Big Data Report - Duncan Milne - 2087186m

Command to run:
java-run.sh Main 2005-12-06T17:44:47Z 2008-01-05T14:48:05Z 50
First two parameters are the timestamps, third parameter specifies k.

Results found:
I ran the above command 5 times, the table below illustrates the results found:

Task name	Query Processing Time	Number of bytes read from hdfs	Number of bytes transferred over the network
First run	13 mins, 11 seconds	31274123655	119017508
Second run	9 mins, 48 seconds	31274123655	119017508
Third run	6 mins, 55 seconds	31274123655	119017508
Fourth run	6 mins, 39 seconds	31274123655	119017508
Mean	9 mins 8 seconds	31274123655	119017508
Standard Deviation	5 mins, 17 seconds	0	0

I then ran the same command with k being 500 which produced the following results

Task name	Query Processing Time	Number of bytes read from hdfs	Number of bytes transferred over the network
First run	5 mins, 53 sec	31274123655	119017508
Second run	3 mins, 54 sec	31274123655	119017508
Third run	3 mins, 31 sec	31274123655	119017508
Fourth run	2 mins, 33 sec	31274123655	119017508
Mean	3 mins 58 secs	31274123655	119017508
Standard Deviation	1 minute 13 seconds	0	0

Scalability:
Surprisingly, the jobs involving a larger value for k were completed much faster than those with the smaller value for k. From this I can assert that runtime is not very dependent on the size of k, external variables such as traffic on the network will have a much larger effect on the query processing time.

Design Decisions:
The main class is where the job is initiated. The class takes three arguments, the two timestamps as well as the number of entries to be displayed at the end. Various configuration variables are then set, this gives us the ability to access the command line arguments in the mapper class.

I use the LineRecordReader class with “/n/n” as the separator to split up the records. This is set in my fileInputFormat file by simply returning a LineRecordReader object with “/n/n” as the argument to the constructor.

In the myMapper class, the record is split into an array with a space as the separator. This allows simple extraction of the article ID as well as the date. if the date is between the dates set in the command line the mapper will then write the to the context with the article ID as the key and 1 as the value, as one revision for the article has been found.

The combiner will group together revisions with the same article ID, sum up the value and pass them to the reducer, this will reduce work required in the single reducer.

In the myReducer class the number of revisions of a certain article will be calculated and then added to the priority queue. If there are above k values in the priority queue, the priority queue will spit out the entry with the lowest number of revisions. The reducer also has a cleanup function which will write the top ten revised articles to the context, this is how we get the result of the map reduce.

I decided to use a single reducer with a priority queue, this was because in conjunction with a combiner, the reducer has very little work to do and allows us to only have one job, simplifying the process and removing the need for post processing. I decided this was a worthwhile trade off because the bottleneck occurs in the mapper as opposed to the reducer, meaning speeding up the reducer would not make a significant impact on the runtime of the program.