Extreme computing lab exercises
Session two

Miles Osborne (originally: Saša Petrović)

October 19, 2012

1 Managing jobs via web interface and the command line

1.1 Using the command line

Listing jobs

Running the following command will list all (completed and running) jobs on the cluster:

```
hadoop job -list all
```

This will produce a list similar to that shown in table 1. Job IDs (first column) are very important as they uniquely identify a job. This will be useful in the next section.

Job status

To get the status of a particular job, we can use

```
hadoop job -status <jobid>
```

where the <jobid> is ID of a job found in the first column of table 1. The status will show the percent of completion of mappers and reducers, along with a tracking URL and the location of a file with all the information about the job. We will soon see that the web interface provides much more details.

Killing jobs

To kill a job, run the following command:

```
hadoop job -kill <jobid>
```

◁ **Task**

where <jobid> is the ID of the job you want to kill. To try this, copy the file `/user/sasa/source/sleeper.py` somewhere on your local filesystem (NOT HDFS) and run it as a streaming job (specifying sleeper.py as the mapper, with no reducers, remember to use the `-file` option). You can set the input to be anything you like because the mapper doesn't actually do anything (except wait for you to kill it), and also remember to set the output to be a directory on HDFS that does not exist. It might be helpful if you name your job to something familiar – this will reduce the time needed to find its ID later. Open another terminal, log into the Hadoop cluster, and list all the jobs. Find the ID of the job you just started (use the name you gave it for faster searching). Find out the status of your job, and finally kill it. After you run the kill command, look at the terminal where you originally started the job and watch the job die.

1.2 Using the web interface

◁ **Task**

All of the previous actions can also be performed using the web interface. Opening a browser and going to

```
http://hcrc1425n01.inf.ed.ac.uk:50030/jobtracker.jsp
```

Table 1: Job listing.

14 jobs submitted
States are:
Running : 1 Succeded : 2 Failed : 3 Prep : 4

JobId	State	StartTime	UserName	Priority	SchedulingInfo
job_201009151640_0001	2	1285081662441	s0894589	NORMAL	NA
job_201009151640_0002	2	1285081976156	s0894589	NORMAL	NA
job_201009151640_0003	2	1285082017461	s0894589	NORMAL	NA
job_201009151640_0004	2	1285082159071	s0894589	NORMAL	NA
job_201009151640_0005	2	1285082189917	s0894589	NORMAL	NA
job_201009151640_0006	2	1285082275965	s0894589	NORMAL	NA
job_201009151640_0009	2	1285083343068	s0894589	NORMAL	NA
job_201009151640_0010	3	1285106676902	s0894589	NORMAL	NA
job_201009151640_0012	3	1285106959588	s0894589	NORMAL	NA
job_201009151640_0013	3	1285107094387	s0894589	NORMAL	NA
job_201009151640_0014	2	1285107283359	s0894589	NORMAL	NA
job_201009151640_0015	2	1285109169514	s0894589	NORMAL	NA
job_201009151640_0016	2	1285109271188	s0894589	NORMAL	NA
job_201009151640_0018	1	1285148710573	s0894589	NORMAL	NA

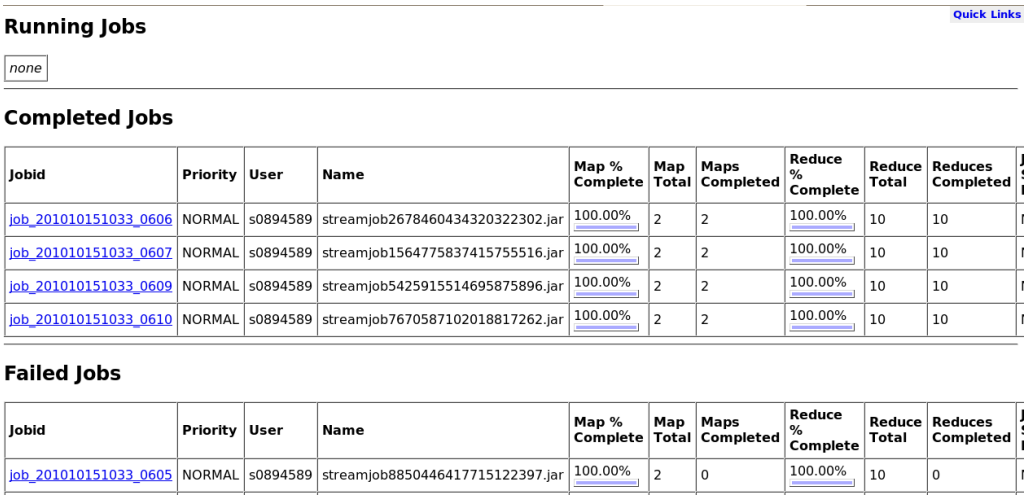

Figure 1: Web interface of the job tracker showing a list of running, completed, and failed jobs.

will show the web interface of the jobtracker. An example of what the interface looks like is shown in figure 1. We can see a list of running, completed, and failed jobs. Clicking on a job ID is similar to requesting its status from the command line, but it shows much more details, including the number of bytes read/written to the filesystem, number of failed/killed task attempts, and nice graphs of job completion.

2 Using side information

It is often useful to package other, external files together with the job. For example, if your application uses a dictionary or a file that stores some configuration settings, one would want these files to be available to the program just as they would in a non-mapreduce setting. This can be achieved using the `-file` option that we already used to package the source files. The following program takes a dictionary and counts only those words that appear in the dictionary, ignoring everything else. First copy the `mapper-dict.py` from `/user/sasa/source` to a LOCAL directory:

```
hadoop fs -get /user/sasa/source/mapper-dict.py ~/
```

Now copy the dictionary to a LOCAL directory:

```
hadoop fs -copyToLocal /user/sasa/data/dict.eng ~/
```

Run the program by typing (assuming you still have example3 in your input directory and that your output directory doesn't exist): ◁ **Task**

```
hadoop jar contrib/streaming/hadoop-0.20.2-streaming.jar \
           -input /user/sXXXXXXX/data/input/example3 \
           -output /user/sXXXXXXX/data/output \
           -mapper mapper-dict.py \
           -file ~/mapper-dict.py \
           -reducer aggregate \
           -file ~/dict.eng
```

The program will use `dict.eng` as the dictionary and count only those words that appear in that list. Look at the source of `mapper-dict.py` to see how to open the dictionary file.

3 Your first Hadoop program

The final task is to write your own crawler which runs on Hadoop. The input is a list of URLs that need to be crawled, and the output should be a list of crawled HTML documents, one per line. Copy the list of URLs that need to be crawled from `/user/sasa/data/urls` to your `input` directory. Make sure you delete any other files from the `input` directory before running your crawler. ◁ **Task**

As in the real world, some of the URLs will be missing or will have other problems so we want to make sure we log that. In case there is an error while retrieving a document, the output should contain a line such as

```
<url> <error>
```

where **<url>** is the URL which caused the error, and **<error>** is the actual error encountered (e.g., 404 Not found, or 403 Forbidden). The following example shows a little snippet that crawls a given URL in Python:

Listing 1: Print the contents of a web page to stdout

```
import urllib2

for line in urllib2.urlopen('http://www.asciitable.com/'):
    print line
```