



School of Computer Science

OPERATING SYSTEMS

COMP30640

MAPREDUCE ENGINE IN BASH

I declare that this submission is my own work.

Elena Lanigan
17205702

Table of Contents

Section	Page
Cover Page	1
Table of Contents	2
Introduction	3
Requirements	4
Architecture/Design	5-7
Challenges	8
Conclusion	9

Introduction

MapReduce is a programming model which provides analytical capabilities for analysing huge volumes of complex data. In other words, it processes Big Data in parallel on multiple nodes. MapReduce decomposes work into small parallelised map and reduce tasks which are scheduled for remote execution on slave nodes.

The MapReduce algorithm contains two important tasks: Map and Reduce. The Map task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key-value pairs). The Reduce task takes the output from the Map as an input and combines those data tuples (key-value pairs) into a smaller set of tuples. The reduce task is always performed after the map job. The Job Master is responsible for organising where computational work should be scheduled between the mapper and reducer.

There were three parts to this project. The first part involved building a Single Node MapReduce (SNMR), the second was to implement the SNMR in an advanced scenario with analysis, and the third was to create a fully distributed version of the MapReduce engine. The aim of the SNMR is to count the number of times a product (for example Product1, Product2, Product3) has been bought from a list of transactions with the files SalesJan2009aa to SalesJan2009ae located in the files_ folder. The aim of the advanced scenario with analysis is to go beyond the requirements of part one, count other tuples (i.e. Payment Type, Product ID) and describe how the map and reduce scripts were modified. The third part involves creating a distributed version of the MapReduce engine. This allows two machines to run the same MapReduce scenario over two machines which involved creating an extra process called the Master Node.

Requirements

Part 1: SNMR

Job Master (JM): The JM is in charge of the MapReduce process when it is executed on a machine. It figures out how many map functions it needs to launch by counting the number of files in the given repository files_folder. The map function then processes each file in separate functions. Once the map function has finished processing these files, the JM can then call the reduce function. The JM communicates with the map function and reduce function using pipes (map_pipe and reduce_pipe). Once the map functions have finished, it sends the output results to the map_pipe which is received by the JM. The JM then creates a keys file to store these results and send them to the reduce function using the reduce_pipe. The JM then outputs the result of the reduce function.

Map Function (MF): MF takes in the files sent by JM, processes the data depending on the instructions given and outputs a processed result in the form of a key/value pair <key, value>. In this case, the instructions were to select the product tuple from a series of given files. The output of the results from this field are then input into separate files for each product (see example Product1, Product 2, Product 3 in other_folder).

Reduce Function (RF): The RF then processes the key/value pairs processed by the MFs and outputs the final result. In other words, it counts the number of values for its paired key <key, total values>.

Semaphores: Semaphores are scripts used to solve critical section issues and achieve process synchronisation in the multi-processing environment within the MapReduce engine.

Part 2: Advanced Scenario and Analysis

This part involved running more complex scenarios on the same data. For example, to find the Payment Type or Country within the given files. To find Payment_Type, the Map Function was modified by changing the instructions in the mapfiles variable. Other forms of analysis included examining the impact of distributing the data. For example, measuring and comparing the time taken by the MapReduce task when it is given one single file instead of multiple files.

Part 3: A fully distributed version of the MapReduce engine

The third part involves creating a distributed version of the MapReduce engine. This allows two machines to run the same MapReduce scenario over two machines which involved creating an extra process called the Master Node.

Architecture/Design

Part 1: SNMR

Job_master.sh: This script implements the functions of the JM as described above. A path to a given repository (`files_folder`) was made into a variable, `dir_name`, which contained a set of files. A for loop was created to count the number of files in `dir_name` and for each file, the `map.sh` function was executed. To allow the map and reduce functions to communicate with the JM, two pipes, `map_pipe` and `reduce_pipe`, were created using `mkfifo`. A while loop was created to read the results of the map function from the `map_pipe`. When the results equalled "map finished", the results are redirected into a `keys.txt` (if it exists, else append it) to keep track of the results from the `map_pipe` (as shown in Figure 1). Within the while loop, there was a count (`finish_count`) which incremented once it received the message "map finished". This message signals to the JM that it is safe to stop listening from the pipe and move on to starting the reduce functions. Once the count of this while loop equalled the number of files in the repository, the while loop finished. The JM then implemented the reduce function similar to the map function as above.

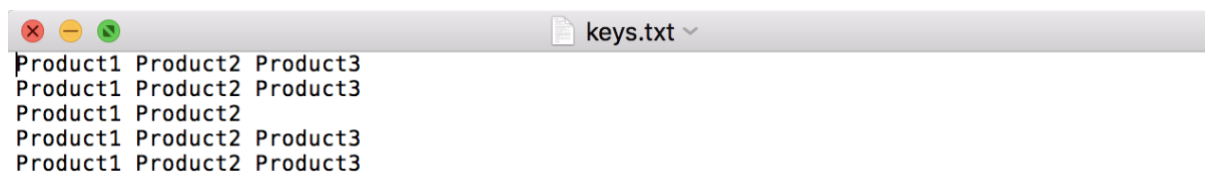


Figure 1: `keys.txt` file containing unique products sent by MF

Map.sh: The script contains two variables: `mapfiles` and `key`. `Mapfiles` contains the instructions on which tuple to cut on each file. In this case, the product tuple. The `key` variable then sorts these files uniquely. Figure 2 shows a for loop which is used to loop through each line in `mapfiles` and for every line, inputs a key/value pair into a new unique key file if it is not created (in this case the files are called Product 1, Product 2 and Product3). The key is then redirected to the `map_pipe` pipe, and after 1 second, "map_finished" is directed into the map to tell JM it has finished working on a file.

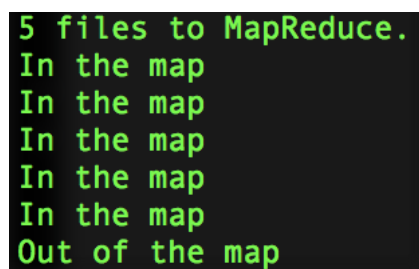


Figure 2: `map.sh` looping through 5 files

Reduce.sh: This script reads the files (Product1, Product2 and Product3 in this example) and counts the number of items in them. It then sends this number to the Job Master using the `reduce_pipe` in the format `<key, total values>`. As shown in Figure 3, only Product1 would correctly be sent to the JM. Figure 4 shows the reduce function working correctly when used on files Product1, Product2 and Product3.

```
Mapper finished. Time to reduce
Product1 847
Product1 847
```

Figure 3: reduce output in JM

```
first_example — -bash — 81x7
Elenas-MacBook-Pro:first_example elenalanigan$ ./reduce.sh Product1
Product1 847
Elenas-MacBook-Pro:first_example elenalanigan$ ./reduce.sh Product2
Product2 136
Elenas-MacBook-Pro:first_example elenalanigan$ ./reduce.sh Product3
Product3 15
Elenas-MacBook-Pro:first_example elenalanigan$ █
```

Figure 4: reduce output when only using reduce.sh.

Semaphores: Three sets of semaphores were created to be implemented within the JM, MF and RF. Two sets were created to protect the two critical sections (CS) in the map function and one set was to protect the CS within the reduce function. Within the two scripts, they are marked out as the JM script would not run if they were marked in (see figure 5 and 6 for examples).

```
# ./p_reduce.sh reduce_pipe
echo "$1 $count" > reduce_pipe
sleep 1
echo "reduce finished" > reduce_pipe
# ./v_reduce.sh reduce_pipe
```

Figure 5: reduce semaphore marked out

```
./p_reduce.sh reduce_pipe
echo "$1 $count" > reduce_pipe
sleep 1
echo "reduce finished" > reduce_pipe
./v_reduce.sh reduce_pipe
```

Figure 6: reduce semaphore marked in

Part 2: Advanced Scenario and Analysis

To test that the JM, MF and RF in a different scenario, the instructions were changed within the MF to the mapfiles variable as shown in Figure 7. This command parses an element with the cat command and using ',' as a delimiter to separate fields, prints the field using -f. For example, -f2 instructed the program to cut the second field followed by a ',' which was used for the SNMP. By using -f4, when the files were sent to the job master to be mapped and reduced, they output the key/value pairs for Payment_Type (see advanced_scenario folder).

```
# variable to find
mapfiles=`cat $1 | cut -d',' -f4`
# variable to sort
key=`cat $1 | cut -d',' -f4 | sort -u`
# echo $key
```

Figure 7: reduce semaphore marked out

As the JM functionality did not work correctly, I was unable to analyse the impact of distributing the data.

Part 3: A fully distributed version of the MapReduce engine

As the JM functionality did not work correctly, I was unable to analyse the impact of distributing the data across two networks. This would have been implemented using a VM, or alternatively, with a colleague.

Challenges

Semaphores and Pipes:

The main problem with the JM, MF and RF functions revolved around semaphores and pipes. The semaphores were changed around in multiple ways; however, the JM would not run whenever the semaphores were marked into the code. This seemed to cause most of the errors in the JM function including:

```
./job_master.sh: line 52: reduce_pipe: Interrupted system call
./reduce.sh: line 7: $1: ambiguous redirect
```

A series of echos were implemented throughout the script to find where the problem may have been but this did not resolve the problem.

Job Master:

In the early stages of writing the JM script, `ls -l | wc -l` was used to count the number of files in a directory (as shown in Figure 8). A while loop was then made to communicate with the map_pipe using a series of if and elif statements. From speaking with tutorial assistants and an extensive search of stackoverflow.com for answers, the current implementation of JM is more efficient (and works to some extent). To test the current JM script to ensure the MF results being redirected into a new file, I added changed two instances of Product 1 to Product4 in SalesJan2009aa (see Figure 9). This showed the correct results were being communicated between JM and MF with the new script.

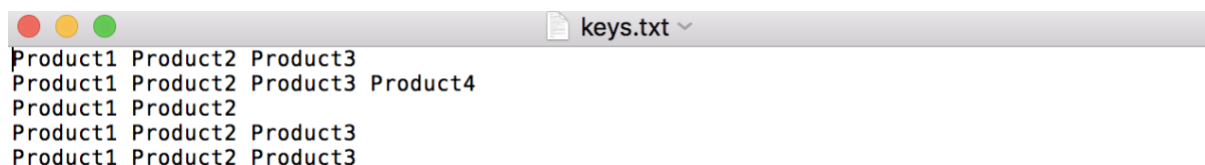
```
count files in a directory
filecount= `ls -l | wc -l`

echo $filecount

#created - mkfifo - map_pipe
#created - mkfifo - reduce_pipe

count=0
while [ "$count" -lt $filecount ]; do
    read input_pipe < map_pipe;
    echo $input_pipe
    #checks keys.txt for input
    if grep -qWF "$input_pipe" >> keys.txt; then
        echo "We found something"
    elif $input_pipe = $MP; then
        ((count++))
    elif $input_pipe = $MP; then
        ((count++))
    elif $input_pipe = $MP; then
        ((count++))
    fi
done
```

Figure 8: first JM attempt



```
Product1 Product2 Product3
Product1 Product2 Product3 Product4
Product1 Product2
Product1 Product2 Product3
Product1 Product2 Product3
```

Figure 9: product4 addition to test JM and MF

Conclusion

The above report illustrates an all most complete SNMR engine made using Bash. There are probably only one or two changes which are needed within the syntax of the three scripts for the MapReduce to work efficiently so I am looking forward to seeing a fully completed solution. I also enjoyed working on this project collaboratively and sharing ideas (and frustrations) with my colleagues.