THE UNIVERSITY OF SYDNEY

School of Information Technologies
Dr. Ying Zhou

COMP5349: Cloud Computing **Sem. 1/2018**

Week 4: MapReduce Tutorial

29.03.2018

In this lab, you will practice basic MapReduce Programming on locally installed Hadoop MapReduce. The lab starts by setting up the environment in the RedHat Linux workstation. Hadoop has grown into a large ecosystem since its launch in 2006. The core of Hadoop includes a distributed file system `HDFS` and a parallel programming engine `MapReduce`. Since `Hadoop 2`, YARN is included as the default resource scheduler.

MapReduce can run in three modes: standalone, pseudo-distributed and full-distributed. Standalone mode needs only the `MapReduce` component. The pseudo and full distributed modes require HDFS and YARN.

In this lab, you will use both the standalone and pseudo cluster mode. You will use some basic HDFS and YARN commands in the pseudo cluster mode. The architecture details of HDFS and YARN will be covered in week 6 lecture.

Question 1: Prepare the Environment

We have installed the `Hadoop MapReduce 2.9` in Redhat workstation. The installation location is `/usr/local/hadoop-2.9.0`. Running Hadoop in standalone mode requires us to setup two environment variables: `HADOOP_HOME` and `JAVA_HOME`. Additional configuration is required to run Hadoop in *pseudo cluster* mode. By default Hadoop put all configuration files under `$HADOOP_HOME/etc/hadoop`. Regular users do not have write permission of that directory. One way to get around this restriction is to put all configuration files in a directory under your home directory and set the environment variable `HADOOP_CONF_DIR` accordingly.

A login script template as well as all required configuration files are pushed in a new repository `lab_commons` in Github `https://github.sydney.edu.au/COMP5349-Cloud-Computing/lab_commons`.

Create a directory call `comp5349` in your home directory and clone this repository locally. You will find it contains three sub directories: `profile`, `hadoop-conf` and `data`.

The login script template `LOGIN.TEMPLATE` is stored under `profile`. Copy this file under your home directory and rename it as `.profile`. The template sets up all necessary environment variables, including future ones for `Spark`. It also updates the `$PATH` variable accordingly.

The login script will be executed each time a new terminal window is opened. If you want the login script to take immediate effect in the current terminal window, run `source .profile`

The login script contains a case-switch statement to setup environment variables based on HOSTNAME patterns. All our lab workstation's hostname starts with letter 'w', the following part of the script is related with workstation environment in this week's lab:

```
export JAVA_HOME=/etc/alternatives/java_sdk
export HADOOP_HOME=/usr/local/hadoop-2.9.0
export HADOOP_CONF_DIR=~/comp5349/hadoop-conf
export HADOOP_LOG_DIR=~/comp5349/hadoop-logs
export YARN_LOG_DIR=~/comp5349/hadoop-logs
export HADOOP_JHS_LOGGER=~/comp5349/hadoop-logs
export HADOOP_MAPRED_LOG_DIR=~/comp5349/hadoop-logs

...
;;
```

The script refers to several directories not yet exist. In particular, it specifies:

- Hadoop should look for configuration files in a directory `comp5349/hadoop-conf` under your home directory.
- Both Hadoop and YARN and history servers should write log files in a directory `comp5349/hadoop-logs` under your home directory.

You can change the login script to use other directories for configuration and logging. In the following we assume you use the given login script. Do the following to prepare required directories:

把hdfs-site.xml的unikey改成自己的

- copy `hadoop-conf` directory from `lab_commons` to `comp5349`
- create a directory `hadoop-logs` under `comp5349`

Question 2: Running Sample Example in Standalone Mode

In standalone mode, Hadoop runs as a single Java process. It reads from and writes to local file system. It is useful for debugging.

Hadoop relies on configuration files to decide its running mode. The default configuration is for standalone mode, while the configuration you obtain from `lab-commons` is for pseudo cluster mode.

To revert back to standalone mode, unset the HADOOP_CONF_DIR environment variable by

```
unset HADOOP_CONF_DIR
```

This only has temporary effect, allowing you to use the standalone mode in the particular terminal window.

Hadoop is written in Java language. A user defined Java hadoop program is usually packed as a jar file and submitted for execution using the `hadoop` script. The submission format is:

`hadoop jar PathToJarFile [mainClass] [ProgramArgLists]`

All Hadoop distribution comes with an example jar with a few sample MapReduce program. These include `wordcount`, `grep` and so on.

In this exercise you will run a sample word count program using the data file `place.txt` as input. You can find the data file from `lab_commons/data`

Below is the command for running the word count program. It is assumed that you have cloned `lab_commons` under `~/comp5349`. If not, you should change the input path accordingly. The last argument specifies the output directory. Hadoop always creates a new output directory each time your run a program. It does not overwrite any existing directory. If you specify an existing directory as the output directory, the program will exit with an error. When you need to run the same program multiple times, make sure you either remove the output directory before submitting the job; or change the name of the output directory in a new run.

```
hadoop jar \
    $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.9.0.jar \
    wordcount \
    ~/comp5349/lab_commons/data/place.txt \
    countout
```

After submitting the job successfully, progress information will be printed out on the console as the job starts to execute.

After the program finishes successfully, you can check the output directory `countout`. It should contain a file called `part-r-00000`. This is the output of the reduce task. The job contains only one reduce task, hence one output file. The output of reduce tasks are named using the pattern `part-r-xxxxx`, where the number is the id of the reduce task, starting from 0. For instance, if your job has two reduce tasks, there will be two result files: `part-r-00000` and `part-r-00001`. All Hadoop output files are named in this style. The letter indicate whether it is produced from a mapper or a reducer. Output produced by map task will be named as `part-m-xxxxx`. The number indicates the id of the mapper/reducer that produces the output.

The output file contains all word in the input file as well as their occurrence count. Each word occupies a line.

Question 3: Running Sample Example in Pseudo-Distributed Mode

In pseudo-distributed mode, a pseudo cluster is running in a single machine. A cluster consists of several components, each requires a few long running services/daemons. Those daemons will be running in the same machine.

There are certain benefits of running pseudo-distributed mode. Nearly all modern machines are configured with a multi-core processor, meaning it is possible to enjoy certain level of parallelism in a single machine. A pseudo cluster also has a rich set of services for exploring running statistics. All following exercises are based on the pseudo distributed mode.

a) **Setup passphraseless ssh**
Hadoop provides several scripts to start or stop all services in a cluster. The script needs to SSH to all workstations in a cluster using key based authentication to start corresponding services there.
A pseudo cluster runs many Hadoop related services on localhost. The start and stop script need to be able to ssh to localhost without password, ie. by using key based authentication as well.
Check that you can ssh to the localhost without a password:

```
ssh localhost
```

If you cannot ssh to localhost without a password, execute the following commands to generate a key to identify yourself:

```
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 0600 ~/.ssh/authorized_keys
```

Check again to ensure you can ssh to localhost without supplying password.

b) **Start HDFS**
A pseudo cluster requires HDFS. HDFS is a distributed file system running on top of a host's native file system. It runs on a designated location in the host's native file system. You do not have write permission on the location specified by the default configuration. We assume that your HDFS should store data in `/comp5349/hadoop-data`. Run the following commands to prepare the directories:

```
mkdir -p ~/comp5349/hadoop-data/dn
mkdir -p ~/comp5349/hadoop-data/nn
```

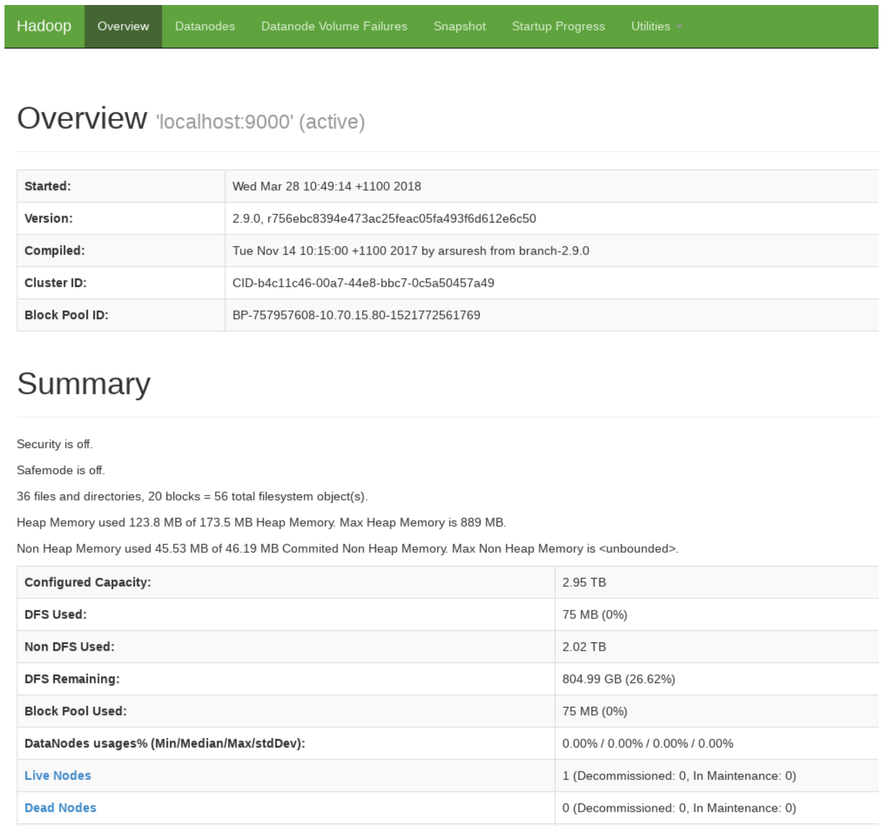
You also need to update the configuration file `hdfs-site.xml` to reflect the change. Open this file and replace the hard coded unikey (e.g. `yzho8449`) with your own one.
Format the file system with:

```
hdfs namenode -format
```

Then start all HDFS services with

```
$HADOOP_HOME/sbin/start-dfs.sh
```

Check that HDFS is successfully started by visiting the HDFS Web UI `http://localhost:50070/` in your browser, you should be able to see something similar to Figure 1



Overview 'localhost:9000' (active)

Started:	Wed Mar 28 10:49:14 +1100 2018
Version:	2.9.0, r756ebc8394e473ac25feac05fa493f6d612e6c50
Compiled:	Tue Nov 14 10:15:00 +1100 2017 by arsuresh from branch-2.9.0
Cluster ID:	CID-b4c11c46-00a7-44e8-bbc7-0c5a50457a49
Block Pool ID:	BP-757957608-10.70.15.80-1521772561769

Summary

Security is off.

Safemode is off.

36 files and directories, 20 blocks = 56 total filesystem object(s).

Heap Memory used 123.8 MB of 173.5 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 45.53 MB of 46.19 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	2.95 TB
DFS Used:	75 MB (0%)
Non DFS Used:	2.02 TB
DFS Remaining:	804.99 GB (26.62%)
Block Pool Used:	75 MB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)

Figure 1: HDFS Web UI

If you cannot see the webUI, go to `/comp5349/hadoop-logs` to check the log file of namenode. The log file contains detailed errors while starting this service.

A few directories need to be prepared for executing Hadoop jobs:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/<YourUniKey>
```

HDFS has a list of commands for normal file operations such as creating directory and copying files. All HDFS file system commands have a prefix `hdfs dfs`. The actual commands have similar names as the linux file system commands. For instance, `mkdir` is the command to create directory in HDFS.

HDFS can be stopped with

```
$HADOOP_HOME/sbin/stop-dfs.sh
```

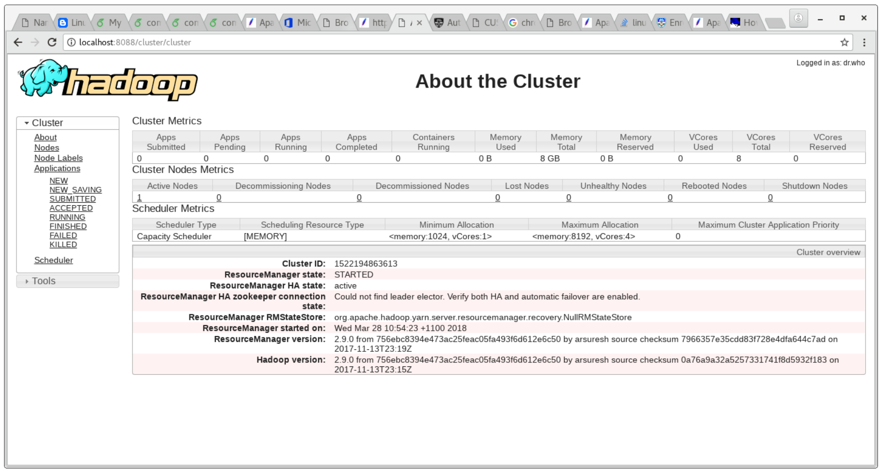
Remember to stop it before leaving the lab.

c) **Start YARN**

YARN is the preferred resource scheduler for MapReduce. We have prepared the configuration files to run YARN on the pseudo cluster. To start YARN:

```
$HADOOP_HOME/sbin/start-yarn.sh
```

To check that YARN has been started successfully, visit the YARN WebUI at `http://localhost:8088/` in your browser. You will see an output similar to Figure 2:



About the Cluster

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved
0	0	0	0	0	0 B	8 GB	0 B	0	8	0

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes	Shutdown Nodes
1	0	0	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation	Maximum Cluster Application Priority
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>	<memory:8192, vCores:4>	0

Cluster overview

Cluster ID:	1522194863613
ResourceManager state:	STARTED
ResourceManager HA state:	active
ResourceManager HA zookeeper connection state:	Could not find leader elector. Verify both HA and automatic failover are enabled.
ResourceManager RMStateStore:	org.apache.hadoop.yarn.server.resourcemanager.recovery.NullRMStateStore
ResourceManager started on:	Wed Mar 28 10:54:23 +1100 2018
ResourceManager version:	2.9.0 from 756ebc8394e473ac25feac05fa493f6d612e6c50 by arsuresh source checksum 7966357e35cdd83f728e4dfa644c7ad on 2017-11-13T23:19Z
Hadoop version:	2.9.0 from 756ebc8394e473ac25feac05fa493f6d612e6c50 by arsuresh source checksum 0a76a9a32a5257331741f8d5932f183 on 2017-11-13T23:15Z

Figure 2: YARN Web UI

Create a directory on HDFS to save application logs

```
hdfs dfs -mkdir -p /var/log/hadoop-yarn/apps
```

Start a job history server so we have a web UI to access job logs:

```
$HADOOP_HOME/sbin/mr-jobhistory-daemon.sh  start historyserver
```

To stop jobhistory, run the following

```
$HADOOP_HOME/sbin/mr-jobhistory-daemon.sh  stop historyserver
```

To stop YARN, run the following:

```
$HADOOP_HOME/sbin/stop-yarn.sh
```

Again make sure you stop all services before leaving the lab.

d) **Run sample program**

The default file system for Hadoop pseudo cluster operation is HDFS. In this exercise, HDFS will be used for program's input and output.

Run the following command to put the `place.txt` file in HDFS, under your home directory.

```
hdfs dfs -put ~/comp5349/lab_commons/data/place.txt \
place.txt
```

0 datanode
删除log data
formate

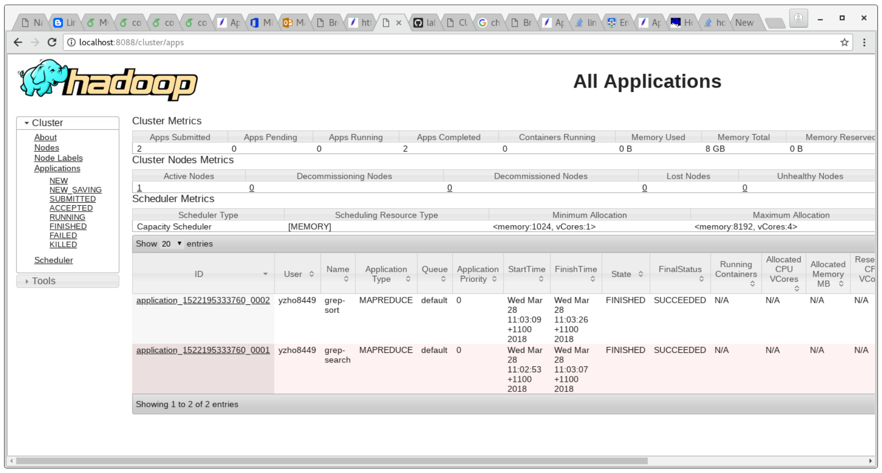
Make sure you are in a terminal with `HADOOP_CONF_DIR` set to your own configuration directory.

The following command runs the `grep` example on `place.txt` to look for all records with the word “Australia” in it

```
hadoop jar \
    $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.9.0.jar \
    grep \
    place.txt \
    placeout \
    "/Australia[\d\w\s\+/]+"
```

The `grep` program requires three parameters: input location, output location and a regular expression to search for. In the above command `place.txt` specifies the input location; `placeout` specify the output directory; the last one is the regular expression.

Similar to previous execution in standalone mode, progress information is printed out on the console as the job starts to execute. Application progress and summary information can be viewed from YARN WebUI as well.Figure 3 shows the application information while it is still running. You will notice that the `grep` example has started two jobs: `grep-search` and `grep-sort`


localhost:8088/cluster/apps

All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved
2	0	0	2	0	0 B	8 GB	0 B

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes
1	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>	<memory:8192, vCores:4>

Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Rese CF VCo
application_1522195333760_0002	yzho8449	grep-sort	MAPREDUCE	default	0	Wed Mar 28 11:03:09 +1100 2018	Wed Mar 28 11:03:26 +1100 2018	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A
application_1522195333760_0001	yzho8449	grep-search	MAPREDUCE	default	0	Wed Mar 28 11:02:53 +1100 2018	Wed Mar 28 11:03:07 +1100 2018	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A

Showing 1 to 2 of 2 entries

Figure 3: YARN Application View

The output of this application is written on HDFS, go to HDFS WebUI `http://localhost:50070/` to inspect the `placeout` directory under your home directory. You should see a file named `part-r-00000`.

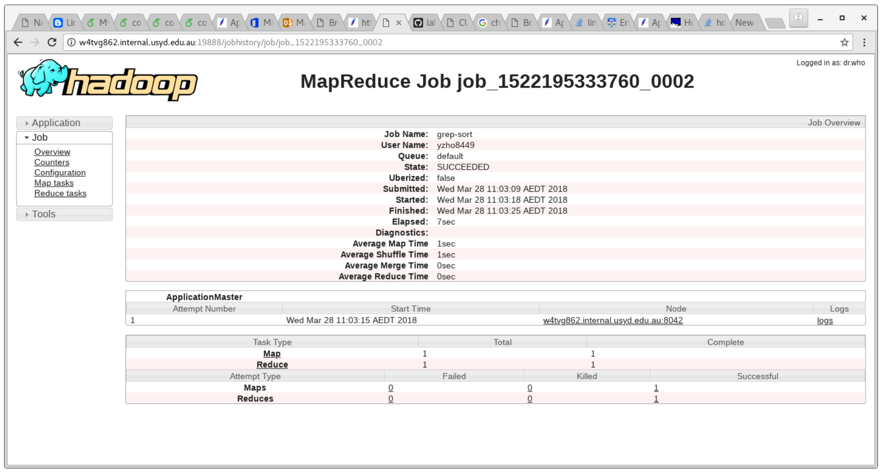
There are two ways to inspect the content of HDFS file. You may inspect it using commands like `cat`, or `tail`:

```
hdfs dfs -cat placeout/part-r-00000
```

You can download the file to host file system and view it using either command line or graphic tools. The following command download the output file `part-r-00000` and save it as `placeout.txt` in your current directory.

```
hdfs dfs -get  placeout/part-r-00000 placeout.txt
```

Details of running statistics of finished applications can be obtained from history sever. Figure 4 is an example of the history server view of the `grep-sort`application.



w4tvg862.internal.usyd.edu.au:19888/jobhistory/job/job_1522195333760_0002

Logged in as: dr.who

MapReduce Job job_1522195333760_0002

Job Overview

Job Name:	grep-sort
User Name:	yzho8449
Queue:	default
State:	SUCCEEDED
Uberized:	false
Submitted:	Wed Mar 28 11:03:09 AEDT 2018
Started:	Wed Mar 28 11:03:18 AEDT 2018
Finished:	Wed Mar 28 11:03:25 AEDT 2018
Elapsed:	7sec
Diagnostics:	
Average Map Time	1sec
Average Shuffle Time	1sec
Average Merge Time	0sec
Average Reduce Time	0sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Wed Mar 28 11:03:15 AEDT 2018	w4tvg862.internal.usyd.edu.au:8042	logs

Task Type	Total	Complete
Map	1	1
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	1
Reduces	0	0	1

Figure 4: MapReduce Job Hisotry View

Question 4: A Customized Sample Program

In this exercise, you will see the full source code of a MapReduce program in `Java` and in `Python`. The input of the program is a csv file. Each line represents a photo record of the following format:

```
photo-id owner tags date-taken place-id accuracy
```

The `tags` are represented as a list of words separated by white spaces. Each word is called a tag. You can find a sample input file called `partial.txt` from `lab_commons`. Copy this file to HDFS so that we can run the program in pseudo cluster.

```
hdfs dfs -put ~/comp5349/lab_commons/data/partial.txt \
partial.txt
```

The MapReduce application will build a tag-owner index for each tag appearing in the data set. The index key is the tag itself, the entry records the number of times an owner has used this tag in the data set. For example, if a data set contains only these three records:

```
509657344 7556490@N05 protest 2007-02-21 02:20:03 xbxI9VGYA5oZH8tLJA 14
520345662 8902342@N05  winter protest 2009-02-21 02:20:03 cvx093MoY15ow 14
520345799 8902342@N05  winter protest 2009-02-21 02:35:03 cvx093MoY15ow 14
```

The program would generate an index with two keys: "protest" and "winter".

The index of "protest" would look like

```
protest 7556490@N05=1, 8902342@N05 =2,
```

The index of "winter" would look like

```
winter 8902342@N05 =2,
```

a) **Java source Code**

The java source code can be found from `Java_Resources` repository under directory `week4`. An ant build file has been provided. To build an executable `jar`, run `ant` in the directory where `build.xml` file resides.

You can find the Java source code in directory `src`. It contains six classes representing two implementation versions. Each version contains a Driver, a Mapper and A Reducer.

The naive implementation contains three classes: `TagDriver`, `TagMapper` and `TagReducer`. The map and the reduce function are implemented in the `TagMaper` and `TagReducer` class respectively. The driver class is the entry point of the program and it specifies important properties of the program. These include the input and output location; how should input be split and feed to each map function; how should output be saved.

The `map` function is designed to processes a line of the input file. The key is the byte offset of the line, which we do not need to use in the future. The value is the actual line content as a string. It output a list of `(tag, owner)` pairs.

The `reduce` function is designed to process all information related with a particular tag. The input key is the tag, the input value is a list of owners that has used this tag. If an owner has used this tag three times, it will appear three times in the value list. The output key is still the tag, the output value is a summary of the owner data with respect to that tag. If `owner2` has used this tag three times, it will have record `owner2 = 3` in the output value.

Below is a summary of the map and reduce function highlighting the input/output key value pairs:

```
Map:
    (fileOffset, line) => {(tag, owner1), (tag,owner2),...}
Reduce:
    (tag,{owner1, owner1, owner2, ...}) => (tag,{owner1=x, owner2=y, ...})
```

The ‘smart’ version contains three classes: `TagSmartDriver`, `TagSmartMapper` and `TagSmartReducer`. This version applies a combiner function to do local aggregation after each map. The combiner function does exactly the same thing as the reduce function. To make it work, the input/output key value pairs have changed slightly to become:

```
Map:
    (fileOffset, line) => {(tag, owner1=1),(tag,owner2=1),...}
Combine:
    (tag,{owner1=a, owner2=b, ...}) => (tag,{owner1=x, owner2=y, ...})
Reduce:
    (tag,{owner1=a, owner2=b, ...}) => (tag,{owner1=x, owner2=y, ...})
```

Both versions take two parameters: an input path and an output directory.
The following command runs the naive version :

```
hadoop jar userTag.jar usertag.TagDriver partial.txt userTagOutNaive
```

To run the ‘smart’ version, just change the main class name to `usertag.TagSmartDriver`

```
hadoop jar userTag.jar usertag.TagSmartDriver partial.txt userTagOutSmart
```

The result would be exactly the same. You should see three files in the output directory because we have set to use three reducers. The running statistics, would be slightly different, in particular, the ‘smart’ version has less data transfer, e.g. Shuffle Bytes, between the map and reduce phase.

b) **Python Source Code**

The Python version of the same program (credit to Andrian Yang) can be found from `Python_Resources` repository. It contains two versions as well, putting under different directories. Each version consists of two python scripts, implementing the “map” and “reduce” function respectively. The program logic is exactly the same as the Java Program. Python MapReduce program does not use a separate driver script to set program specific properties. These are instead specified as command line arguments. A script file, `tag_driver.sh` or `tag_driver_combiner.sh` is provided to show you how to run the python script using the streaming jar with all necessary arguments.
Please note that the input to Python’s reduce program is only sorted but not grouped. You need to implement your own program logic to identify the key change. Below are examples of grouped and not grouped reduce input, using the combiner version
Two records with grouped format, typical input of reduce function in Java code:

```
protest:{7556490@N05=1, 8902342@N05 =2}
winter:{8902342@N05 =2}
```

Sorted but not grouped format, typical input of reduce function in Python code:

```
protest:7556490@N05=1
protest:8902342@N05 =2
winter:8902342@N05 =2
```

Question 5: Write your own code

You have seen a MapReduce like program in week 1 homework exercise. You are asked to convert it to proper Hadoop MapReduce application. The week 1 exercise asks you to compute the average movie rating for those movies whose ID is in a given range in a given range. This involves passing command line arguments to the MapReduce program. In this exercise, you are asked to compute the average movie ratings for all movies in the data set.

References

- Hadoop: Setting up a Single Node Cluster. `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html`
- HDFS Commands Guide `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html`