School of Computer Science

COMP30640

Project 1
MapReduce Engine in Bash

Teaching Assistant:	Ellen Rushe
Coordinator:	Anthony Ventresque
Date:	Thursday 9th November, 2017
Total Number of Pages:	7

General Instructions

- This project has three main parts. You will successively create (i) a simple, single node MapReduce engine; (ii) a more advanced scenario (an you'll analyse of the performance of different architectures); and (iii) a fully distributed MapReduce engine. It can be difficult to understand all the concepts and features of this project a priori, so I really encourage you to start coding as soon as possible, reading every section/-subsection/paragraph, one at a time, and implementing what is described. It should all make sense when you have the project implemented.

- You are encouraged to collaborate with your peers on this project, but all written work must be your own. In particular we expect you to be able to explain every aspect of your solution if asked.

- We ask you to hand in an archive (zip or tar.gz) of your solution: code/scripts, README.txt file describing how to run your programs, a 5-10 page pdf report of your work (no need to include code in it).

- The report should include the following sections:

  1. a short introduction
  2. a requirement section that answers the question *what* is the system supposed to do
  3. an architecture/design section that answers the question *how* your solution has been designed to address the requirements described in the previous section
  4. a series of sections that describe the different challenges you faced and your solutions. For instance, take one of the script, describe the difficulty you faced and your solution. These sections can be short – the objective here is to show how you crafted the solutions with the tools you have learned so far. Add some testing elements (e.g., "to make sure this particular feature works I ran such and such tests").
  5. a short conclusion

- **a presentation sometime?**

- The project is worth **30% of the total grade** for this module. The breakdown of marks for the project will be as follows:

  – Single Node MapReduce: 50%
  – Advanced Scenario and Analysis: 10%
  – Distributed Solution: 15%
  – Report: 25%

- **Due date: 08/12/2017**

1 Introduction. *Big Data and MapReduce*

‘Big Data refers to datasets that are too big, or change too quickly, for traditional data management and data processing approaches. Big Data has forced the field of data management to rethink some of its design concepts and architectural patterns. In particular, Big Data always requires parallel and distributed processing to be able to cope with the large datasets that it is supposed to process.

Big Data platforms (e.g., Hadoop, Spark, Hive etc.) can be seen as massively parallel and distributed systems – sort of operating systems able to manage large number of processes/threads running data analysis jobs, at scale. In this project, we will try to emulate one particular type of Big Data system, MapReduce. MapReduce is based on two main ideas: (i) it’s better to move small programs/processes than to move data around in a cluster of machines running Big Data jobs, and (ii) the implementation of the jobs has to be simple (developers of MapReduce jobs should not have to know the complexity of the system).

So when they use MapReduce, developers only have to implement two functions: **map** and **reduce** that are usually very short programs:

map functions (there is usually a large number of these) process the (raw) data and output an intermediary, processed, result in the form of key/value pairs.

reduce functions (there is also usually a large number of these) process the key/value pairs given by the map functions and output the final result.

Let’s take an example. Let’s say we want to define a MapReduce task that can count the number of words in a Shakespeare play. The output would be in the form of key/value pairs, such as, $\{(the, 300), (in, 256), (sooth, 15), \cdots\}$ which means there are 300 ‘the’, 256 ‘in’ etc. in the play. The input (a play) can be divided in multiple files, for instance one per scene. Assume there are 20 scenes in the play. There could then be 20 map functions, each of them processing in parallel the 20 scenes, and creating outputs always in the following simple form: $\{(in, 1), (sooth, 1), (the, 1), (the, 1) \cdots\}$ (note the repetition and the fact that every key/value pair has the value 1). Each of these map functions is a very simple function that does only one thing: for each word, generate a key/value pair of value 1. Now there will be a lot of these pairs (as many as there are words in the play). The reduce functions are now going to process these key/value pairs, and in particular each reduce function receives all the key/value pairs having the same key. One reduce function is then going to be in charge of ‘the’ and receives the following list of key/value pairs: $\{(the, 1), (the, 1), (the, 1), \cdots\}$ and will only count how many of these it got. The output of this particular reduce function will be, for instance, (the, 300). That is all map and reduce functions do: simple operations on key/value pairs. They are however very powerful at processing very large datasets, as we will see in this project.

Figure 1 shows you an example with 4 sentences, 2 map functions and some reduce functions.

Now, there is obviously a lot that is being done in the background to organise all these processes. Again, the programs that make the map and reduce functions are very simple. But they are executed as a multitude of processes, that have to be started (how many? when? where?) and scheduled (when? where? on what data?) and managed (what data can they access? is there concurrency/synchronisation issues?) etc. All these concepts are... operating systems concepts :)

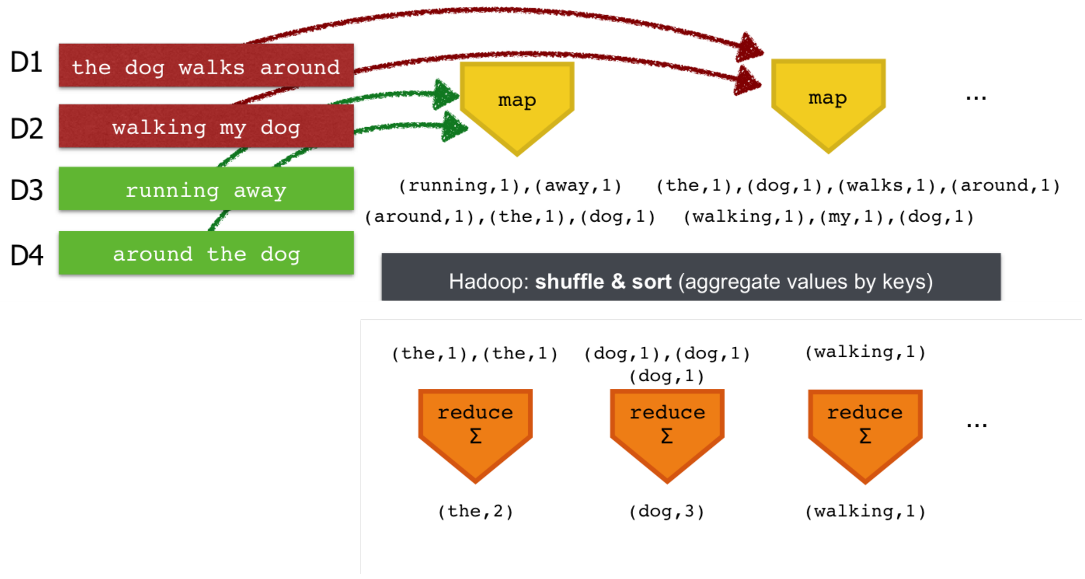

Figure 1: Example of a MapReduce process on a small text made of 4 sentences. Map functions are represented by yellow boxes, reduce functions by orange boxes. Note the input/output of each of these functions, and in particular the repetitions after the map functions and the clustering of input for the reduce functions.

In particular, there is a program/process called **job manager** that will be in charge of each machine in a MapReduce cluster and will start the map and reduce functions, check that they are not having any problems (crash, performance issues) and sends back intermediary results to the main MapReduce master node. The **master node** is the global coordinator of each task. It is in charge of the job managers during the execution of the MapReduce task. The master node is made aware of all the keys generated by map functions and decides which reduce functions will process them etc. Figure 2 gives an overview of the architecture of our MapReduce engine.

2 A simple example

In this example, we want to count the number of times each one of two products has been bought from a list of transactions. Download the files SalesJan2009aa to SalesJan2009ae from Moodle. Create a repository `first_example` in your machine or on the server and move the files in this repository. Each of these files contains lines (tuples) that follow a simple structure:
$< Transaction\_date, Product, Price, Payment\_Type, Name, City, State, Country, ccount\_Created, Last\_Login, Latitude, Longitude >$
Read the files and try to understand what they contain and how each line contains a transaction (a product has been bought).

Now the two functions that we are interested in, map and reduce, will be quite simple:

- the map functions (one per file) will read one file and split each line into one key/-value pair containing the product number and a value (always 1). For instance

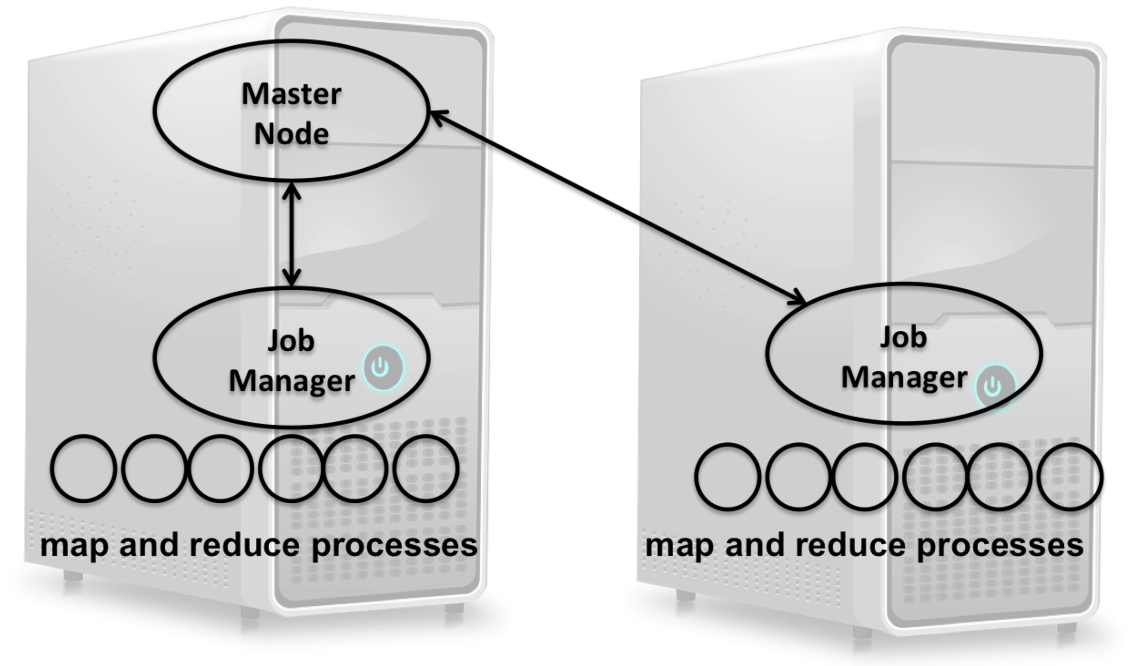

Figure 2: Overview of our architecture: one job manager per machine is in charge of the map and reduce processes, while one single master node manages the overall engine.

$\{(Product1, 1), (Product1, 1), (Product2, 1), \cdots\}$

- the reduce functions (2) will then receive all the pairs corresponding to one single key (one reduce function will receive the pairs with the keys "Product1" and one reduce function will receive the pairs with the keys "Product2") and will sum the values.

The output will be the number of Product1 transactions and the number of Product2 transactions. But we need to set up the whole system to be able to run these two types of functions.

3 Implementation of our single node MapReduce engine

3.1 Job Master

The Job Master is a program (process) running on each machine, and in charge of the MapReduce processes running on the machine.

1. The first thing your Job Master has to do it to figure out how many map functions need to run. In our case, we can assume that this corresponds to the number of files on the machine. So create a script `job_master.sh` that reads a given repository (our `first_example` repository) and counts the number of files in it. This gives you the number of map functions to launch.

2. eventually your Job Master is going to start processes (map and reduce functions) and will certainly need to communicate with them. Create two named pipes, one for the

communication with the map functions (`map_pipe`) and one for the communication with the reduce functions (`reduce_pipe`).

3. when the map functions process their files, they have to let the Job Master know what keys they've come across, and they will use the pipe for that. When the Job Master receives a key in the `map_pipe`, it has to keep track of it in some way. For instance, you could just write it in a file, `keys`, which will contain a list of unique keys. Write the code in the `job_master.sh` script that can read from the pipe and write in the `keys` file if the key is not already in it. Test it by sending some random input in the pipe and check what the file contains.

4. Now the Job Master cannot wait for ever, reading from the (`map_pipe`). The Job Master needs to know when it is safe to stop listening from the pipe and move on to starting the reduce functions. One simple solution is to make sure all the map processes send a specific message in the pipe when they are finished. For instance something like "map finished". If the Job Master creates 5 map processes, it knows it has to receive 5 of such messages before moving on. Write a while loop with one condition: the value of a variable (initialised with the value 0) is less than the number of map functions you have created in the first step of the Job Master (5 in our case). In the loop, read from the pipe (`map_pipe`) (and write in the `keys` file) and if the content of the pipe is "map finished", then increment the value of the variable by 1. When the maximum value is reached, you'll exit the loop.

5. At this point your Job Master script can figure out how many map functions to start (and will be able to start them), can listen what map processes send on to a named pipe and knows when to stop listening (when it has received a message as many times as there are map processes).

6. Next, your Job Master reads from the `keys` file how many keys there are and starts as many reduce functions. It reads from the `reduce_pipe` and waits to receive as many messages, which will indicate that all the reduce functions have finished. Each message received from the pipe should be in the form "key value" and the Job Master can just print it in the terminal at this stage

Try and test your `job_master.sh` script. Indicate the tests you've done in your report.

3.2 Map Functions

1. create a `map.sh` script that takes one argument (a file name) and reads a file following the format given in the previous section. In particular, you want to get the content of the field *Product*. Think `cut`.

2. for each of the values you get (one per line in the file), write a key/value pair in a file with the name of the key. In our example, the files will be `Product1` and `Product2`. So if your map script reads a line with Product1 in the field 'Product', just add one line to the file `Product1` (if it exists; otherwise first create it) with the following format:

```
-bash-4.2$ less Product1
Product1 1
Product1 1
...
```

3. you will have multiple map functions running in parallel, so you need to protect the files (think semaphores may be?).
4. after each line, also send a message to the Job Master on the named pipe with the key you've just read.
5. when the map script has finished with the file, send a last message to the Job Master with the message "map finished".

Try and test you map script. When you think it's working ok, modify the `job_master.sh` script to start as many `map.sh` as required (with the correct parameters). Check that the integration works: all the files (`keys`, `Product1` and `Product2`) are created, there is no concurrency issues, the `job_master.sh` knows when all the map function are finished etc.

3.3 Reduce Functions

The reduce processes read the intermediary files (`Product1` and `Product2` in our example) and count the number of items in them.

1. create a `reduce.sh` script that takes one argument (a key, which is also a file name) and reads this file/count the number of items in it
2. sends this number to the Job Master using the correct named pipe and the format "key value" – for instance, the reduce script in charge of key "Product1" will send a message like "Product1 345" (if there are 345 Product1).

That's it :) Try and test your reduce script. When you think it's working ok, modify the `job_master.sh` script to start as many reduce processes as there are keys (2 in our example). Check that the integration works: the Job Master receives the messages.

3.4 A complete scenario

At this stage, you should have a proper MapReduce system running (on a single node). Test your system (e.g., change the number of input files, add different product names: Product3, Product4, etc. and check that everything works.

4 Advanced Scenario and Analysis

Now, run a couple of more 'complex' scenarios on the same data: can you check other fields than the product ids? the country? the state? Describe in the report which one you chose and how you modified the map and reduce scripts.

Also, can you analyse the impact of distributing the data? Download the single file `SalesJan2009.csv` and run the same map reduce job. This time (there is only one file) there will be only 1 map process. Can you measure the time taken by the MapReduce task in this case? How bad is it compared to running 5 map processes (as in the previous section)?

5 A fully distributed version of our MapReduce engine

Now most of you have two ways to play with Linux: on your machines and on the server. If you do not have Linux on your own machine, please tell either Ellen or Anthony.

What we want to do now is to create a distributed version of the MapReduce engine that you've created. We want the two machines to run the same MapReduce scenario over 2 machines. We need an extra process for that, the Master Node.

5.1 Master Node

1. create a script `master.sh` that creates two networking pipes (with the other machine, one in and one out) and two named pipes (local to the machine where the script is running, one in and one out).

2. change the Job Managers to wait for instructions from the Node Manager (the instructions are probably only going to be a repository name). The first thing the Master Node does is to send the file names to the Job Managers (they then start their map processes)

3. The master node will receive from the two Job Masters (one local and one distant) the list of keys when the map processes are finished in each machine. These keys will be sent (one at a time in either the networking pipe or the named pipe) to the Master Node which then saves them in a file `global_keys`. When all the keys are sent, the Job Managers send a message "job masters finished" and the Master node knows they are finished.

4. the Master Node can then read the `global_keys` file and pick half of the keys for one of the Job Master (e.g., the local one) and the other half for another Job Master (e.g., the distant one). The Master Node then sends these keys to the Job Masters (distant and local) using their own pipes (named pipe for the local node and networking pipe for the distant one). At this stage the Job Masters knows which keys their reduce functions have to process.

5. the problem is that they have some key/value pairs files that the other Job Master needs (the ones corresponding to the keys they have to process). You have to swap these files between the distant and the local machines now - to make sure the Job Masters have their right key/value pairs locally on their systems. Try to figure out a solution to send (over a networking pipe?) a file, probably line by line.

6. Eventually the Job Masters run their reduce processes and send the outputs of their reduce processes to the Master Node which receives them and displays them.

Compare the centralised (only one machine) and distributed versions of your engine. Do you see a difference in performance? Can you measure the overhead of distributing the processing?