THE UNIVERSITY OF SYDNEY

School of Information Technologies
A/Prof Uwe Roehm

COMP5349: Cloud Computing **1.Sem./2017**

Week 4: MapReduce Programming

30.03.2017

Tutorial Practice

Question 1: Execute Sample Hadoop Programs

Log in to one of the slave nodes using your `unikey` and password.

Java hadoop job(s) is usually packed as a jar file and submitted to the cluster using hadoop script. The command format is:

```
hadoop jar PathToJarFile [mainClass] [ProgramArgLists]
```

All Hadoop distribution comes with an example jar containing a few typical MapReduce job(s), such as `pi`, `grep` and so on. The following command runs the `grep` example on `place.txt` to look for all occurrences with the word "Australia" in it

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar
grep /share/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 `/share/place.txt` specifies the input location; `placeout` specify the output directory; the last one is the regular expression. A regular expression is a concise representation of a pattern in text, which we're using here to search through some input. You can learn more about regular expressions here: `http://www.computerhope.com/jargon/r/regex.htm`.

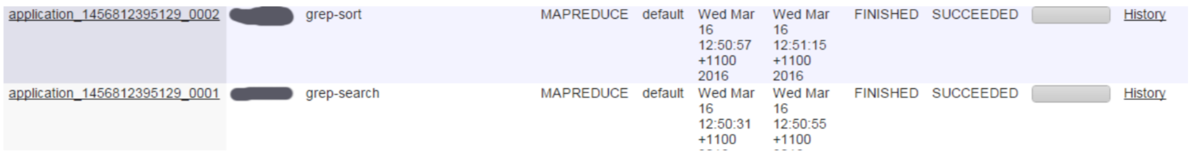
After submitting the job successfully, progress information will be printed out on the console as the job starts to execute. Application progress and summary information can be viewed from YARN Resource manager WebUI as well.

`http://soit-hdp-pro-1.ucc.usyd.edu.au:8088/cluster/apps`.

Figure 1 shows that the `grep` example has started two applications/jobs: `grep-search` and `sort`

Detailed job information can be viewed from the Job History Server by clicking the `history` link of each job.

After all jobs finish running successfully, go to Namenode WebUI `http://soit-hdp-pro-1.ucc.usyd.edu.au:50070/` to inspect the `placeout` directory under your home directory.



application_1456812395129_0002	[redacted]	grep-sort	MAPREDUCE	default	Wed Mar 16 12:50:57 +1100 2016	Wed Mar 16 12:51:15 +1100 2016	FINISHED	SUCCEEDED		History
application_1456812395129_0001	[redacted]	grep-search	MAPREDUCE	default	Wed Mar 16 12:50:31 +1100	Wed Mar 16 12:50:55 +1100	FINISHED	SUCCEEDED		History

Figure 1: Application Summary on YARN Resource Manager WebUI

You should see a file named `part-r-00000`. All Hadoop output files are named in this style. The number indicates the index of the reducer that produces it. If your job has two reduce tasks, there will be two result files: `part-r-00000` and `part-r-00001`. Output produced by map task will be named as `part-m-0000x`. You can use the `hdfs dfs -get` command to download the result or use the `hdfs dfs -cat` command to inspect the content (but be careful not to use `-cat` with very large files).

Question 2: Build an Executable Hadoop Program from Java Source Code

a) Download the Java Source Code

Run the following commands to download a tar file, extract the content, and build a jar file `userTag.jar` in the current directory:

```
wget http://web.it.usyd.edu.au/~comp5349/2016/code/userTagSrc.tar
tar xvf userTagSrc.tar
ant
```

b) Execute the program

The program takes two parameters: an input path and an output directory. You can submit the job to Hadoop cluster using a small input file `partial.txt` with the following command:

```
hadoop jar userTag.jar usertag.TagDriver /share/partial.txt userTagOut
```

If you want to play with large input files, you can use any of the `n0*.txt` file under `/share/photo` folder.

c) The input and output of the program

Use `hdfs dfs -cat` or `hdfs dfs -tail` command to inspect the input and the output of the program. [Note: if the textfile is too big, you are recommended to download it from HDFS and use `head` or `less` to inspect them. HDFS does not implement all text viewing commands in Linux]The small input data set can be found under the HDFS directory `/share`. It consists of many lines of text. 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.

The program builds 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,
```

d) The design of the program

You can view/modify the source code using a text editor in Linux. Running the command `ant` again will generate a new jar.

The Java source contains three classes: a `TagDriver`, a `TagMapper` and a `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, ...}
```

Question 3: Python source code of the tag-owner index program

We also provide a Python version of the same program (credit to Andrian Yang). Run the following `wget` command to download a zip file containing the Python source code.

```
wget http://web.it.usyd.edu.au/~comp5349/2016/code/py_user_tag.zip
unzip py_user_tag.zip
```

There are 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. The `tag_driver.sh` shows you how to run the python script using the streaming jar with all necessary arguments.

Question 4: Write Your Own Code

Now try to add a combiner to the given source code.

The simple program does not use a combiner to perform intermediate result aggregation on the map side. If a user uses a particular tag a thousand times. This user’s id will appear in the map output, as well as reduce input a thousand times. This ends up with large volume of data shuffled between map and reduce phases. You are asked to add a combiner in the map stage to decrease the amount of data shuffled between map and reduce phases. You can design the logic of the combiner during the tutorial and leave the implementation as homework.

Java programmers can follow the lecture slides to see how to specify combiner in the driver. Python programmers can use `tag_driver_combiner.sh` as a template to run your combiner version.

IMPORTANT TIPS

- Output directory

  The MapReduce framework will always create a new output directory for you. 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.

- How to obtain aggregated logs

  If a job fails, the error message printed on the console might not be detailed enough to diagnose the real issue. We have configured to aggregate all logs related to an application in HDFS and you can obtain it using the following yarn command:

  ```
  yarn logs -applicationId [yourAppID]
  ```

  You can find your application ID from the YARN Web UI.

- How to develop Java MapReduce program locally using an IDE

  Compiling Java MapReduce program requires many MapReduce related jars. If you want to use an IDE such as `Eclipse` to inspect and compile Java MapReduce program, you are recommended to use `maven` to mange the dependencies. The cluster has MapReduce version 2.7.2 installed. Here is the corresponding repository: `https://maven-repository.com/artifact/org.apache.hadoop/hadoop-mapreduce/2.7.2`

The cluster nodes have several java version installed including 1.7 and 1.8. The default one is 1.7, make sure you compile your code using the same Java version.

- How to do the tutorial in the Azure HDInsight cluster

  Different Hadoop installation will have slightly different directory structure. If you want to run the tutorial in Azure HDInsight cluster, make sure you find the location of the jar files for compiling java code or for submitting Python code.

- How to Kill Jobs

  If your job starts running away (for instance, running for more than 5 or 10 minutes using the partial.txt), you should kill it to release the resources to other students. To kill your job, you need to find out its job id. You can find it from the job history server or from the shell command `mapred job -list`. Both will show all running jobs. The job id is just a string starting with "job_", then a timestamp and a counter, e.g. "job_201303110531_0009"

  To kill a job, run the shell command `mapred job -kill job-id`.

  When you count the job's running time, remember to exclude the queuing time. When there are many users, most jobs are queued for a while before they run.

- Always Test Your Code Before Running on Very Large Data Set

  Before submitting your job on very large data set, it is always good practice to test them using JUnit for basic map/reduce logic and to run it on a much smaller data set.