



Developing Hadoop Applications

Spring 2015

This Lab Guide is protected under U.S. and international copyright laws, and is the exclusive property of MapR Technologies, Inc. © 2015, MapR Technologies, Inc. All rights reserved.



Get Started

Welcome to *Developing Hadoop Applications*

This course teaches developers, via lectures and hands-on lab exercises, how to write Hadoop Applications using MapReduce in Java. This course prepares you to successfully pass the *MapR Certified Hadoop Professional Developer*, provided that you complete all lab exercises. Covered are:

- Describe the MapReduce programming paradigm
- Use MRv1 and MRv2 job frameworks
- Implement a basic MapReduce application
- Use the MapReduce API
- Manage and test MapReduce jobs
- Tune MapReduce performance
- Launch and manage multiple jobs
- Work with different data in MapReduce
- Implement and manage streaming MapReduce jobs

Prerequisites

- Beginner-to-intermediate fluency with Java
- A laptop (Linux, PC or Mac) to connect to a Hadoop cluster via SSH and web browser
- Basic Linux end-user skills (e.g `ls`, `cd`, `vi`)

How to Use this Guide

Typographical Conventions

Courier	Things you type as shown
Courier	Things as seen on screen
<i>italics</i>	Things you replace with a value
Durations	Suggested time it takes to complete an exercise. Some are required, other are optional

Lab Exercise Timing

Each Lesson's Labs are listed with estimated time to completion.

Exercise	Required	Duration
Connect to the MapR cluster (sandbox or AWS)	Yes	30 min

Tools Needed to Complete Labs

Please download all associated course materials. You will find documents that offer instructions on setting up the lab environment you will need to complete the labs:

- MapR Sandbox for Hadoop– Download the sandbox and follow instructions to install the sandbox and get running. (f you use the MapR sandbox, you will not be able to do any of the labs that require you to view metrics – in order to view the metrics, use an Amazon Web Services node)
- Amazon Web Services Node – instructions on how you can purchase and set up your own lab environment in the cloud.
- doc.mapr.com – a repository of documentation on MapReduce

Connect to the MapR Cluster (Sandbox or AWS)

The purpose of this lab is for you to establish connectivity to your MapR cluster environment that you will use for the hands-on exercises in this course.

Logging into the Sandbox

This procedure assumes that you have downloaded and installed the MapR Sandbox as part of the class prerequisites. If not, please do that before continuing here. Make a note of the IP address assigned to the Sandbox, as you will need it to log in remotely.

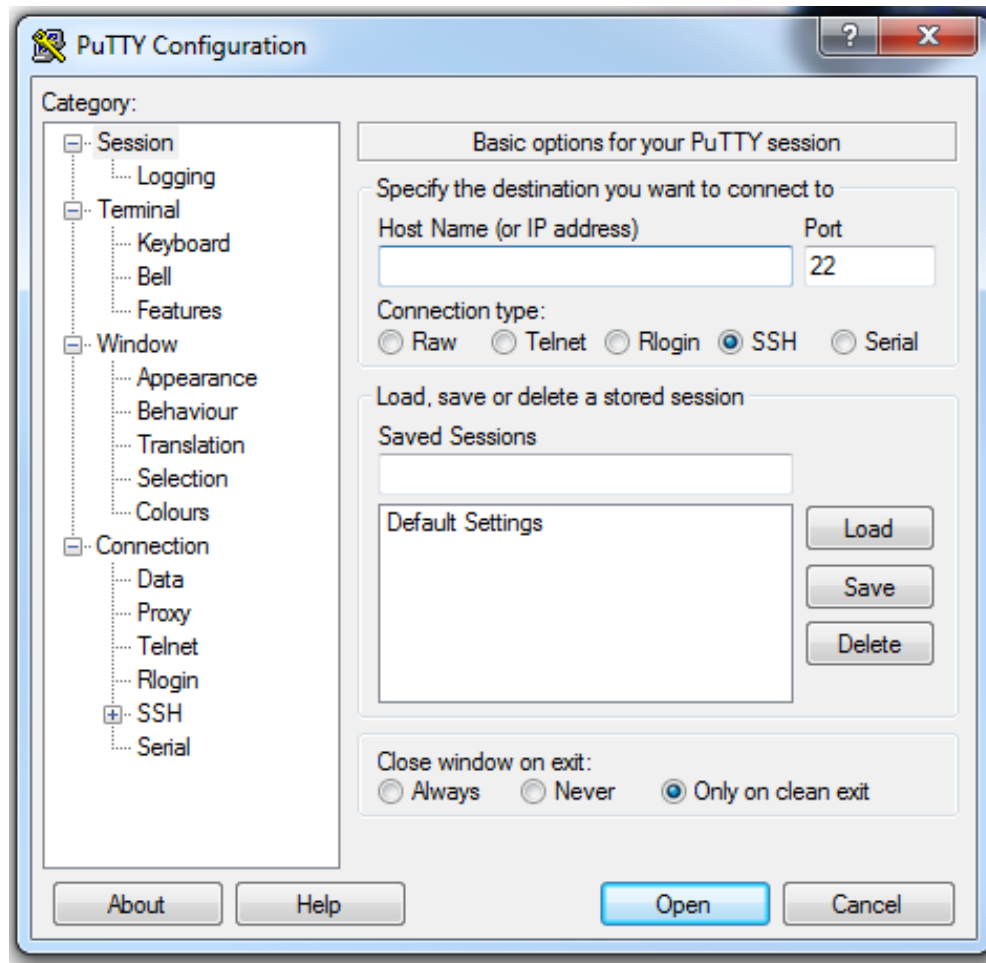
The Sandbox is preconfigured with several user accounts. You will be logging in as user01 to complete most of the lab exercises.

You will need to use a Secure Shell (SSH) client to log into the Sandbox. PuTTY, Cygwin, or a bash shell on a Unix-like OS with OpenSSH installed are all acceptable clients. If you do not have an SSH client, you can log in to the Sandbox using the virtual machine console mode. Connect to your Sandbox as user01.



OPTION 1: Log in using PuTTY

1. Start your PuTTY Client.



2. In the Host Name (or IP address) box, enter the host name or IP address for your Sandbox virtual machine.
3. Click the Open button to open the connection to your node.
4. When the terminal window appears, log into the terminal as user01.

OPTION 2: Log in using terminal SSH client

Start your terminal client. Use the `ssh` program to connect to the hostname or IP address for your Sandbox. Connect as user01 , password mapr

```
$ ssh user01@IP-address
```



OPTION 3: Log in using Virtual Machine Console

You can log in directly to the Sandbox console (in VMWare Player or Workstation, or in Virtual Box).

```
=== MapR Sandbox for Hadoop ===
```

```
Version: 1.0
```

```
MapR Sandbox for Hadoop Installation finished successfully.
```

```
Please go to http://192.168.30.129:8443 to begin your experience
```

```
Open a browser on your host machine
```

```
And enter the URL in the browser address field
```

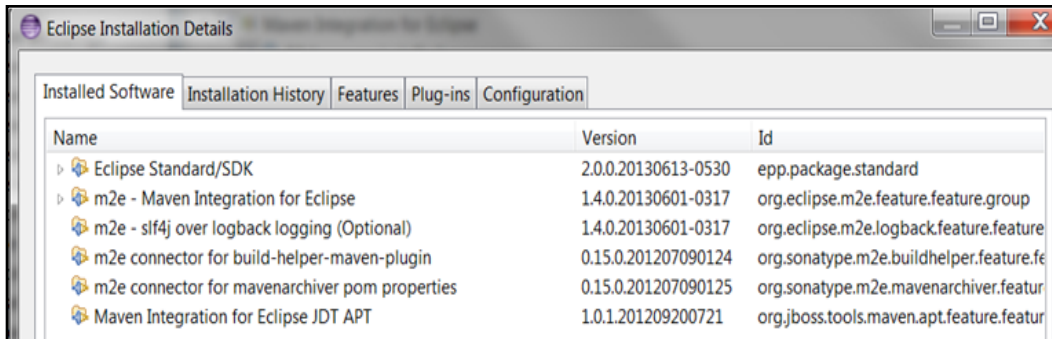
```
Log in to this virtual machine: Linux/Windows <Alt + F2>, Mac OS  
X <Option + F5>
```

Use the appropriate key combination for your host machine (Alt + F2 for Windows or Linux, Option + F5 for Mac OS X) to reach the login prompt.

```
CentOS release 6.5 (Final)
```

```
Kernel 2.6.32-431.el6.x86_64 on an x86_64
```

```
Maprdemo login: _
```



Lesson 1: Run MapReduce Jobs

In the lesson's lab exercises, you will run a few MapReduce jobs from the command line.

IMPORTANT NOTE: Some of the commands in this lab are too long to fit on a single line in this document. The backslash character in a command indicates that the command continues on the next line. You don't need to type the backslash character at all, but if you do, make sure the very next character following the backslash is a carriage return.

Exercise	Duration
1.1: Run wordcount - Explore running the classic "word count" MapReduce application that ships with Hadoop. You will run the "word count" on MRv2.	20 min

Exercise 1.1: Run wordcount

Run wordcount against a set of text files

1. Create a directory in your home directory as follows:

```
$ mkdir -p /user/user01/1.1/IN1
```

2. Create a set of text files in that directory as follows.

```
$ cp /etc/*.conf /user/user01/1.1/IN1 2>/dev/null
```

3. Determine how many files are in that directory.

```
$ ls /user/user01/1.1/IN1 | wc -l
```

4. Run the MRv2 version of the wordcount application against the directory.

```
$ hadoop jar /opt/mapr/hadoop/hadoop-\
2.5.1/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.5.1-mapr-\
1503.jar wordcount /user/user01/1.1/IN1 /user/user01/1.1/OUT1
```

5. Check the output of the wordcount application.

```
$ wc -l /user/user01/1.1/OUT1/part-r-00000
```

```
$ more /user/user01/1.1/OUT1/part-r-00000
```

Run wordcount against a binary File

1. Create a directory in your home directory as follows:

```
$ mkdir -p /user/user01/1.1/IN2
```
2. Create a binary file in that directory as follows.

```
$ cp /bin/cp /user/user01/1.1/IN2/mybinary
```
3. Verify the file is a binary.

```
$ file /user/user01/1.1/IN2/mybinary
```
4. See if there is any readable text in the binary.

```
$ strings /user/user01/1.1/IN2/mybinary | more
```
5. Run the MRv1 version of the wordcount application (using the MRv2 client) against the input file. This will show that binaries compiled for MRv1 will run in a MRv2 framework.

```
$ hadoop jar /opt/mapr/hadoop/hadoop-0.20.2/hadoop-0.20.2-dev-  
examples.jar wordcount /user/user01/1.1/IN2/mybinary \  
/user/user01/1.1/OUT2
```
6. Check the output of the wordcount application.

```
$ more /user/user01/1.1/OUT2/part-r-00000
```
7. Cross-reference the frequency of the “word” ATUH in the binary and in the wordcount output.

```
$ strings /user/user01/1.1/IN2/mybinary | grep -c ATUH  
$ egrep -ac ATUH /user/user01/1.1/OUT2/part-r-00000
```



Lesson 2: Run Yarn Job

In the lesson's lab exercises, you will run a YARN job from the command line and examine job information in the History Server Web UI.

Exercise	Duration
2.1: Run distributed shell - Explore running the sample distributed shell YARN application that ships with Hadoop.	10 min
2.2: Examine Job Results in History Server - Use the History Server Web UI to examine metrics for jobs you previously launched.	10 min

Exercise 1: Run DistributedShell

In this exercise, you will explore running a sample YARN job that ships with Hadoop.

Run DistributedShell With a Shell Command

1. Launch the YARN job using the yarn command.

```
$ yarn jar /opt/mapr/hadoop/hadoop-2.5.1/share/hadoop/yarn/hadoop-yarn-  
applications-distributedshell-2.5.1-mapr-1503.jar -shell_command \  
/bin/ls -shell_args /user/user01 -jar /opt/mapr/hadoop/hadoop-\  
2.5.1/share/hadoop/yarn/hadoop-yarn-applications-distributedshell-\  
2.5.1-mapr-1503.jar
```

2. When the job completes, scroll back through the output to determine your container ID for the shell (hint: look for the string "Submitted application application....").

```

15/04/14 12:32:48 INFO distributedshell.Client: Setting up app master command
15/04/14 12:32:48 INFO distributedshell.Client: Completed setting up app master
command {{JAVA_HOME}}/bin/java -Xmx10m org.apache.hadoop.yarn.applications.distr
ibutedshell.ApplicationMaster --container_memory 10 --container_vcores 1 --num_c
ontainers 1 --priority 0 1><LOG_DIR>/AppMaster.stdout 2><LOG_DIR>/AppMaster.stde
rr
15/04/14 12:32:48 INFO distributedshell.Client: Submitting application to ASM
15/04/14 12:32:49 INFO security.ExternalTokenManagerFactory: Initialized externa
l token manager class = com.mapr.hadoop.yarn.security.MapRTicketManager
15/04/14 12:32:49 INFO impl.YarnClientImpl: Submitted application application_14
29037364711_0003

```

Check Standard Output and Standard Error for Job

1. Change directory to the output directory for YARN jobs.
\$ **cd /opt/mapr/hadoop/hadoop-2.5.1/logs/userlogs**
2. List the contents of the directory.
\$ **ls**
3. Change directory to your application output directory.
\$ **cd application_timestamp_appid**
4. List the contents of the directory.
\$ **ls**
5. Change directory to the second container output directory.
\$ **cd container_timestamp_appid_01_000002**
6. Display the contents of the `stdout` file. You should see a listing of the `/user/user01` directory.
\$ **cat stdout**
7. Display the contents of the `stderr` file. It should be empty.
\$ **cat stderr**



Exercise 2: Examine Results in History Server Web UI

In this exercise, you will use the Web UI provided by the History Server to examine information for the job you previously launched.

1. Connect to the History Server in your Web browser.
<http://node-IP:8088>
2. Scroll through the applications to find your application ID.
3. Click the link associated with your YARN job.
4. How long did the job take?



Lesson 3: Modify a MapReduce Program

Lab Overview

The labs for this lesson cover how to make some modifications to an existing MapReduce program, compile it, run it, and examine the output. The goal of these lab exercises is to calculate the mean surplus or deficit. The existing code calculates the minimum and maximum values. Your output will include these values with the year that those extrema occurred.

Here is the output expected from the map and reduce code for this lab:

```
min(2009) : -1412688.0
max(2000) :  236241.0
mean:      -93862.0
```

Exercise	Duration
3.1: Copy lab files into your home directory	5 min
3.2: Modify the driver	5 min
3.3: Compile and run the map-only program	5 min
3.4: Implement code in the reducer	15 min
3.5: Compile and run the MapReduce program	10 min

Summary of Data

The data set we're using is the history of the United States federal budget from the year 1901 to 2012. The data was downloaded from <http://www.whitehouse.gov/omb/budget/Overview> and has been massaged for this exercise.

Here is a sample record from the data set:

```
1968 152973 178134 -25161 128056 155798 -27742 24917 22336 2581
```

The fields of interest in this exercise are the first and fourth fields (year and surplus or deficit). The second field is the total income derived from federal income taxes, and the third field is the expenditures for that year. The fourth field is the difference between the second and third fields. A negative value in the fourth field indicates a budget deficit and a positive value indicates a budget surplus.

Exercise 3.1: Copy the Lab Files

1. Create a directory as follows:

```
$ mkdir /user/user01/3
```

2. Copy the source code from your laptop into that directory

```
$ scp RECEIPTS_LAB.zip user01@node-ip:/user/user01/3
```

3. Change directory into the directory.

```
$ cd /user/user01/3
```

4. Unzip the RECEIPTS_LAB.zip file.

```
$ unzip RECEIPTS_LAB.zip
```

Exercise 3.2: Modify Code in the Driver

1. Change directory into the RECEIPTS_LAB directory.

```
$ cd /user/user01/3/RECEIPTS_LAB
```

2. Open the ReceiptsDriver.java source file with your favorite text editor.

```
$ vi ReceiptsDriver.java
```

3. Make the following changes to the driver

```
//TODO Change the name of the job to a string of your choice
```

HINT: Change the string “my receipts” to any string of your choice

```
//TODO Comment out the Reducer definition
```

HINT: The reducer is defined by the `setReducerClass()` method of the Job object

4. Save the ReceiptsDriver.java file.



Exercise 3.3: Compile and Run the Map-only MapReduce Program

1. Change directory into the `RECEIPTS_LAB` directory.

```
$ cd /user/user01/3/RECEIPTS_LAB
```
2. Execute the `rebuild.sh` script to compile your code.

```
$ ./rebuild.sh
```
3. Execute the `rerun.sh` script to run your code. This script will execute your code.

```
$ ./rerun.sh
```
4. Examine the output from your MapReduce job. Note you may need to wait a minute before the job output is completely written to the output files.

```
$ cat /user/user01/3/RECEIPTS_LAB/OUT/part*
```

Here is partial output expected for this exercise:

```
summary      1901_63
summary      1902_77
summary      1903_45
summary      1904_-43
summary      1905_-23
summary      1906_25
summary      1907_87
summary      1908_-57
summary      1909_-89
summary      1910_-18
summary      1911_11
```

If you did not obtain the results above, you'll need to revisit your Mapper class. Ask your instructor for help if you need. Once you obtain the correct intermediate results from the map-only code, proceed to the next exercise.



Exercise 3.4: Implement Code in the Reducer

In this exercise, you will implement code in the reducer to calculate the mean value. The code has already been provided to calculate minimum and maximum values.

Recall that your mapper code will produce intermediate results. One such record looks like this:

```
summary 1968_-25161
```

When you execute the code for this lab, there will only be one reducer (since there is only one key – “summary”). That reducer will iterate over all the intermediate results and pull out the year and surplus or deficit. Your reducer will keep track of the minimum and maximum values (as temp variables) as well as the year those values occurred. You will also need to keep track of the sum of the surplus or deficit and count of the records in order to calculate the mean value.

Recall that the output of your reducer will look like this:

```
min(2009): -1412688.0
```

```
max(2000): 236241.0
```

```
mean: -93862.0
```

1. Change directory into the `RECEIPTS_LAB` directory.

```
$ cd /user/user01/3/RECEIPTS_LAB
```

2. Open the `ReceiptsReducer.java` source file with your favorite text editor.

```
$ vi ReceiptsReducer.java
```

3. Make the following changes to the reducer

```
//TODO declare and initialize int values for sum and count
```

```
//TODO increment the sum by tempValue
```

```
//TODO increment the count by 1
```

```
//TODO declare and calculate float value for mean
```

```
//TODO set the keyText to "mean:"
```

```
//TODO write keyText and the mean to the context
```

4. Save the `ReceiptsReducer.java` file.

5. Open the `ReceiptsDriver.java` source file with your favorite text editor and make the following changes :

```
//TODO uncomment the Reducer definition
```

6. Save the `ReceiptsDriver.java` file.



Exercise 3.5: Compile and Run Your Code

1. Change directory into the `RECEIPTS_LAB` directory.

```
$ cd /user/user01/3/RECEIPTS_LAB
```

2. Execute the `rebuild.sh` script to compile your code.

```
$ ./rebuild.sh
```

3. Execute the `rerun.sh` script to run your code.

```
$ ./rerun.sh
```

4. Examine the output from your MapReduce job.

```
$ cat /user/user01/3/RECEIPTS_LAB/OUT/part*
```

Here is the output expected for this exercise:

```
min(2009): -1412688.0
```

```
max(2000): 236241.0
```

```
mean: -93862.0
```

Note: The solution for this lab is found in `RECEIPTS_SOLUTION.zip`.



Lesson 4: Write a MapReduce Program

Lab Overview

The high level objective of this lab is to write your first complete MapReduce program using the numerical summary pattern we've been focusing on. The lab provides generic templates for the map, reduce, and driver classes. The exercises guide you through how to calculate the minimum, maximum, and mean SAT verbal and SAT math scores over the whole data set.

You should use the lab solution for Lab 4 to guide you through this lab. The overall structure of both solutions is the same since it follows the same min-max-mean pattern. How you parse records in this lab will be different because the format of the UNIVERSITY records is different than the format of the RECEIPTS records.

To get started, login to a MapR node as your user and perform the following exercises.

Exercise	Duration
4.1: Copy lab files into your directory	5 min
4.2: Implement the mapper class	20 min
4.3: Implement the mapper class	45 min
4.4: Implement the reducer class	10 min
4.5: Compile and run the MapReduce program	10 min

Summary of Data

This lab examines data sampled from university students across North America. The data set can be downloaded from <http://archive.ics.uci.edu/ml/datasets/University>.

Not every record contains the same number of fields, but every record starts with the string “(def-instance” and ends with the string “))”. Each record contains information for a single university in the survey. Here is a sample record:

```
(def-instance Adelphi
  (state newyork)
  (control private)
  (no-of-students thous:5-10)
  (male:female ratio:30:70)
  (student:faculty ratio:15:1)
  (sat math 475)
  (expenses thous$:7-10)
  (percent-financial-aid 60)
  (no-applicants thous:4-7)
  (percent-admittance 70)
  (percent-enrolled 40)
  (academics scale:1-5 2)
  (social scale:1-5 2)
  (sat verbal 500)
  (quality-of-life scale:1-5 2)
  (academic-emphasis business-administration)
  (academic-emphasis biology))
```



Exercise 4.1: Copy the Lab Files to Your Directory

1. Create a directory in your directory as follows:

```
$ mkdir /user/user01/4
```

2. Copy the lab ZIP file into that directory from your laptop

```
$ scp UNIVERSITY_LAB.zip user01@node-IP:/user/user01/4
```

3. Change directory into your directory.

```
$ cd /user/user01/4
```

1. Unzip the lab ZIP file.

```
$ unzip UNIVERSITY_LAB.zip
```

Exercise 4.2: Implement the Driver Class

1. Change directory into the UNIVERSITY_LAB directory.

```
$ cd /user/user01/4/UNIVERSITY_LAB
```

2. Open the `UniversityDriver.java` source file with your favorite text editor.

```
$ vi UniversityDriver.java
```

3. Implement each TODO in the `UniversityDriver.java` file as follows:

```
// TODO check the CLI
// TODO: instantiate a new job with a unique name
// TODO: define driver, mapper, and reducer classes in job
// TODO: define input format for job
// TODO: define record delimiter in input format class to ")))"
```

```
HINT:  conf.set("textinputformat.record.delimiter",")))");
```

```
// TODO define output key and output value class
// TODO define input and output paths
// TODO: launch job synchronously
// TODO: instantiate a new configuration object
// TODO: call the ToolRunner run() method
```

4. Save the `UniversityDriver.java` file.



Exercise 4.3: Implement the Mapper Class

Recall that a single record contains an unknown number of fields after the start of the record and before either the “sat math” or “sat verbal” field.

```
(def-instance xxx
  . . .
  (sat verbal 500)
  . . .
  (sat math 475)
  . . .))
```

Or the “sat math” field may come before the “sat verbal” field. Or one or both of the “sat math” and “sat verbal” fields may not be part of the record at all. Skip records that do not contain both fields.

NOTE: the data set is not uniform from beginning to end. Examine the first few records in the file, and then skip to line 1000 or so in the file. Use the *less* or *view* command to examine the input data set (UNIVERSITY_LAB/DATA/university.txt).

1. Change directory into the UNIVERSITY_LAB directory.

```
$ cd /user/user01/4/UNIVERSITY_LAB
```

2. Open the UniversityMapper.java source file with your favorite text editor.

```
$ vi UniversityMapper.java
```

3. Implement each TODO in the UniversityMapper.java file as follows:

```
// TODO convert record (value) to String
// TODO determine if the record contains fields for both "sat verbal"
AND "sat math"
// TODO iterate through tokens until you find "sat verbal" field
// TODO split the "satverbal" field token
// TODO pull out the score from the "sat verbal" field
// TODO continue through iterator until you find "sat math" field
// TODO split the "sat math" field token
// TODO pull out the score from the "sat math" field
// TODO emit key-value as "satv", score
// TODO emit key-value as "satm", score
```

4. Save the UniversityMapper.java file.



Exercise 4.4: Implement the Reducer Class

Recall that one reducer will be given a list of key-value pairs that looks like this:

```
satv 480 500 530 . . .
```

The other reducer will be given a list of key-value pairs that looks like this:

```
satm 400 500 510 . . .
```

1. Change directory into the `UNIVERSITY_LAB` directory.

```
$ cd /user/user01/4/UNIVERSITY_LAB
```

2. Open the `UniversityReducer.java` source file with your favorite text editor.

```
$ vi UniversityReducer.java
```

3. Implement each TODO in the `UniversityReducer.java` file as follows:

```
// TODO loop through the values and calculate min, max, and mean
```

```
// TODO emit keystore_min, min
```

```
// TODO emit keystore_max, max
```

```
// TODO emit keystore_mean, mean
```

4. Save your `UniversityReducer.java` file.



Exercise 4.5: Compile and Run the MapReduce Program

Compile the MapReduce program

1. Launch the `rebuild.sh` script to recompile the source code.

```
$ ./rebuild.sh
```

2. If you have any errors you can't resolve, you may wish to check the output from your map phase by setting `mapred.num.reduce.tasks` to 0 in your configuration.

Run the MapReduce Program

Launch the `rerun.sh` script to execute the code.

```
$ ./rerun.sh
```

Check the Output of the Program

```
$ cat /user/user01/4/UNIVERSITY_LAB/OUT/part-r-00000
```

You should see something similar to what is shown below:

```
[user01@maprdemo UNIVERSITY_LAB]$ cat /user/user01/4/UNIVERSITY_LAB/OUT/part-r-00000
satm_min      325.0
satm_max      780.0
satm_mean     592.0
satv_min      300.0
satv_max      700.0
satv_mean     539.0
```

NOTE: the solution for this exercise is located in `UNIVERSITY_SOLUTION.zip`.



Lesson 5: Manage, Monitor, and Test MapReduce Jobs

The goal of the exercises in the lesson's lab is to use the tools and methods described in the lecture to manage and debug your MapReduce jobs and applications.

Login to one of the nodes in the MapR cluster as your user and perform the following exercises. Note that some commands may have multiple lines separated by a backslash character ("\
"). This indicates that the command should continue without typing a carriage return.

Exercise	Duration
5.1: Copy lab files to your directory	5 min
5.2: Examine default job output	30 min
5.3: Write MapReduce code to use custom counters	30 min
5.4: Write MapReduce code to use standard output, error, and logging	30 min
5.5: Use <code>MRUnit</code> to test a MapReduce application	45 min
5.6: Use the <code>hadoop</code> CLI to manage jobs	30 min

Exercise 5.1: Copy the Lab Files into Your Directory

1. Create a directory in your directory as follows:

```
$ mkdir /user/user01/5
```

2. Copy the source code into that directory

```
$ scp SLOW_LAB.zip user01@node-IP:/user/user01/5
```

```
$ scp VOTER_LAB.zip user01@node-IP:/user/user01/5
```

3. Change directory into the directory.

```
$ cd /user/user01/5
```

4. Unzip the source code ZIP files.

```
$ unzip SLOW_LAB.zip
```

```
$ unzip VOTER_LAB.zip
```

5. Uncompress the data file for the VOTER_LAB

```
$ gunzip /user/user01/5/VOTER_LAB/DATA/myvoter.csv.gz
```

6. Inject some faulty records into your data set. For example:

```
$ echo "0,james,14,independent,100,100" >>\
```

```
/user/user01/5/VOTER_LAB/DATA/myvoter.csv
```

```
$ echo "0,james,25" >>\
```

```
/user/user01/5/VOTER_LAB/DATA/myvoter.csv
```

Exercise 5.2: Examine Default Job Output

In this exercise, you will run the `teragen` and `terasort` MapReduce applications from the examples provided in the Hadoop distribution. You will then examine the records produced from running each one.

Run teragen

1. Run the `teragen` MapReduce application to generate 1000 records.

```
$ hadoop jar /opt/mapr/hadoop/hadoop-0.20.2/\
```

```
hadoop-0.20.2-dev-examples.jar \
```

```
teragen 1000 /user/user01/5/TERA_IN
```



2. Look at the `teragen` job output counters to examine output

Question: Why are there no input or output records for the reducer in the job output?

Answer: The `teragen` application is a map-only application.

2. Examine the files produced by `teragen` and answer the questions below.

- What type of file is produced?

```
$ file /user/user01/5/TERA_IN/part-m-0000*
```

- Why is the number of records we generated with `teragen` different than the total number of lines in the files?

```
$ wc -l /user/user01/5/TERA_IN/part-m-0000*
```

- Can you make sense out of the files by looking at them?

```
$ view /user/user01/5/TERA_IN/part-m-00000
```

Run `terasort`

1. Run the `terasort` application to sort those records you just created and look at the job output.

```
$ hadoop jar /opt/mapr/hadoop/hadoop-0.20.2/\
hadoop-0.20.2-dev-examples.jar \
terasort /user/user01/5/TERA_IN /user/user01/5/TERA_OUT
```

2. Look at the `terasort` standard output (in your terminal window)./rerun.sh to determine the following:

- Number of mappers launched

Question: is this equal to the number of input files?

Answer: yes (2)

- Number of map and reduce input and output records.

Question: When would the number of map input records be different than the number of map output records?

Answer: if the map method is doing any sort of filtering (e.g. dropping "bad" records).



- Number of combine input and output records

Question: what does this imply about the `terasort` application?

Answer: `terasort` does not use a combiner.

Exercise 5.3: Write MapReduce Code with Custom Job Counters

In this exercise, you will write the logic to identify a “bad” record in a data set, then define a custom counter to count “bad” records from that data set. This is what a “good” record looks like:

```
1,david davidson,10,socialist,369.78,5108
```

There are 6 fields total – a primary key, name, age, party affiliation, and two more fields you don’t care about. You will implement a record checker that validates the following:

- there are exactly 6 fields in the record
 - third field is a “reasonable” age for a voter
1. Change directory to the `VOTER_LAB` directory.

```
$ cd /user/user01/5/VOTER_LAB
```
 2. Open the `VoterDriver.java` file with the `view` command. What character separates the keys from the values in the records? Close the file.
 3. Open the `VoterMapper.java` file with your favorite editor. Which character is the value of the record tokenizing on? Keep the file open for the next step.
 4. Implement the TODOs below to validate the record.

```
//TODO check number of tokens in iterator
//TODO increment "wrong number of fields" counter and return if wrong
//TODO eat up the first two tokens
//TODO convert age to an int
//TODO validate the age is a reasonable age
//TODO increment "bad age" counter and return if bad
//TODO convert age to an IntWritable
//TODO pull out party from record
//TODO emit key-value as party-age
```
 5. Compile and execute the code. Based on the minimum, maximum, and mean values for voter ages, what do you conclude about the nature of the data set?



6. Examine the output in your terminal from the job to determine the number of bad records.
 - How many records have the wrong number of fields?
 - How many records have a bad age field?
 - Does the total number of bad records plus the total number of reduce input records equal the total number of map input records?

Exercise 5.4: Write MapReduce Code to Use Standard Output, Error, and Logging

In this exercise, you will generate standard error and log messages and then consume them in the MCS.

Modify the mapper to send messages to standard error

In this task, use standard error instead of job counters to keep track of bad records.

1. Change directory to the `VOTER_LAB` directory.

```
$ cd /user/user01/5/VOTER_LAB
```
2. Open the `VoterMapper.java` file with your favorite editor.

```
$ vi VoterMapper.java
```
3. Implement the following TODOs in the code
 1. Instead of incrementing the bad record counter for incorrect number of tokens, write a message to standard error. Include the bad record in the message.

HINT: Use `System.err.println()`
 2. Instead of incrementing the bad record counter for invalid age, write a message to standard error. Include the bad record in the message.

HINT: Use `System.err.println()`
4. Save the file.
5. Compile and execute the application.

```
$ ./rebuild.sh
```

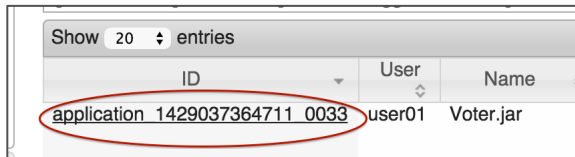
```
$ ./rerun.sh "-DXmx1024m"
```
6. Examine standard error messages for your job in the history viewer as follows:
 3. Launch the history viewer using the URL that is displayed in the terminal window



```
15/04/15 00:29:26 INFO mapreduce.Job: The url to track the job: http://mapdemo:8088/proxy/application_1429037364711_0033/
15/04/15 00:29:26 INFO mapreduce.Job: Running job: job_1429037364711_0033
```

NOTE: If you are using the MapR Sandbox for Hadoop, replace “mapdemo” with “localhost” or the *NodeIp*.

- Click the application id from the list



ID	User	Name
application_1429037364711_0033	user01	Voter.jar

- Click the Logs link from the Application overview page

NOTE: if you are using the MapR Sandbox for Hadoop, replace “mapdemo” with “localhost” or the NodeIP in the URL for the Logs.

Modify the mapper to write messages to syslog

In this task, you will perform the same logic as in the previous task, except you'll write the message to syslog.

- Change directory to the `VOTER_LAB` directory.

```
$ cd /user/user01/5/VOTER_LAB
```

- Open the `VoterMapper.java` file with your favorite editor.

```
$ vi VoterMapper.java
```

- Implement the following TODOs in the code

- Instead of incrementing the bad record counter or writing to standard error for incorrect number of tokens, write a message to syslog. Include the bad record in the message.

HINT: Use `log.error()` from Apache Commons Logging

- Instead of incrementing the bad record counter or writing to standard error for invalid age, write a message to syslog. Include the bad record in the message.

HINT: Use `log.error()` from Apache Commons Logging

- Save the file.



5. Compile and execute the application. Replace the “username” variable with your login user name.

```
$ ./rebuild.sh
$ ./rerun.sh "-DXmx1024m"
```

Examine syslog messages for your job

1. Change directory to the output directory for YARN jobs.

```
$ cd /opt/mapr/hadoop/hadoop-2.5.1/logs/userlogs
```
2. Change directory to your application output directory.

```
$ cd application_timestamp_appid
```
3. Change directory to the second container output directory.

```
$ cd container_timestamp_appid_01_000002
```
4. Display the contents of the `syslog` file. If you scroll through, you will see the messages with the bad age and incorrect tokens.

```
$ cat syslog
```

Exercise 5.5: Use MRUnit to Test a MapReduce Application

In this exercise, the code to test the mapper using `MRUnit` is already provided. You will follow that example to implement the reducer test.

Recall the `VoterMapper` map method emits the key-value pair: (party, age). For example, with input "1,david davidson,20,socialist,369.78,5108" you should expect output (socialist, 20).

Test the mapper against different input

1. Change directory to the `VOTER_LAB` directory.

```
$ cd /user/user01/5/VOTER_LAB
```
2. View the `mymaptest.dat` file with the `cat` command. The first line of the file is the input record. The second line of the file is the output from the map method for the input.

```
$ cat mymaptest.dat
```
3. Test the map method against the test file – you should get a "success" message.

```
$ ./retest.sh map mymaptest.dat
```
4. Now edit the test file so that the input and expected output do not match.
5. Test the map method against the test file – this time you should get an exception



```
$ ./retest.sh map mymaptest.dat
```

Implement code to test the reducer

1. Change directory to the `VOTER_LAB` directory.

```
$ cd /user/user01/5/VOTER_LAB
```

2. View the `myreducetest.dat` file with the `cat` command. The first line of the file is the input record (including the key and list of values). The second line of the file is the output from the reduce method for the associated input.

```
$ cat myreducetest.dat
```

3. Implement the TODO in the `VoterTest.java` file to write the unit test for the reducer:

```
//TODO declare the reduceDriver object
//TODO instantiate a reducer object and reducer driver
//TODO implement the testReducer method
//TODO create a list object for reduce input
//TODO tokenize the first line from the input file
//TODO pull out the key from the tokenized line
//TODO loop through tokens to add IntWritable to reduce input list
```

Build and execute the unit test for the reducer

1. Change directory to the `VOTER_LAB` directory.

```
$ cd /user/user01/5/VOTER_LAB
```

2. Edit the `VoterReducer.java` file such that only the key-value pair for the mean is emitted (comment out the lines that write the min and max to the context). This is because we are only examining the output for mean in our MRUnit test.

```
$ vi VoterReducer.java
```

3. Rebuild the jar file.

```
$ ./rebuild.sh
```

4. Run the reduce test against the input file.

```
$ ./retest.sh reduce myreducetest.dat
```

5. Make a change in the `myreducetest.dat` file so that the expected output intentionally does not match the expected output. Then retest the reduce method to see what the error looks like.

```
$ ./retest.sh reduce myreducetest.dat
```



Exercise 5.6: Use the Hadoop CLI to Manage Jobs

In this exercise, we use the Hadoop CLI to manage jobs.

Launch a long-running job

1. Change directory to the `SLOW_LAB` directory.

```
$ cd /user/user01/5/SLOW_LAB
```

2. Launch the MapReduce application and specify the sleep time (in ms).

```
$ ./rerun.sh "-DXmx1024m -D my.map.sleep=4000"
```

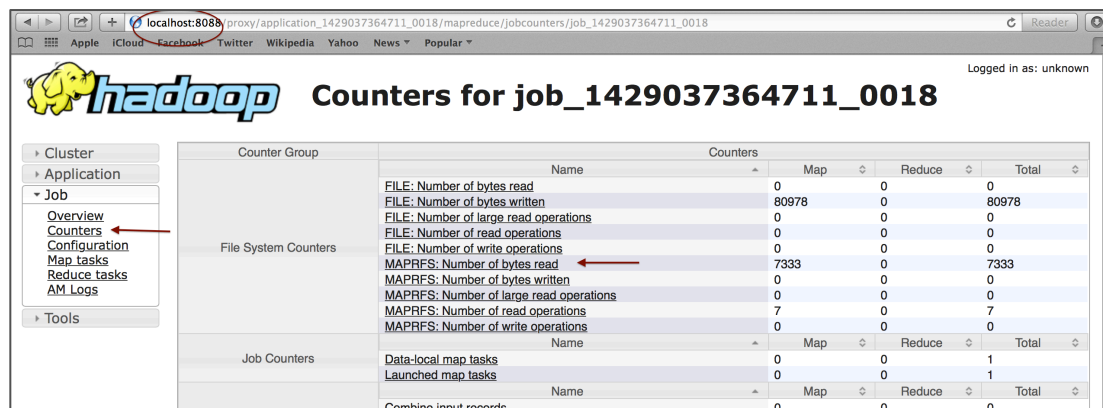
Get counter values for job

To view the job history viewer, open the URL that is shown in the terminal window when you run the above command.

```
[user01@maprdemo SLOW_LAB]$ ./rerun.sh "-DXmx1024m -D my.map.sleep=1000"
15/04/14 16:36:25 INFO client.MapRZKBasedRMFailoverProxyProvider: Updated RM address to maprdemo/10.0.2.15:8032
15/04/14 16:36:25 INFO input.FileInputFormat: Total input paths to process : 1
15/04/14 16:36:26 INFO mapreduce.JobSubmitter: number of splits:1
15/04/14 16:36:26 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1429037364711_0018
15/04/14 16:36:26 INFO security.ExternalTokenManagerFactory: Initialized external token manager class - com.mapr.h
adoop.yarn.security.MapRTicketManager
15/04/14 16:36:26 INFO impl.YarnClientImpl: Submitted application application_1429037364711_0018
15/04/14 16:36:26 INFO mapreduce.Job: The url to track the job: http://maprdemo:8088/proxy/application_14290373647
11_0018/
15/04/14 16:36:26 INFO mapreduce.Job: Running job: job_1429037364711_0018
15/04/14 16:36:41 INFO mapreduce.Job: Job job_1429037364711_0018 running in uber mode : false
```

NOTE: If you are using the MapR Sandbox for Hadoop, replace “maprdemo” with “localhost” or the nodeIP.

Display the job counter for `MAPRFS_BYTES_READ`.



Counter Group	Name	Map	Reduce	Total
File System Counters	FILE: Number of bytes read	0	0	0
	FILE: Number of bytes written	80978	0	80978
	FILE: Number of large read operations	0	0	0
	FILE: Number of read operations	0	0	0
	FILE: Number of write operations	0	0	0
	MAPRFS: Number of bytes read	7333	0	7333
	MAPRFS: Number of bytes written	0	0	0
	MAPRFS: Number of large read operations	0	0	0
	MAPRFS: Number of read operations	7	0	7
	MAPRFS: Number of write operations	0	0	0
Job Counters	Data-local map tasks	0	0	1
	Launched map tasks	0	0	1
	Combine input records	0	0	0

Kill the job

1. Kill your job. Replace the *applicationid* variable appropriately.



```
$ yarn application -kill applicationID
```

2. Verify your job is no longer running.

```
$ yarn application -list
```

(Appendix: if the job hangs, change the memory allocation for filesystem in the warden.conf file.

```
service.command.mfs.heapsize.max=1024
```



Lab 6: Manage MapReduce Job Performance

The goal of the exercises in this lesson's lab is to use the tools and methods in lecture to learn to de-tune a job and measure the performance.

Exercise	Duration
6.1: De-tune a job and measure its impact on performance	30 min

Exercise 6.1: De-Tune a Job and Measure Performance Impact

In this exercise, you will actually "de-tune" a job and monitor its performance.

Establish a baseline for CPU time

In this task, you will run the `teragen` and `terasort` MapReduce applications from the examples provided in the Hadoop distribution.

1. Create the lab directory.

```
$ mkdir /user/user01/6
```
2. Run the `teragen` MapReduce application to generate 1,000,000 records.

```
$ hadoop jar /opt/mapr/hadoop/hadoop-0.20.2/hadoop-0.20.2-dev-\  
examples.jar teragen 1000000 /user/user01/6/TERA_IN
```
3. Run the `terasort` application to sort those records you just created.

```
$ hadoop jar /opt/mapr/hadoop/hadoop-0.20.2/hadoop-0.20.2-dev-\  
examples.jar terasort -DXmx1024m -Dmapred.reduce.tasks=2 \  
/user/user01/6/TERA_IN /user/user01/6/TERA_OUT_1
```
4. Determine the aggregate map phase run time of the job. You can do this by going to the jobhistory viewer. The url for this is provided in the terminal window as shown below.

```

-Dmapred.reduce.tasks=2 /user/user01/6/TERA_IN/ /user/user01/6/TERA_OUT_1
15/04/14 13:48:07 INFO terasort.TeraSort: starting
15/04/14 13:48:09 INFO input.FileInputFormat: Total input paths to process : 2
Spent 63ms computing base-splits.
Spent 12ms computing TeraScheduler splits.
Computing input splits took 79ms
Sampling 2 splits of 2
Making 2 from 100000 sampled records
Computing partitions took 614ms
Spent 698ms computing partitions.
15/04/14 13:48:09 INFO client.MapRZKBasedRMFailoverProxyProvider: Updated RM address to maprdemo/10.0.2.15:8032
15/04/14 13:48:10 INFO mapreduce.JobSubmitter: number of splits:2
15/04/14 13:48:10 INFO Configuration.deprecation: mapred.reduce.tasks is deprecated. Instead, use mapreduce.job.re
duces
15/04/14 13:48:11 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1429037364711_0006
15/04/14 13:48:11 INFO security.ExternalTokenManagerFactory: Initialized external token manager class - com.mapr.h
adoop.yarn.security.MapRTicketManager
15/04/14 13:48:11 INFO impl.YarnClientImpl: Submitted application application_1429037364711_0006
15/04/14 13:48:11 INFO mapreduce.Job: The url to track the job: http://maprdemo:8088/proxy/application\_1429037364711\_0006/
15/04/14 13:48:11 INFO mapreduce.Job: Running job: job_1429037364711_0006
15/04/14 13:48:28 INFO mapreduce.Job: Job job_1429037364711_0006 running in uber mode : false
15/04/14 13:48:28 INFO mapreduce.Job: map 0% reduce 0%

```

NOTE: If you are using the MapR Sandbox for Hadoop, replace “maprdemo” in the URL with “localhost” or the node IP address.

From the Job History viewer, you can find the map phase run time as shown below.

JOB_1429037364711_0006

☐
☐
☐
☒ **n**
☐ **s**
☐

Job Over

Job Name: TeraSort
User Name: user01
Queue: root.user01
State: SUCCEEDED
Uberized: false
Submitted: Tue Apr 14 13:48:11 PDT 2015
Started: Tue Apr 14 13:48:26 PDT 2015
Finished: Tue Apr 14 13:49:07 PDT 2015
Elapsed: 41sec
Diagnostics:

Average Map Time
7sec

Average Shuffle Time
5sec

Average Merge Time
1sec

Average Reduce Time
2sec

5. Run it a few times more to establish a "good" baseline. Remove the output directory /user/user01/6/TERA_OUT_1 before each run. Here are some values for a few runs. Fill in the aggregate map phase run times below the ones given for your runs.

Run 1	Run 2	Run 3	Run 4
2m1s=121s	2m30s=150s	2m10s=130s	1m55s=115s

Modify the configuration and determine impact

In this task, you will run the `teragen` and `terasort` MapReduce applications from the examples provided in the Hadoop distribution.



1. Run the `terasort` application to sort those records again, but this time with a modification in the job parameters.

```
$ hadoop jar /opt/mapr/hadoop/hadoop-0.20.2/hadoop-0.20.2-dev-
examples.jar terasort -DXmx1024m -Dmapred.reduce.tasks=2 \
-Dio.sort.mb=1 /user/user01/6/TERA_IN /user/user01/6/TERA_OUT_2
```

2. Determine the aggregate time spent in the map phase. Use the job history viewer as described earlier.
3. Run it a few times more to establish a "good" test. Change the name of the output directory for each rerun. Here are some values for a few runs. Fill in the results below the ones given for your runs.

Run 1	Run 2	Run 3	Run 4
7m13s=433s	5m5s=305s	5m49s=349s	5m21s=321s

NOTE: there is a **significant** difference in time between the first run and the rest of the runs. Without accounting for it, this is one reason we take several samples when benchmarking. Statistically, we should probably throw out this outlier.

4. It appears that the change has impacted the amount of time spent in the map phase (which makes sense given we are changing the `io.sort.mb` parameter). Calculate the speedup (or in this case, slowdown) due to the configuration change.

- Average aggregate time baseline = $(121 + 150 + 130 + 115) / 4$
= 129 seconds
- Average aggregate time enhanced = $(305 + 349 + 321) / 3$
= 325 seconds
- Speedup = baseline/enhanced - 1 = $(129 / 325) - 1 = -0.60$

In other words, the "enhanced" job performs 60% slower than the baseline.



Lab 7: Working With Data

The goal of this exercise is to use the tools and methods learned in lecture to have a hands-on experience of using other data. You will run a MapReduce program that uses HBase as a source.

Exercise	Duration
7.1: Run a MapReduce program to use HBase as source	30 min

Exercise 7.1: Run a MapReduce Program to Use HBase as Source

In this exercise, you will create and populate a table in HBase to store the voter data from previous exercises. You will then run a MapReduce program to calculate the usual maximum, minimum, and mean values using data read from that table.

Create an HBase Table Using `importtsv`

In this task, you will use the `importtsv` utility to create an HBase table using the tab-separated VOTER data we used in a previous lab. Log in to one of your cluster nodes to perform the steps in this task and use the `hbase` command to create an empty table in MapR-FS.

1. Create a directory for this lab.

```
$ mkdir /user/user01/7
```
2. Copy the jar file to the lab directory.

```
$ scp VOTERHBASE_SOLUTION.zip user01@node-IP:/user/user01/7
```
3. Extract the jar file.

```
$ cd /user/user01/7
$ unzip VOTERHBASE_SOLUTION.zip
```
4. Launch the hbase shell:

```
$ hbase shell
```
5. Create the table in MapR-FS.

```
hbase> create '/user/user01/7/myvoter_table', {NAME => 'cf1'}, {NAME => 'cf2'}, {NAME => 'cf3'}
```
6. Verify the file was created in MapR-FS.

```
hbase> quit
$ ls -l /user/user01/7/myvoter_table
```
7. Use the `importtsv` utility to import the data into the HBase table.

```
$ hadoop jar /opt/mapr/hbase/hbase-0.98.9/lib/hbase-server-\
0.98.9-mapr-1503.jar importtsv -Dimporttsv.columns=\
HBASE_ROW_KEY,cf1:name,cf2:age,cf2:party,cf3:contribution_amount,\
cf3:voter_number /user/user01/7/myvoter_table \
/user/user01/7/VOTERHBASE_SOLUTION/myvoter.tsv
```
8. Use the `hbase` command to validate the contents of the new table.

```
$ echo "scan '/user/user01/7/myvoter_table'" | hbase shell
```



```

ROW                COLUMN+CELL
 1                column=cf1:name, timestamp=1406142938710, value=d
                    avid davidson
 1                column=cf2:age, timestamp=1406142938710, value=49
 1                column=cf2:party, timestamp=1406142938710, value=
                    socialist
 1                column=cf3:contribution_amount, timestamp=1406142
                    938710, value=369.78
 1                column=cf3:voter_number, timestamp=1406142938710,
                    value=5108
10                column=cf1:name, timestamp=1406142938710, value=o
                    scar xylophone
. . . <output omitted>
1000000 row(s) in 1113.9850 seconds

```

Run the MapReduce Program to Calculate Statistics

In this task, you will run your MapReduce program and then analyze the results.

1. Change directory to the location of the MapReduce program jar file.

```
$ cd /user/user01/7/VOTERHBASE_SOLUTION
```

2. Run the MapReduce program.

```
$ java -cp `hbase classpath`:VoterHbase.jar \
VoterHbase.VoterHbaseDriver \
/user/user01/7/myvoter_table \
/user/user01/7/OUT
```

3. Analyze the results (min, max, and mean age). Do you notice anything unusual about the results?

```
$ cat /user/user01/7/OUT/part-r-00000
democrat 18.0
democrat 77.0
democrat 47.0
green 18.0
green 77.0
green 47.0
independent 18.0
```



independent	77.0
independent	47.0
libertarian	18.0
libertarian	77.0
libertarian	47.0
republican	18.0
republican	77.0
republican	47.0
socialist	18.0
socialist	77.0
socialist	47.0



Lesson 8: Launching Jobs

The goal of the exercise in the lesson's lab is to use the tools and methods in lecture to be able to launch multiple jobs. This lab has only one exercise with several tasks.

Exercise	Duration
8.1: Write a MapReduce Driver to launch two jobs	30 min



Exercise 8.1: Write a MapReduce Driver to Launch Two Jobs

In this exercise, you will modify a MapReduce driver that launches two jobs. The first job calculates minimum, maximum, and mean values for the SAT verbal and math scores. The second job calculates the numerator and denominator for the Spearman correlation coefficient between the verbal and math scores. The driver then calculates the correlation coefficient by dividing the numerator by the square root of the denominator. The code for both MapReduce jobs has been provided.

Pearson's Correlation Coefficient

This statistic is used to determine the level of dependence (or correlation) between two variables. The value of the coefficient ranges from -1 to +1, where

- +1 → directly proportional (high positive correlation)
- 0 → no correlation
- -1 → inversely proportional (high negative correlation)

The formula to calculate this coefficient is given here:

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$$

In words, the numerator is the sum of the products of the differences between each value and its mean. The denominator is the square root of the sum of the squares of the differences between each value and its mean.

Programming Objective

Let X represent the SAT verbal scores and Y represent the SAT math scores. The first MapReduce job calculates the mean values for X and Y, and the second MapReduce job calculates the numerator and the squared value of the denominator. The driver you write must configure and launch both jobs and then calculate the Spearman correlation coefficient.

Copy Lab Files to Cluster

In this task, you will copy the lab files to your cluster and extract them in a folder for use in this exercise.

1. Create a directory for this exercise.

```
$ mkdir /user/user01/8
```

2. Copy the lab ZIP file to your cluster.

```
$ scp STATISTICS_SOLUTION.zip user01@node-IP:/user/user01/8
```



3. Extract the ZIP file.

```
$ cd /user/user01/8
$ unzip STATISTICS_SOLUTION.zip
```

Implement the TODOs in the `WholeJobDriver.java` File

In this task, you will configure the `WholeJobDriver.java` file to configure and launch the two MapReduce jobs and then calculate the correlation coefficient.

- `//TODO: set driver, mapper, and reducer classes for first job (university)`
- `//TODO: set input format, output key, and output value classes for university job`
- `//TODO: set input and output paths for university job`
- `//TODO: launch the first job synchronously`
- `//TODO: set driver, mapper, and reducer classes for the second job (stat)`
- `//TODO: set input format, output key, and output value classes for stat job`
- `//TODO: set input and output paths for stat job`
- `//TODO: launch the second job synchronously`
- `//TODO: return 0 or 1 based on the return codes from your 2 jobs`

Run the MapReduce Jobs

In this task, you will launch the `WholeJobDriver` to calculate the Spearman correlation coefficient.

1. Run the `whole_rerun.sh` script.
2. Your results should be as follows:

```
product_sumofsquares is 243128.0
var1_sumofsquares is 259871.0
var2_sumofsquares is 289679.0
spearman's coefficient is 0.886130250066755
```



Lab 9: Using Non-Java Languages for MapReduce Applications (Streaming)

The goal of the exercises in this lesson's lab is to use non-Java programming to implement a MapReduce jobs. This lab has one exercise. This lab is not required.

Exercise	Duration
9.1: Implement a MapReduce streaming application	30 min

Exercise 9.1: Implement a MapReduce Streaming Application

In this exercise, you will implement a MapReduce streaming application using the language of your choice (Python or Perl). Guidance will be provided for building the application in the UNIX `bash` shell. We return to the `RECEIPTS` data set to calculate the minimum, maximum, mean and the years associated with the those values.

Copy the Lab Files to Your Cluster

In this task, you will copy and extract the lab files for this exercise.

1. Create a directory for this exercise.

```
$ mkdir /user/user01/9
```

2. Copy the lab ZIP file to this directory.

```
$ scp STREAMING_RECEIPTS.zip user01@node-IP:/user/user01/9
```

3. Extract the lab ZIP file.

```
$ cd /user/user01/9
```

```
$ unzip STREAMING_RECEIPTS.zip
```

Implement the Mapper

In this task, you will implement the mapper in the language of your choice (Python or Perl) based on the following `bash` shell implementation of the same logic.

```
#!/bin/bash
while read record
do
    year=`echo $record | awk '{print $1}'`
    delta=`echo $record | awk '{print $4}'`
    printf "summary\t%s_%.s\n" "$year" "$delta"
done
```

Implement the Reducer

In this task, you will implement the reducer in the language of your choice (Python or Perl) based on the following `bash` shell implementation of the same logic.

```
#!/bin/bash -x
count=0
sum=0
max=-2147483647
min=2147483647
minyear=""
maxyear=""
while read line
do
```



```

value=`echo $line | awk '{print $2}'`
if [ -n "$value" ]
then
    year=`echo $value | awk -F_ '{print $1}'`
    delta=`echo $value | awk -F_ '{print $2}'`
fi
if [ $delta -lt $min ]
then
    min=$delta
    minyear=$year
elif [ $delta -gt $max ]
then
    max=$delta
    maxyear=$year
fi
count=$(( count + 1 ))
sum=$(( sum + delta ))
done

mean=$(( sum / count ))
printf "min year is %s\n" "$minyear"
printf "min value is %s\n" "$min"
printf "max year is %s\n" "$maxyear"
printf "max value is %s\n" "$max"
printf "sum is %s\n" "$sum"
printf "count is %s\n" "$count"
printf "mean is %d\n" "$mean"

```

Launch the Job

In this task, you'll modify a run script to match your environment and then run the MapReduce streaming job.

1. Modify the `receipts_driver.sh` script to match the paths to your language choice (Python or Perl), naming convention, and locations for input and output.

```

#!/usr/bin/env bash

USER=`whoami`

# 1) test map script

echo -e "1901 588 525 63 588 525 63" | ./receipts_mapper.sh | od -c

# 2) test reduce script

echo -e "summary\t1901_63" | ./receipts_reducer.sh | od -c

# 3) map/reduce on Hadoop

export JOBHOME=/user/$USER/9/STREAMING_RECEIPTS

export CONTRIB=/opt/mapr/hadoop/hadoop-2.5.1/share/hadoop/tools/lib

export STREAMINGJAR=hadoop-streaming-2.5.1-mapr-1503.jar

```



```
export THEJARFILE=$CONTRIB/$STREAMINGJAR
rm -rf $JOBHOME/OUT
hadoop jar $THEJARFILE \
  -mapper 'receipts_mapper.sh' \
  -file receipts_mapper.sh \
  -reducer 'receipts_reducer.sh' \
  -file receipts_reducer.sh \
  -input $JOBHOME/DATA/receipts.txt \
  -output $JOBHOME/OUT
```

2. Launch the MapReduce streaming job.

```
$ ./receipts_driver.sh
```

3. Examine the output.

```
$ cat /user/user01/9/OUT/part-r-00000
```

```
min year is 2009
min value is -1412688
max year is 2000
max value is 236241
sum is -10418784
count is 111
mean is -93862
```

