9.3.msi and following the instruction
4) Open the MobaXTerm application
5) Click **Session** -> **New Session** choose Secure Shell (SSH) Session
6) Type the IP Address as shown in Figure 5 as the “Remote Host”, type “cloudera” as the “username”, and type “CDH-5.5.0” as the “session name” on “Bookmark settings” tab, as shown below:

![MobaXTerm Settings](image)

Then click “OK”.
7) The MobaXTerm will try to connect to the CDH 5.5 virtual machine, and once it's connected type “cloudera” as the password. This is only required for first time setup. For subsequent access, MobaXTerm will not ask for the password again.
8) Now, instead of running GNU/Linux commands or Hadoop commands by typing it on the terminal in the guest operating system, we will now run GNU/Linux commands and Hadoop commands by typing it on the MobaXTerm in the host operating system.

We will also use HUE (Hadoop User Experience) as the Apache Hadoop User Interface. Open Internet browser and type the following address: [http://192.168.100.130:8888/](http://192.168.100.130:8888/). You will see something similar with this:
Type “cloudera” as the username and “cloudera” as the password. Now just leave it that way and we will get back to use this web based application on the subsequent chapters.

**B.2 Administering Hadoop Cluster**

Cloudera provides Cloudera Manager that can be used to manage the Hadoop cluster. Open Internet browser and type the following address: [http://192.168.100.130:7180/cm/][1]. You will see something similar with this:
Log in to the page using “admin” username and “admin” password. After logging in, we will see a window similar with this one:

**Figure 12 Cloudera Manager**

Log in to the page using “admin” username and “admin” password. After logging in, we will see a window similar with this one:
Using the Cloudera Manager, we can manage available services, manage available hosts, perform administration, perform configuration changing, and many other administrative tasks.

To be able to start Cloudera Manager, minimal memory requirement is 8 GB. A demonstration will be given to show administrative tasks commonly performed on a Hadoop cluster.

**B.3 Distributed Data Storage with HDFS**

Hadoop Distributed File System (HDFS) is the distributed storage component of Apache Hadoop. In this chapter, we will learn how to put files/directories in to HDFS, how to perform operation towards the files/directories in HDFS, and also perform administrative tasks on the HDFS.

For more complete list of HDFS command, please refer to this page: [https://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-common/FileSystemShell.html](https://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-common/FileSystemShell.html).

**B.3.1 Operational Commands**

This sub-chapter will describe HDFS command for operational use.
B.3.1.1 Listing Contents of a Directory
Run the following command to list a directory “hdfs dfs -ls /path/to/directory”. For example:

```
$ hdfs dfs -ls /user
Found 9 items
```

<table>
<thead>
<tr>
<th>Mode</th>
<th>Owner</th>
<th>Group</th>
<th>Size</th>
<th>Date</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>drwxr-xr-x</td>
<td>cloudera</td>
<td>cloudera</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/cloudera</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>hdfs</td>
<td>supergroup</td>
<td>0</td>
<td>2016-10-08</td>
<td>/user/hdfs</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>mapred</td>
<td>hadoop</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/history</td>
</tr>
<tr>
<td>drwxrwxrwx</td>
<td>hive</td>
<td>supergroup</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/hive</td>
</tr>
<tr>
<td>drwxrwxrwx</td>
<td>jenkins</td>
<td>supergroup</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/jenkins</td>
</tr>
<tr>
<td>drwxrwxrwx</td>
<td>oozie</td>
<td>supergroup</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/oozie</td>
</tr>
<tr>
<td>drwxrwxrwx</td>
<td>root</td>
<td>supergroup</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/root</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>hdfs</td>
<td>supergroup</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/spark</td>
</tr>
</tbody>
</table>

**Note:** HDFS is not POSIX compliant, one implication is that HDFS doesn’t have a concept of current working directory as in GNU/Linux. However, if an absolute path is not specified, it is always assumed the default directory being referred as the “home” directory of current user running the HDFS command. In HDFS, “home” directory of current user is /user/USERNAME. For example, since we are logged in as “cloudera” user, then if we run “hdfs dfs -ls”, the command will be interpreted as “hdfs dfs -ls /user/cloudera”. See the following example to better understand the concept:

```
$ hdfs dfs -ls
$ hdfs dfs -ls /user/cloudera
```

Both commands above will list the content of the same directory.

To list the contents of a directory recursively, use the “hdfs dfs -ls -R /path/to/directory” command, as follows:

```
$ hdfs dfs -ls -R /user
```

<table>
<thead>
<tr>
<th>Mode</th>
<th>Owner</th>
<th>Group</th>
<th>Size</th>
<th>Date</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>drwxr-xr-x</td>
<td>cloudera</td>
<td>cloudera</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/cloudera</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>hdfs</td>
<td>supergroup</td>
<td>0</td>
<td>2016-10-08</td>
<td>/user/hdfs</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>hdfs</td>
<td>supergroup</td>
<td>0</td>
<td>2016-10-08</td>
<td>/user/hdfs/.Trash</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>hdfs</td>
<td>supergroup</td>
<td>0</td>
<td>2016-10-08</td>
<td>/user/hdfs/.Trash/Current</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>hdfs</td>
<td>supergroup</td>
<td>0</td>
<td>2016-10-08</td>
<td>/user/hdfs/.Trash/Current/tmp</td>
</tr>
<tr>
<td>-rw-r--r--</td>
<td>1 hdfs</td>
<td>supergroup</td>
<td>0</td>
<td>2016-10-08</td>
<td>/user/hdfs/.Trash/Current/tmp/hue_config_validation.16477409453450728108</td>
</tr>
<tr>
<td>-rw-r--r--</td>
<td>1 hdfs</td>
<td>supergroup</td>
<td>0</td>
<td>2016-10-08</td>
<td>/user/hdfs/.Trash/Current/tmp/hue_config_validation.8794248309598342547</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>mapred</td>
<td>hadoop</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/history</td>
</tr>
<tr>
<td>drwxrwxrwx</td>
<td>hive</td>
<td>supergroup</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/hive</td>
</tr>
<tr>
<td>drwxrwxrwx</td>
<td>hive</td>
<td>supergroup</td>
<td>0</td>
<td>2015-11-19</td>
<td>/user/hive/warehouse</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
B.3.1.2 Creating a Directory in HDFS

Run the following command to list a directory “hdfs dfs -mkdir /path/to/create”. For example:

```
[cloudera@quickstart ~]$ hdfs dfs -mkdir labs-itdel
[cloudera@quickstart ~]$ hdfs dfs -ls
Found 1 items
drwxr-xr-x  - cloudera cloudera 0 2016-10-08 13:14 labs-itdel

[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera
Found 1 items
drwxr-xr-x  - cloudera cloudera 0 2016-10-08 13:14
```

B.3.1.3 Copy a Local File/Directory to HDFS

We are going to copy a local file into HDFS. The local source file will be `/usr/lib/hadoop-0.20-mapreduce/README.txt` of size 1366 bytes, and the HDFS destination directory will be `/user/cloudera/labs-itdel`:

```
[cloudera@quickstart ~]$ hdfs dfs -put /usr/lib/hadoop-0.20-mapreduce/README.txt /user/cloudera/labs-itdel
[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera/labs-itdel
Found 1 items
-rw-r--r-- 1 cloudera cloudera 1366 2016-10-08 13:19
/user/cloudera/labs-itdel/README.txt
```

**Warning:** If HDFS directory “labs-itdel” is not exist, it will copy local file “README.txt” as HDFS file “/user/cloudera/labs-itdel” (note that “labs-itdel” will be simply a file and not a directory).

**Exercise:**

- Try to perform the same exact command again, what will happen?
- How can you copy a same local file into HDFS without overriding the existing file?

B.3.1.4 Read a File in HDFS

To read a file within an HDFS, run the “hdfs dfs -cat /path/to/file” command. For example:

```
[cloudera@quickstart ~]$ hdfs dfs -cat /user/cloudera/labs-itdel/README.txt
For the latest information about Hadoop, please visit our website at:

http://hadoop.apache.org/core/

and our wiki, at:

http://wiki.apache.org/hadoop/

This distribution includes cryptographic software. The country in which you currently reside may have restrictions on the import, possession, use, and/or re-export to another country, of

...etc
```
B.3.1.5 Copy a File/Directory within HDFS
To copy a file in a directory within HDFS to another directory within HDFS, run the “hdfs dfs -cp /path/directory/sourcefile /path/destinationdir/” command. For example:

```
[cloudera@quickstart ~]$ hdfs dfs -cp /user/cloudera/labs-itdel/README.txt /user/cloudera/
[cloudera@quickstart ~]$ hdfs dfs -ls
Found 2 items
-rw-r--r--  1 cloudera  cloudera  1366 2016-10-08 13:31 README.txt
drwxr-xr-x  2 cloudera  cloudera          0 2016-10-08 13:19 labs-itdel
```

Exercise:

- Try to copy the file into non-existing directory /user/cloudera/labs-itdel2 (note that the directory is without trailing “/”). What will happen?

B.3.1.6 Move a File/Directory within HDFS
To move a file in a directory within HDFS to another directory within HDFS, run the “hdfs dfs -mv /path/directory/sourcefile /path/destinationdir/” command. The original file /path/directory/sourcefile will be deleted. For example:

```
[cloudera@quickstart ~]$ hdfs dfs -mkdir labs-itdel3
[cloudera@quickstart ~]$ hdfs dfs -mv labs-itdel/README.txt labs-itdel3/
[cloudera@quickstart ~]$ hdfs dfs -ls labs-itdel/
[cloudera@quickstart ~]$ hdfs dfs -ls labs-itdel3/
Found 1 items
-rw-r--r--  1 cloudera  cloudera  1366 2016-10-08 13:19 labs-itdel3/README.txt
```

B.3.1.7 Copy an HDFS File/Directory to Local
To copy (“download”) an HDFS file to local file system, run the “hdfs dfs -get /path/directory/file” command. For example:

```
[cloudera@quickstart ~]$ hdfs dfs -get labs-itdel3/README.txt
[cloudera@quickstart ~]$ ll
total 192
-rwxrwxr-x  1 cloudera  cloudera  5315 Nov 19 2015 cloudera-manager
-rwxrwxr-x  1 cloudera  cloudera  9964 Nov 19 2015 cm_api.py
drwxrwxr-x  2 cloudera  cloudera  4096 Nov 19 2015 Desktop
drwxrwxr-x  4 cloudera  cloudera  4096 Nov 19 2015 Documents
drwxr-xr-x  2 cloudera  cloudera  4096 Oct  7 21:40 Downloads
drwxrwxr-x  9 cloudera  cloudera  4096 Feb 19 2015 eclipse
-rw-rw-r--  1 cloudera  cloudera  53652 Nov 19 2015 enterprise-deployment.json
-rw-rw-r--  1 cloudera  cloudera  50512 Nov 19 2015 express-deployment.json
-rwxrwxr-x  1 cloudera  cloudera  5007 Nov 19 2015 kerberos
drwxrwxr-x  2 cloudera  cloudera  4096 Nov 19 2015 lib
drwxr-xr-x  2 cloudera  cloudera  4096 Oct  7 21:40 Music
drwxrwxr-x  1 cloudera  cloudera  4327 Nov 19 2015 parcels
drwxr-xr-x  2 cloudera  cloudera  4096 Oct  7 21:40 Pictures
drwxr-xr-x  2 cloudera  cloudera  4096 Oct  7 21:40 Public
-rw-rw-r--  1 cloudera  cloudera  1366 Oct  8 14:03 README.txt
drwxr-xr-x  2 cloudera  cloudera  4096 Oct  7 21:40 Templates
```
Exercise:

- Try to copy the whole “labs-itdel3” directory to local file system.

B.3.1.8 Remove a File/Directory in HDFS

To remove a file in HDFS, run the “hdfs dfs –rm /path/to/file” command. For example:

```
[cloudera@quickstart ~]$ hdfs dfs -rm labs-itdel3/README.txt
16/10/08 14:08:16 INFO fs.TrashPolicyDefault: Namenode trash configuration:
Deletion interval = 0 minutes, Emptier interval = 0 minutes.
Deleted labs-itdel3/README.txt
[cloudera@quickstart ~]$ hdfs dfs -ls labs-itdel3
```

To remove a directory in HDFS, run the “hdfs dfs –rm /path/to/directory” command. For example:

```
[cloudera@quickstart ~]$ hdfs dfs -rm -r labs-itdel3
16/10/08 14:09:02 INFO fs.TrashPolicyDefault: Namenode trash configuration:
Deletion interval = 0 minutes, Emptier interval = 0 minutes.
Deleted labs-itdel3
```

B.3.2 Administrative Commands

Aside from the operational commands, HDFS also has administrative commands that can be used to administrate the HDFS, as follows:

B.3.2.1 Checking HDFS Capacity

Run the following command to check the capacity of the HDFS:

```
[cloudera@quickstart ~]$ hdfs dfs -df
Filesystem Size Used Available Use%
hdfs://quickstart.cloudera:8020 58665738240 750145536 47121735680 1%
[cloudera@quickstart ~]$ hdfs dfs -df -h
Filesystem Size Used Available Use%
hdfs://quickstart.cloudera:8020 54.6 G 715.4 M 43.9 G 1%
```

Run the following command to get a full report of the HDFS (including the capacity):

```
[cloudera@quickstart ~]$ hdfs dfsadmin -report
Configured Capacity: 58665738240 (54.64 GB)
Present Capacity: 47871168512 (44.58 GB)
DFS Remaining: 4712173568 (43.88 GB)
DFS Used: 750137344 (715.39 MB)
DFS Used%: 1.57%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0
```
Exercise:

- The available size is around 43.9 GB; does it mean we can still store files of total size around 43 GB?

B.3.2.2 Checking Usage of a Directory in HDFS
To check how much the size of a directory in HDFS is, run the following command “hdfs dfs –du /path/to/directory”. Use additional parameter “-s” to get aggregated result, and additional parameter “-h” to display in human-readable size format (MB instead of Bytes). For example:

```
[cloudera@quickstart ~]$ hdfs dfs –du /user
2732  2732  /user/cloudera
0    0    /user/hdfs
0    0    /user/history
0    0    /user/hive
0    0    /user/hue
0    0    /user/jenkins
740441860 740441860  /user/oozie
0    0    /user/root
0    0    /user/spark
[cloudera@quickstart ~]$ hdfs dfs –du –h /user
2.7 K  2.7 K  /user/cloudera
0    0    /user/hdfs
0    0    /user/history
0    0    /user/hive
0    0    /user/hue
0    0    /user/jenkins
706.1 M 706.1 M  /user/oozie
0    0    /user/root
0    0    /user/spark
[cloudera@quickstart ~]$ hdfs dfs –du –s –h /user
706.1 M 706.1 M  /user
```
B.3.2.3 Setting Quota on a directory in HDFS
Blah.. blah.. blah..

B.4 Distributed Data Processing with MapReduce

MapReduce is the core of distributed processing in Hadoop. Most of framework/tools in Hadoop ecosystem will use MapReduce in their internal processing mechanisms.

This chapter will focus on developing a very basic MapReduce application that will read all files in an input directory in HDFS, count every occurrence of every word in all files, and write the result in an output directory in HDFS.

For this development, an input directory in HDFS will be created, and the same file from previous chapter will be used as an input file:

```
[cloudera@quickstart ~]$ hdfs dfs -mkdir mr-input
[cloudera@quickstart ~]$ hdfs dfs -put /usr/lib/hadoop-0.20-mapreduce/README.txt mr-input
```

B.4.1 MapReduce Program with Java

Create a directory in local home directory named “mapreduce”, so it will be in “/home/cloudera/mapreduce”. This will be our main directory for developing MapReduce program.

```
[cloudera@quickstart ~]$ mkdir mapreduce
[cloudera@quickstart ~]$ cd mapreduce/
```

We are going to create a MapReduce program in a java package named “id.ac.del.hadoop”, therefore we need to create respective directory:

```
[cloudera@quickstart mapreduce]$ mkdir -p id/ac/del/hadoop/
```

Create a .java file inside the above directory and name it WordCount.java.

The content of the file is as follow:

```java
package id.ac.del.hadoop;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
```
public class WordCount {

    public static class Map extends MapReduceBase implements 
        Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, 
                    OutputCollector<Text, IntWritable> output, 
                    Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }

    public static class Reduce extends MapReduceBase implements Reducer<Text, 
        IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, 
                            OutputCollector<Text, IntWritable> output, 
                            Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        JobClient.runJob(conf);
    }
}

Compile the MapReduce WordCount program above by running the following command:

    [cloudera@quickstart mapreduce]$ javac -classpath /usr/lib/hadoop/client-
Check whether the Java bytecode has been successfully generated from the compilation process above:

```bash
[cloudera@quickstart mapreduce]$ tree id
id
  └── ac
      └── del
          └── hadoop
              ├── WordCount.class
              ├── WordCount.java
              ├── WordCount$Map.class
              └── WordCount$Reduce.class
```

Package the *.class files into a JAR file so that it can be submitted as Hadoop job:

```bash
jar -cf wordcount.jar -C . id/ac/del/hadoop/*.class
```

Submit the JAR file as a Hadoop job:

```bash
[cloudera@quickstart mapreduce]$ hadoop jar wordcount.jar id.ac.del.hadoop.WordCount mr-input mr-output
```

**Note:** “mr-input” is the input directory in HDFS that already created previously, “mr-output” is non-existent directory that will be created by the MapReduce program to store the result of the process. The output directory should be deleted if the program will be run again.

Check the output result from “mr-output” directory:

```bash
[cloudera@quickstart mapreduce]$ hdfs dfs -ls mr-output
Found 2 items
-rw-r--r-- 1 cloudera cloudera 0 2016-10-08 15:20 mr-output/_SUCCESS
-rw-r--r-- 1 cloudera cloudera 1306 2016-10-08 15:20 mr-output/part-00000
[cloudera@quickstart mapreduce]$ hdfs dfs -cat mr-output/part-00000
(BIS), 1
(ECCN) 1
(TSU) 1
(see 1
5D002.C.1, 1
740.13) 1
<http://www.wassenaar.org/> 1
Administration 1
Apache 1
BEFORE 1
BIS 1
```
B.4.2 MapReduce Program with Hadoop Streaming and Python

A MapReduce program can also be programmed using executable script, such as GNU/Linux BASH script or python. Apache Hadoop provides Hadoop Streaming mechanism to enable executing scripts as a Map and Reduce process. This chapter will focus on developing MapReduce WordCount program using Python.

Create a Mapper program in Python called mapper.py as follows:

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

import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print '%s\t%s' % (word, 1)
```

Create a Reducer program in Python called reducer.py as follows:

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

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
    line = line.strip()

    word, count = line.split('\t', 1)

    try:
        count = int(count)
    except ValueError:
        continue
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print '%s\t%d' % (current_word, current_count)
        current_count = count
        current_word = word

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

if current_word == word:
    print '%s\t%s' % (current_word, current_count)

We will use the same input directory in HDFS with the input directory used in previous MapReduce program, but the output directory will be “mr-output2”. Submit the MapReduce program written in Python using Hadoop Streaming as follows:

[cloudera@quickstart mapreduce]$ hadoop jar /usr/lib/hadoop-0.20-mapreduce/contrib/streaming/hadoop-streaming-mr1.jar -file
/home/cloudera/mapreduce/mapper.py -mapper /home/cloudera/mapreduce/mapper.py
-file /home/cloudera/mapreduce/reducer.py -reducer
/home/cloudera/mapreduce/reducer.py -input mr-input -output mr-output2

Exercise:

- Modify the above programs (either with Java or Python) to count only words starts with a letter “a” either upper case or lower case.

B.5 Querying Data with Apache Hive (and Impala)

There are many alternatives when choosing SQL-on-Hadoop tools. Amongst them are Apache Hive and (Apache) Impala.

Hive is (arguably) the first SQL engine on Hadoop. It was designed to receive SQL queries from any source and process them directly against data stored in HDFS. This helped to bridge the gap between the existing skill sets of business users and the new Hadoop platform. It is possible to continue working with SQL queries, but actually querying big data on Hadoop on the back end. Apache Hive is a batch-oriented SQL-on-Hadoop tool; hence it’s most often used for running big and complex queries, including ETL and production data pipelines against massive data sets.

Impala was the first tool to attempt to deliver interactive-like response to SQL queries running over data on HDFS. Impala is an open source expansion of Hive SQL on top of the Hadoop system with particular emphasis on optimized data-local execution and concurrent, multi-user performance. Impala supports querying data within HDFS, HBase, and other widely used file formats such as Avro, Parquet, RCFile, etc.

This chapter will describe how to ingest data into and analyze data using Apache Hive and (Apache) Impala. We will be using HUE to perform the operations in this chapter.
B.5.1 Loading Data into Apache Hive

In this chapter, we will describe a mechanism to load data from local file system into Apache Hive. The first step will be to copy local file into HDFS. The dataset we are using in this chapter is driver_data.zip. Extract the file into C:\Users\USERNAME\Documents\driver_data. The datasets contain 3 (three) files: drivers.csv, timesheet.csv, truck_event_text_partition.csv.

B.5.1.1 Upload Datasets File into HDFS

Open the HUE web based application by running the web browser application and open the following address: http://192.168.100.130:8888/. After logging in with “cloudera” user name and “cloudera” password, click on the top right side of the page with the following icon to manage files in HDFS. The following is the “File Browser” page of the HUE:

![HUE File Browser](image)

Create a new directory by clicking New -> Directory in the right side of the page as follows:
Type “hive-datasets” and click “Create”. A new directory named “hive-datasets” will be created as “/user/cloudera/hive-datasets”. Click the newly created directory “hive-dataset”. We will now upload the three files in the local file system into “hive-datasets” directory in HDFS.

Click Upload -> Files in the right side of the page as follows:

Then, click “Select files” and select the three files and click “Open”, as follows:
The three files are now uploaded into the HDFS, as follows:

**Figure 18 Upload Selected Files**

![Upload to /user/cloudera/hive-datasets](image)

- drivers.csv 2.0kB
- timesheet.csv 25.6kB
- truck_event_text_partition.csv 2.2MB

**Figure 19 The three files uploaded**

### B.5.1.2 Create Empty Hive Table

After the three files are made available in the HDFS, we will now create a Hive table that will represent the data. The original files in CSV format, but by creating the Hive table, we will able to query the data inside CSV format as if it is a relational table using SQL language.
We can create either Managed table or External table. One crucial difference between the two is, if a Managed table is dropped, then the underlying data inside HDFS will also be deleted. Whereas, if an External table is dropped, only the table metadata is deleted, the underlying data inside HDFS will be kept intact.

This sub-chapter will describe how to create empty Hive table for “drivers.csv” data.

Open Hive Query Editor by clicking Query Editors -> Hive as follows:

Figure 20 Open Hive Query Editor

The Hive Query Editor page will be opened as follows:
Managed Table

External Table

To create Hive External table, type the following DDL on the SQL editor:

```sql
CREATE EXTERNAL TABLE drivers (  
  driverId INT,  
  name STRING,  
  ssn BIGINT,  
  location STRING,  
  certified STRING,  
  wage_plan STRING  
)
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ',',  
LOCATION '/user/user12/data/drivers'  
TBLPROPERTIES("skip.header.line.count"="1");
```

After typing the DDL code in the SQL editor, click “Execute” to run the DDL on Apache Hive, as follows:

![Hive DDL for drivers Table](image)
A query can also be saved in case it is needed to be executed again sometime in the future. Click “Save as…” to save the query:

![Choose a name](image1)

**Figure 23 Saving a Query**

After executing creating DDL code, data from drivers.csv need to be populated into the “drivers” table. To populate data from drivers.csv, type the following SQL code on the SQL editor:

```
LOAD DATA INPATH '/user/cloudera/hive-datasets/drivers.csv' OVERWRITE INTO TABLE drivers;
```

And click “Execute”.

After loading the data has been successfully executed, check the resulting table by clicking **Data Browsers -> Metastore Tables** as follows:

![Opening Metastore Tables](image2)

**Figure 24 Opening Metastore Tables**

A page similar with this one will be shown, displaying Databases and Table objects in the Apache Hive:
Figure 25 Metastore Tables view

Clicking the “drivers” table will open a page that can display list of columns of a table, sample data of a table, and properties of a table:
Exercise:

- Click Data Browser -> Metastore Tables. Use menu “Create a new table from a file” to create table for timesheet.csv and truck_event_text_partition.csv. Name the tables as “timesheet” and “truck_event_text_partition” respectively.
B.5.2 Querying Data in Apache Hive

After the three tables “drivers”, “timesheet”, and “truck_event_text_partition” have been successfully created, we can now query the data inside the tables. We will be using the Hive Query Editor to run SQL code against the tables that have just created.

B.5.2.1 Simple Query
Query 1 – Calculating the number of records of “drivers” table

```sql
SELECT COUNT(*) FROM drivers;
```

Exercise:

- Calculate the number of records of “timesheet” and “truck_event_text_partition” tables.

Query 2 – Aggregating sum of the hours and miles of each driver

```sql
SELECT driverid, SUM(hourslogged), SUM(mileslogged) FROM timesheet GROUP BY driverid;
```

B.5.2.2 Join Query
Query 3 – Aggregating sum of the hours and miles of each driver with additional column displaying the name of the driver.

```sql
SELECT d.driverId, d.name, t.total_hours, t.total_miles
FROM drivers d
JOIN
(SELECT driverid, SUM(hourslogged) total_hours, SUM(mileslogged) total_miles
FROM timesheet GROUP BY driverid ) t
ON (d.driverid = t.driverid);
```

B.5.3 Querying Data in (Apache) Impala

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. Therefore, tables created in Apache Hive will be also available to query using (Apache) Impala.

For tables created through Apache Hive to be available in (Apache) Impala, the metadata needs to be invalidated. From the HUE web based interface, click Query Editors -> Impala, it will open a page similar to below:
As we can see from Figure 23, no tables are currently available on the databases (while there are actually three tables have been created through Apache Hive). Therefore, we need to invalidate the metadata by executing the following code:

```
INVALIDATE METADATA;
```

After running the above code, we can see that the three tables created previously through Apache Hive are now available on (Apache) Impala:
Exercise:

- Try running Query 1, Query 2, and Query 3 that have been run in Apache Hive using (Apache) Impala. Notice the difference?

B.5.4 Using Hive to Store and Query XML Data

Apache Hive can be used to handle files with XML format. However, we need third-party library to start working with XML file in Apache Hive. We will be working with a simple XML file containing employee data as below:

```xml
<records>
  <record employee_id="100">
    <name>Michael</name>
    <datadiri>
      <gender>Male</gender>
      <age>30</age>
      <sin_number>547-968-091</sin_number>
    </datadiri>
    <datakantor>
      <workplace>Montreal, Toronto</workplace>
      <skill>DB:80</skill>
      <departement>Product</departement>
    </datakantor>
  </record>
</records>
```
Save the above file as employee.xml and put it into the HDFS as /user/cloudera/employee.xml.

Follow procedures below to enable Apache Hive to work with XML file format:

1) Download hivexmlserde JAR file from [http://search.maven.org/#search|ga|hlivexmlserde](http://search.maven.org/#search|ga|hlivexmlserde), save the file as hivexmlserde-1.0.5.3.jar.
2) Open Apache Hive Query Editor in HUE.
3) Click Settings tab, and click “Add” File Resources as follows:
The JAR file will be now located at /user/cloudera/udfs/hivexmlserde-1.0.5.3.jar

4) Register the JAR files with Apache Hive as follow:

   ADD JAR /user/cloudera/udfs/hivexmlserde-1.0.5.3.jar;

5) Create a table that will contain the employee.xml data as follows:

   ```
   CREATE TABLE xml_pegawai(
       employee_id INT,
       name STRING,
       datadiri map<string,string>,
       datakantor map<string,string>
   )
   ROW FORMAT SERDE 'com.ibm.spss.hive.serde2.xml.XmlSerDe'
   WITH SERDEPROPERTIES (  
   "column.xpath.employee_id"="/record/@employee_id",
   "column.xpath.name"="/record/name/text()",
   "column.xpath.datadiri"="/record/datadiri/*",
   "column.xpath.datakantor"="/record/datakantor/*"
   )
   STORED AS
   INPUTFORMAT 'com.ibm.spss.hive.serde2.xml.XmlInputFormat'
   OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat'
   TBLPROPERTIES (  
   "xmlinput.start"="<record employee_id",
   "xmlinput.end"="</record>"
   );
   ```

6) Load the /user/cloudera/employee.xml into xml_pegawai table as follow:

   ```
   LOAD DATA INPATH '/user/cloudera/employee.xml' INTO TABLE xml_pegawai;
   ```

7) Verify if the table has been successfully created and populated:
B.6 Distributed Data Processing with Apache Spark

Apache Spark is a fast, in-memory data processing engine with elegant and expressive development APIs in Scala, Java, Python, and R. Spark on Apache Hadoop YARN enables deep integration with Hadoop and other YARN enabled workloads in the enterprise.

At the core of Spark is the notion of a Resilient Distributed Dataset (RDD), which is an immutable collection of objects that is partitioned and distributed across multiple physical nodes of a YARN cluster and that can be operated in parallel. Typically, RDDs are instantiated by loading data from a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat on a YARN cluster.

Once an RDD is instantiated, a series of operations can be applied. All operations fall into one of two types: transformations or actions. Transformation operations, as the name suggests, create new datasets from an existing RDD and build out the processing Directed Acyclic Graph (DAG) that can then be applied on the partitioned dataset across the YARN cluster. An Action operation, on the other hand, executes DAG and returns a value.

Spark shell is an interactive shell for learning about Apache Spark, ad-hoc queries and developing Spark applications. It is a very convenient tool to explore the many things available in Spark and one of the many reasons why Spark is so helpful even for very simple tasks. There are variants of Spark for different languages: spark-shell for Scala and pyspark for Python. In this exercise, we will use spark-shell.
Start the Spark shell by typing “spark-shell” in the terminal. Depends on the setting of the logger, it will display some startup messages. Wait until it completes and show “scala>” prompt.

We will first try to execute a very simple operation:

```
scala> 2 + 3
res0: Int = 5
```

### B.6.1 Creating RDD from a File and Writing RDD to HDFS

Load an HDFS file /user/cloudera/mr-input/README.txt as an RDD:

```
scala> val words = sc.textFile("/user/cloudera/mr-input/README.txt")
```

Count number of lines in README.txt file:

```
scala> words.count
```

Explode every line of text as a collection of words:

```
scala> val wordsFlatMap = words.flatMap(_.split("\W+"))
```

Assign 1 for every exploded word:

```
scala> val wordsMap = wordsFlatMap.map( w => (w,1))
```

Use operation “reduce by key” to sum values which has the same keys:

```
scala> val wordCount = wordsMap.reduceByKey( (a,b) => (a+b))
```

Sort the result:

```
scala> val wordCountSorted = wordCount.sortByKey(true)
```

Display the result in the terminal:

```
scala> wordCountSorted.collect.foreach(println)
```

Store the result in HDFS:

```
scala> wordCountSorted.saveAsTextFile("/user/cloudera/README-WC")
```

Does the above code looks familiar? Yes, it is a WordCount program implemented in Apache Spark using Scala language. Which one you prefer, Java based MapReduce program or Scala based Spark program?

Better yet, the above code can be written in a single line as follows:

```
scala> sc.textFile("/user/cloudera/mr-input/README.txt").flatMap(_.split("\W+")) .map( w => (w,1)) .reduceByKey( (a,b) =>(a+b)) .sortByKey(true).collect.foreach(println)
```
B.6.2 Performing Operations

Below are common operations against RDD in Apache Spark. Try to execute each and every code below to understand how operations work.

```scala
scala> val pairRDD = sc.parallelize(List( "cat",2), ("cat", 5),
("mouse", 4), ("cat", 12), ("dog", 12), ("mouse", 2)), 2)
pairRDD.aggregateByKey(0)(math.max(_, _), _ + _).collect

scala> val x = sc.parallelize(List(1,2,3,4,5))
scala> val y = sc.parallelize(List(6,7,8,9,10))
scala> x.cartesian(y).collect

scala> val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
scala> c.count

cscala> val c = sc.parallelize(List((3, "Gnu"), (3, "Yak"), (5, "Mouse"),
(3, "Dog")), 2)
scala> c.countByKey

scala> val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1,1))
scala> b.countByKey

scala> val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu",
"Rat"), 2)
scala> c.distinct.collect

scala> val a = sc.parallelize(1 to 10, 3)
scala> val b = a.filter(_ % 2 == 0)
scala> b.collect

scala> val randRDD = sc.parallelize(List( (2,"cat"), (6, "mouse"),(7,
"cup"), (3, "book"), (4, "tv"), (1, "screen"), (5, "heater")), 3)
scala> val sortedRDD = randRDD.sortByKey()
scala> sortedRDD.filterByRange(1, 3).collect

scala> val a = sc.parallelize(1 to 10, 5)
scala> a.flatMap(1 to _).collect

scala> val a = sc.parallelize(List("dog", "tiger", "lion", "cat",
"panther", "eagle"), 2)
scala> val b = a.map(x => (x.length, x))
scala> b.flatMapValues("x" + _ + "x").collect

scala> val c = sc.parallelize(List("cat", "dog", "tiger", "lion", "gnu",
"crocodile", "ant", "whale", "dolphin", "spider"), 3)
scala> c.foreach(x => println(x + "s are yummy"))

scala> val b = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9), 3)
scala> b.foreachPartition(x => println(x.reduce(_ + _)))

scala> val pairRDD1 = sc.parallelize(List( ("cat",2), ("cat", 5),
("book", 4),("cat", 12)))
scala> val pairRDD2 = sc.parallelize(List( ("cat",2), ("cup", 5),
("mouse", 4), ("cat", 12)))
scala> pairRDD1.fullOuterJoin(pairRDD2).collect
```
scala> val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)
scala> val b = a.keyBy(_.length)
scala> b.groupByKey().collect

scala> val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
scala> val b = a.map(_.length)
scala> val c = a.zip(b)
scala> c.collect

scala> val y = sc.parallelize(10 to 30)
scala> y.max

scala> val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0, 8.9, 5.5), 3)
scala> a.mean

scala> val y = sc.parallelize(10 to 30)
scala> y.min

scala> val a = sc.parallelize(1 to 100, 3)
scala> a.reduce(_ + _)

scala> val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
scala> val b = a.map(x => (x.length, x))
scala> b.reduceByKey(_ + _).collect

scala> val y = sc.parallelize(Array(5, 7, 1, 3, 2, 1))
scala> y.sortBy(c => c, true).collect

Exercise:

- Load “drivers.csv” from the previous exercise as an RDD. Use Apache Spark operations to calculate the number of drivers whose name starts with letter “G”.

**B.7 Manipulating Structured Data Using Apache Spark**

Apache Spark introduces a programming module for structured data processing called Spark SQL. It provides a programming abstraction called DataFrame and can act as distributed SQL query engine. In this exercise, we will learn how to use DataFrame API with various data sources, such as JSON file and Apache Hive table.

**B.7.1 DataFrame Operation with JSON File**

Create a simple JSON file named people.json with contents as follows:

```json
{
  "name": "Michael"

  "name": "Andy", "age": 30

  "name": "Justin", "age": 19
}  ```
Run Spark shell, and execute the following commands:

```
scala> val df = sqlContext.jsonFile("file:///home/cloudera/people.json")
scala> df.count
scala> df.show
scala> df.printSchema
scala> df.select("name").show()
scala> df.select("name", "age").show()
scala> df.select("*").show()
scala> df.select(df("name"), df("age") + 1).show()
scala> df.filter(df("age") > 25).show()
scala> df.filter(df("name").startsWith("M")).show()
scala> df.groupBy("age").count().show()
```

**Exercise:**

Create a new JSON file named people2.json with contents as follows:

```
{"name":"Michael", "address":{"city":"Columbus","state":"Ohio"}}
{"name":"Andy", "age":30, "address":{"city":null, "state":"California"}}
{"name":"Justin", "age":19, "address":{"city":"Atlanta", "state":"Georgia"}}
```

1. Display only “name” and “state” from the above data.
2. Display “name” and “city” from the above data where the city is not null.

**B.7.2 Specifying a Schema Programmatically**

Apache Spark provide “registerTempTable”( tableName ) method for a DataFrame, because in addition to being able to use the Spark-provided methods of a DataFrame, we can also issue SQL queries via the sqlContext.sql( sqlQuery ) method, that use that DataFrame as an SQL table.

Use the following code to register a DataFrame as a temporary table and use SQL queries on that table:

```
scala> df.registerTempTable("people")
scala> sqlContext.sql("select name, age from people").collect.foreach(println)
```

**Exercise:**

- Use the methods in B.7.2 to answer exercise on B.7.1.
B.7.3 DataFrame Operation with Hive Table

Hive comes bundled with the Spark library as HiveContext, which inherits from SQLContext. Using HiveContext, we can create and find tables in the HiveMetaStore and write queries on it using HiveQL. Users who do not have an existing Hive deployment can still create a HiveContext. When not configured by the hive-site.xml, the context automatically creates a metastore called metastore_db and a folder called warehouse in the current directory.

Run Spark shell, and execute the following commands:

```scala
scala> val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
scala> hiveContext.setConf("hive.metastore.uris", "thrift://127.0.0.1:9083");
scala> hiveContext.sql("SELECT * FROM drivers").collect().foreach(println)
```

Exercise:

- Perform queries specified in B.5.2 using Apache Spark SQL.

B.8 Introduction to Machine Learning with Apache Spark

MLlib is Spark’s machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. It consists of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as lower-level optimization primitives and higher-level pipeline APIs.

It divides into two packages:

- spark.mllib contains the original API built on top of RDDs.
- spark.ml provides higher-level API built on top of DataFrames for constructing ML pipelines.

Available algorithms:

1. Classification and regression
   - linear models (SVMs, logistic regression, linear regression)
   - naive Bayes
   - decision trees
   - ensembles of trees (Random Forests and Gradient-Boosted Trees)
   - isotonic regression

2. Collaborative filtering
   - alternating least squares (ALS)

3. Clustering
   - k-means
4. Dimensionality reduction

- singular value decomposition (SVD)
- principal component analysis (PCA)

In this exercise, we will be using one of a simple available algorithm, Linear Regression. Linear regression is the approach to model the value of a response variable \( y \), based on one or more predictor variables or feature \( x \). The example we are going to use is house prices prediction based on historical data as follows:

<table>
<thead>
<tr>
<th>House size (sq ft)</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2100</td>
<td>$1,620,000</td>
</tr>
<tr>
<td>2300</td>
<td>$1,690,000</td>
</tr>
<tr>
<td>2046</td>
<td>$1,400,000</td>
</tr>
<tr>
<td>4314</td>
<td>$2,000,000</td>
</tr>
<tr>
<td>1244</td>
<td>$1,060,000</td>
</tr>
<tr>
<td>4608</td>
<td>$3,830,000</td>
</tr>
<tr>
<td>2173</td>
<td>$1,230,000</td>
</tr>
<tr>
<td>2750</td>
<td>$2,400,000</td>
</tr>
<tr>
<td>4010</td>
<td>$3,380,000</td>
</tr>
<tr>
<td>1959</td>
<td>$1,480,000</td>
</tr>
</tbody>
</table>

*Figure 32 Historical House Prices*

Based on the above data, we will predict the price of a house, if its size is 2500 sq ft, as follows:

```scala
scala> import org.apache.spark.mllib.linalg.Vectors
scala> import org.apache.spark.mllib.regression.LabeledPoint
scala> import org.apache.spark.mllib.regression.LinearRegressionWithSGD

scala> val points = Array(
LabeledPoint(1620000,Vectors.dense(2100)),
LabeledPoint(1690000,Vectors.dense(2300)),
LabeledPoint(1400000,Vectors.dense(2046)),
LabeledPoint(2000000,Vectors.dense(4314)),
LabeledPoint(1060000,Vectors.dense(1244)),
LabeledPoint(3830000,Vectors.dense(4608)),
LabeledPoint(1230000,Vectors.dense(2173)),
LabeledPoint(2400000,Vectors.dense(2750)),
LabeledPoint(3380000,Vectors.dense(4010)),
LabeledPoint(1480000,Vectors.dense(1959))
)
```
scala> val pricesRDD = sc.parallelize(points)

scala> val model = LinearRegressionWithSGD.train(pricesRDD,100,0.0000006,1.0,Vectors.zeros(1))

scala> val prediction = model.predict(Vectors.dense(2500))
prediction: Double = 1814061.9445510695

From the above result, we can see that for a house of size 2500 square feet, the predicted price is $1,814,061.94.