Programming for Big Data

Assignment 2 - Hadoop

D10126532 John Warde DT230B

Note: the word “actor” is used throughout this document (and in the included code) to describe the “character” names in a play - because we also talk about “text characters” in parsing tasks of this nature “actor” is used to remove ambugity.

1. Hadoop Job Design, Code and Execution for Co-Appearance Network

The task is to generate a co-appearance network for a Shakespeare play which will show what actors appear in a play together and the strength of that relationship.

The chosen play is “ROMEO AND JULIET” by William Shakespeare. The first port of call was to review content structure of the RomeoJuliet.txt file to identify repeating patterns in order to isolate the actor names in the file. The actor names (as they speak their lines) start at the first column of a line and then the name is followed by a tab character; this is consistent throughout the file. The following Regular Expression is used to isolate the actor name in a line.

```
^([A-Z ]+?)\t
```

The approach taken in general was:

- Mapper: for each text line that was fed to the map() call:
  - Attempt to match for an actor name, if not get out of the map() call
  - Is the matched actor name within N number of lines of the last matched actor?
  - If yes output a pairing of the last actor name and this actor name, with a count of 1
- Sort & Shuffle will order the relationships and send a list of 1s repeated for each identical relationship.
- Reducer: for each relationship pair and list of occurrences sent to the reduce() call:
  - For the output key concatenate the two actor names and the word “undirected” all separated by a tab character and the total count for the value
  - Total up the values in the associated list and use this as the key value (weight)

The following is a snapshot of sample data as it passes through the Hadoop Job, where actor names are paired if they are within 5 lines of each other:

Input to Mapper	Output from Mapper
(145, "CAPULET What noise is this? Give me my long sword, ho!")	
(285, "")	
(293, "LADY CAPULET A crutch, a crutch! why call you for a sword? ")	("CAPULET|LADY CAPULET",1)
(353, "")	
(560, "CAPULET My sword, I say! Old Montague is come, ")	("CAPULET|LADY CAPULET",1)
(599, " And flourishes his blade in spite of me. ")	
(655, "")	
(660, "[Enter MONTAGUE and LADY MONTAGUE] ")	
(798, "")	
(800 "MONTAGUE Thou villain Capulet,--Hold me not, let me go. ")	("CAPULET| MONTAGUE", 1)
(845, "")	
(849, "LADY MONTAGUE Thou shalt not stir a foot to seek a foe. ")	("LADY MONTAGUE|MONTAGUE",1)
(912, "")	
(915, " [Enter PRINCE, with Attendants] ")	
(967, "")	
(970, "PRINCE Rebellious subjects, enemies to peace, ")	("LADY MONTAGUE|PRINCE",1)

You will notice that the pairing is sorted keeping the lexicographically less actor name to the left – this is done to count the relationship occurrences correctly e.g. the relationship pairing "CAPULET| MONTAGUE" is identical to "MONTAGUE |CAPULET" i.e. commutative.

Input to Reducer	Output from Reducer			
("CAPULET|LADY CAPULET|", [1,1])	CAPULET	LADY CAPULET	undirected	2
("CAPULET|MONTAGUE", [1])	CAPULET	MONTAGUE	undirected	1
("LADY MONTAGUE|MONTAGUE",[1])	LADY MONTAGUE	MONTAGUE	undirected	1
("LADY MONTAGUE|PRINCE",[1])	LADY MONTAGUE	PRINCE	undirected	1

To determine the number of lines between each actor name a line counter is maintained at the class level in the Mapper class (ActorMapper.lMapperLineNumber).

This is incremented for each call to map() (once per line of text). The last matched actor name and the line number it was found on, are also recorded to determine if a pairing will be created between the current matched actor name and the last matched actor name.

One instance of the ActorMapper class is instantiated for each mapper that the Hadoop system creates to process the input file and this allows us to maintain the said variables between map() calls.

As with the nature of the Hadoop system, we will inevitably miss some relationships when Hadoop splits up the file into the physical Hadoop Distributed File System (HDFS) blocks. As only a small number of relationships will be missed, in relation to the entire file, it is deemed that this will have little effect on the overall result.

The majority of the processing for creating the co-appearance network happens in the ActorMapper class, so particular attention was placed on this, to make it more efficient:

- Compiled the regular expression pattern once per created Mapper task, this was done in the configure() call.

- The map() call was optimised by getting out of the call straight away (by using the "return" keyword) when no more processing needed to be done for that particular call to map(). You will see a "return" keywords for each of these situations:
    - When a blank line is passed to map()
    - When the matched actor name contains the word "SCENE"
    - When recording the first occurrence of an actor name.
    - When the same actor name is found as the last actor name matched.

As per the requirement "that the characters names appear within a certain number of lines of each other", we need to have the number of lines configurable. To that end Hadoops's ToolRunner class is used on the name node to process the command line options.

The following are all the configurable parameters this Generate Co-Appearance Hadoop Job uses:

- "**match-within-n-lines**", optional – if not present then the number of lines is infinite.
- "**sort-pairs**", default is true when not present, this allow the relationships to be counted correctly
- "**use-combiner**", default is true. The combiner is used after the mapper task is complete and is a sort of a mini-reducer, the same reducer code (PairReducer.class) is used to run the combiner (before the shuffle and sort phase) on the same task node as the mapper to reduce the work that reducer needs to do. This also the effect of optimising the overall Hadoop job.
- "**match-minor-actors**", default false. An example of a minor actor is "Nurse" where the actor name also contains lower case characters. The major actor names are all in uppercase.

Warning messages are sent to the Hadoop logs to help in identifying possible bugs in the code, in fact the "match-minor-actors" parameter was introduced after the logs began reporting that the same actor name was appearing one after another. After reviewing the logs it was determined that minor actor names, such as "Nurse" were getting ignored because they weren't all uppercase. So the regular expression gets changed slightly depending on the true or false value of the "match-minor-actors" parameter.

To further alert the operator of possible parsing errors, Hadoop's built-in counter mechanism is use to counter the number of actor names appearing one after another. This information is retrieved back on the name node to alert the operator of possible problems.

There was one strange bug that has not been resolved: the content of the output CSV file had "undirected" appearing twice on each line, the duplicated word does not appear in the logs. The workaround was to write a shell script (post-process.sh, included) to correct this and add a header record at the top of the file, this was done using Linux's streaming editor (sed).

Finally, the included Gephi file and image was generated with the following command lines:

```
hadoop jar GenerateCoAppearanceNetwork.jar \
literaryanalysis.GenerateCoAppearanceNetwork –D match-within-n-lines=15 \
plays/RomeoJuliet.txt gen29
./post-process.sh gen29 ~/OnPC/Hadoop
```

2. Hadoop Job Design for Multi-Play Processing

Having reviewed all the files contained in the plays.zip compressed folder, the content structure of each play is identical to the chosen play (RomeoJuliet.txt) in task 1 above.

In particular the actor names are positioned in identical locations i.e. at the start of the line and a tab character follows the name. Therefore the same parsing logic can be used to process all the plays.

Also, the play files are independent of each other and we want a separate output file for each play. Therefore we can run the processes in parallel.

Hadoop started out life by processing jobs sequentially i.e. another job could not start until the current job has completed (even though that job may not be using all nodes in the cluster), it operated in a first in first out (FIFO) fashion.

In 2008, a new pluggable scheduler class (JobScheduler) was created (independent of the Job Tracker) and allows different scheduling algorithms to be used.

By enabling a job scheduler in the Hadoop configuration we can process multiple files by cycling through all the files contained in a folder and starting separate Hadoop jobs in the **background**.

Add the following to the conf/core-site.xml configuration file:

```
<property>
  <name>mapred.jobtracker.taskScheduler</name>
  <value>org.apache.hadoop.mapred.CapacityTaskScheduler</value>
  <description>Use the Capacity Scheduler</description>
</property>
```

And restart the Hadoop cluster.

The script below is a sample bash script to send the individual play files to Hadoop's HDFS and then starts a Hadoop job in the background by using the ampersand symbol which will return almost immediately then we can loop around a to put the next play file into HDFS and so on.

Note: the script below is also included with this submission. Also, I ran this script on my VM and it ran successfully without modifying conf/core-site.xml configuration file as mentioned above.

```
SRCDIR=$1
JARDIR=~/workspace/D10126532/bin
HDFSIN=/user/soc/autoplaysin
HDFSOUT=/user/soc/autoplaysout

FILES=$SRCDIR/*
for f in $FILES
do
  FILENAME=`basename $f .txt`
  echo "Processing $FILENAME play ..."
  hadoop fs -put $SRCDIR/$FILENAME.txt $HDFSIN
  hadoop jar $JARDIR/GenerateCoAppearanceNetwork.jar \
         literaryanalysis.GenerateCoAppearanceNetwork \
         -D match-within-n-lines=15 \
         $HDFSIN/$FILENAME.txt \
         $HDFSOUT/$FILENAME.csv &
done
```

After all the jobs have completed then the post-process.sh script should be executed on each file to ensure successful importing into Gephi.