Big Data 4

Assessed Exercise 1 Report

Team C: Mircea Iordache [1106729], Vlad Iulian Schnakovszki [1106695]

Note: In the code provided, for Task1.java, Task 2.java, Task3.java you will need to add the following line:

job.getConfiguration().set("mapred.jar", "file://<path/to/built.jar>"); (replace as appropriate)

Design

The design of the tasks is straight-forward, with three task jobs (that run client-side), four mappers and four reducers. We also have a helper class that implements a custom tokenizer, and handles date formatting and sorting of Iterable items given to reducers.

We use two custom key-value classes for tasks 2 and 3, each customised to provide sorting of values during Hadoop's shuffle between mapper and reducer.

Task 2 is split between two separate MapReduce jobs, and the second one has just one reducer to ensure global memory for extracting the top K elements. Optionally, this task could have been done with the output of Task 1.

Optimisation

Optimisation is done in-code through a custom tokenizer and custom key-value pairs that aid in sorting by value. The custom tokenizer is designed to be faster than any of the functionality Java provides. It does all the required processing for a line in a single pass with the minimum amount of operations.

One interesting aspect about the data provided is that not all of it is relevant to the given tasks. Only one out of ~9 lines is relevant, but line length is significantly longer for some unused lines. Thus, we decided to create a truncation job that outputs to HDFS only the relevant information. This eliminates future read time of irrelevant data, and dramatically improves query performance. The total size of the truncated file is 3.27% of the original file.

We decided to use only one reducer for Tasks 1 and 3 because of file merging issues in the printer (The resulting file would not be ordered properly). Thus, results are guaranteed to be ordered as intended.

Challenges and Lessons Learned

The distributed nature of the system posed a challenge in debugging the code, and in identifying the location of faulty code (local or cluster-side).

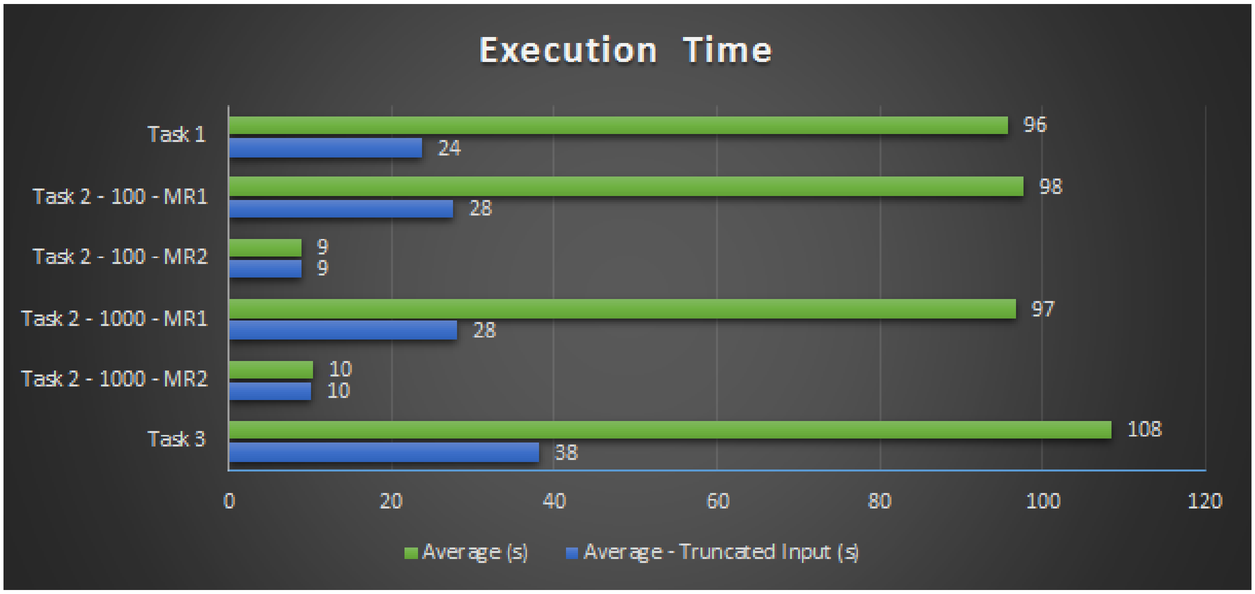
About half of the time for this exercise was spent on researching the sorting of values in the reducer. Issues were also encountered where the titles of the articles contained Japanese characters with spaces (Wikipedia URLs only contain underscores). This problem was solved by using a regex string to extract the date in case it couldn’t be extracted by the custom tokenizer.

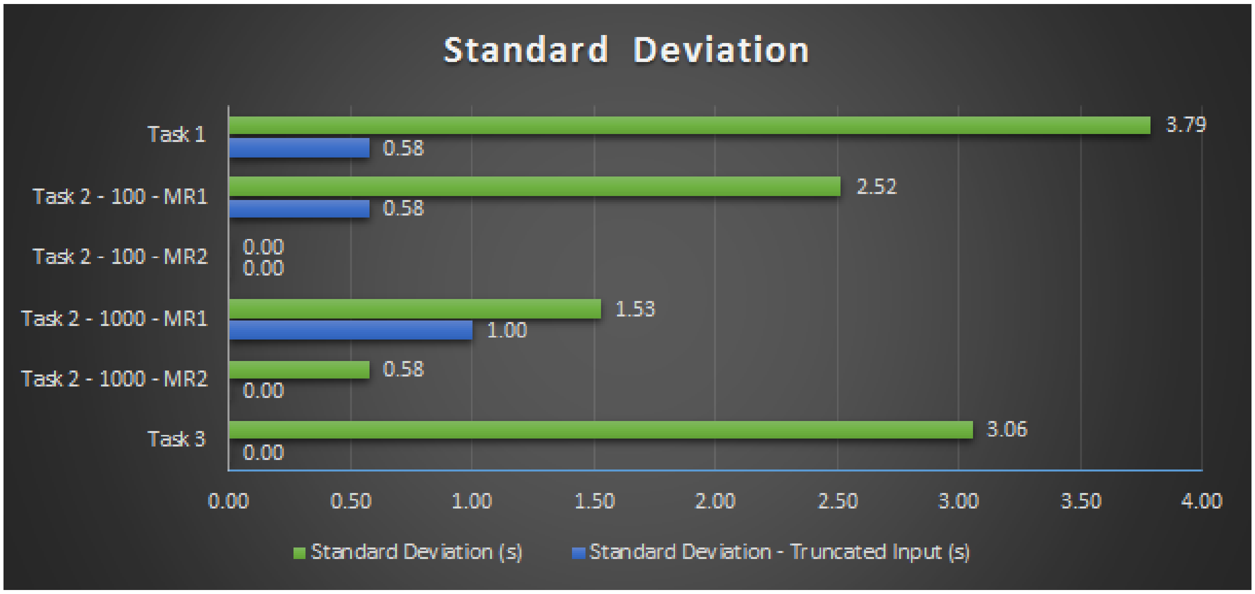
Experimental Results

The code was run on the cluster as per the instructions. The graphs below show our findings for both the original file and the truncated one. Each task was run 3 times.

Execution Time

The execution time on the truncated data set was 3.5 to 4.1 times faster than on the full data set. The second stages for the second task are identical on both data sets because their input is identical. One thing to notice is that the execution time for getting the top 100 records was almost the same as the execution time for getting the top 1000 records in the second task. The execution time for the truncation task was 98 seconds.





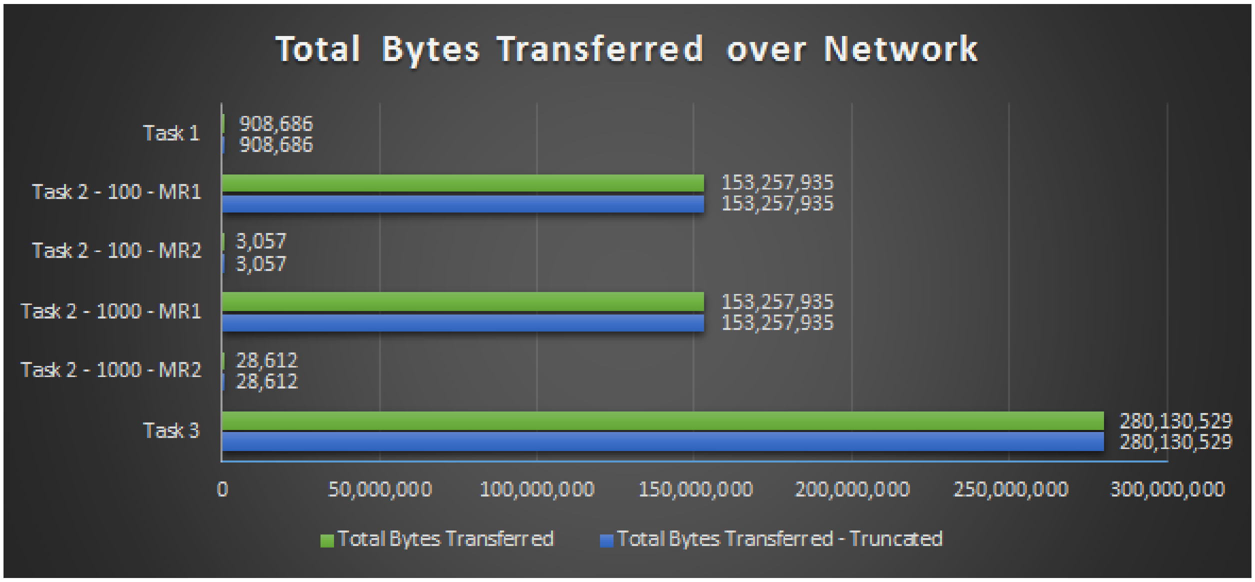
Total Bytes Transferred over the Network

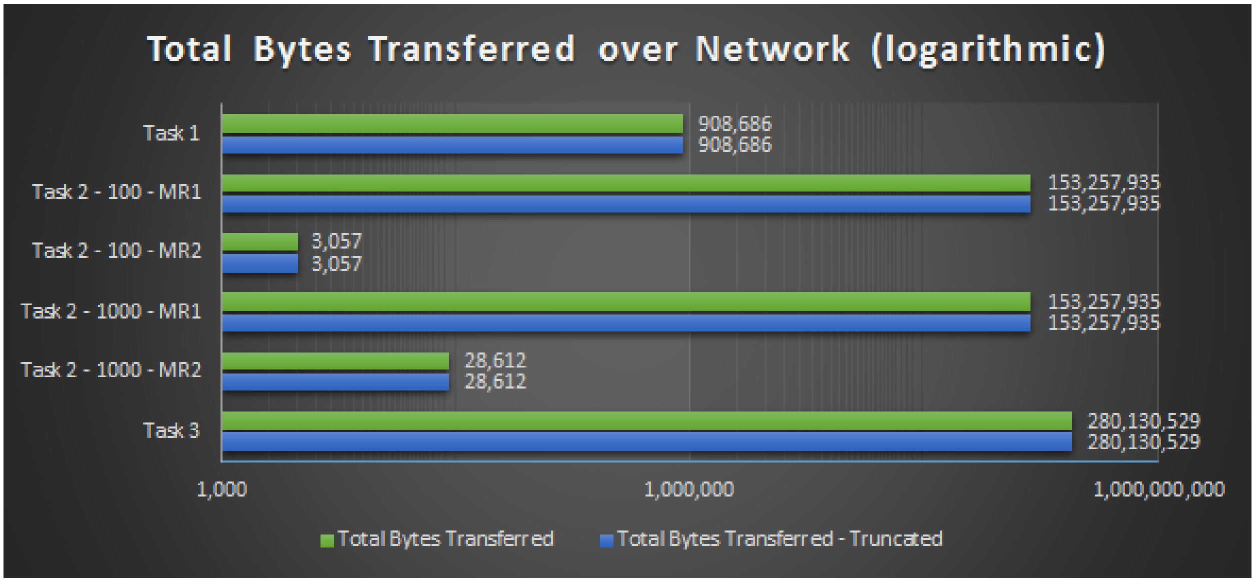
Below you can see the total number of bytes transferred over the network. This is the sum of the map output bytes, the reduce shuffle bytes, and the number of bytes written to the screen.

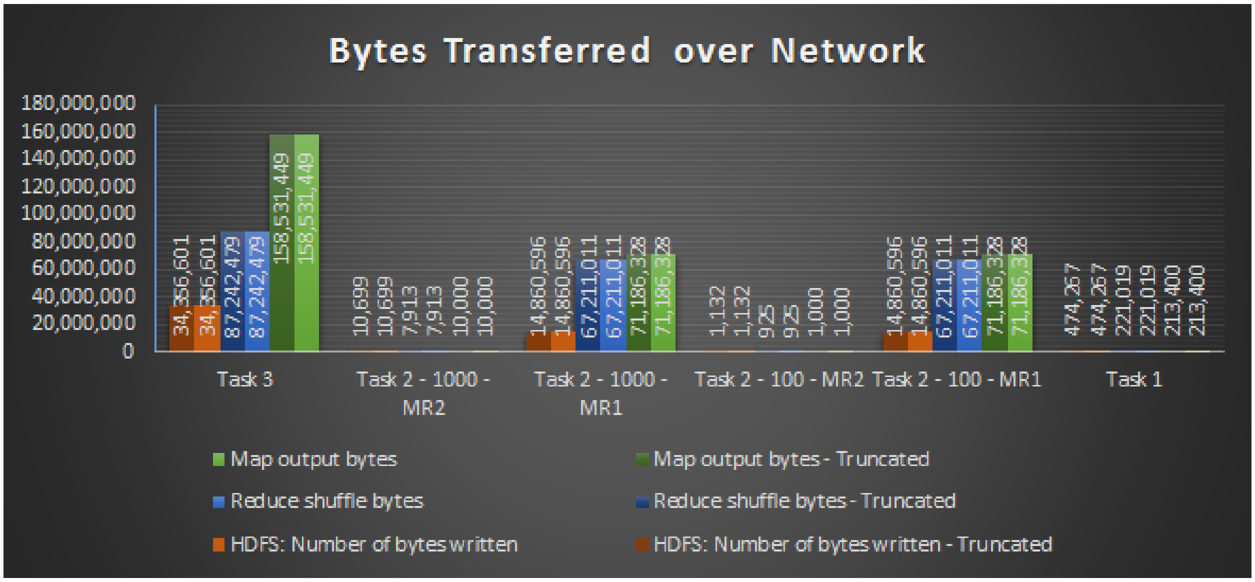
The result of all the test runs were identical, for both datasets. This means that the standard deviation will be 0 for all the tasks. The graphs are also useful as a confirmation that the truncation task produces a valid data set.

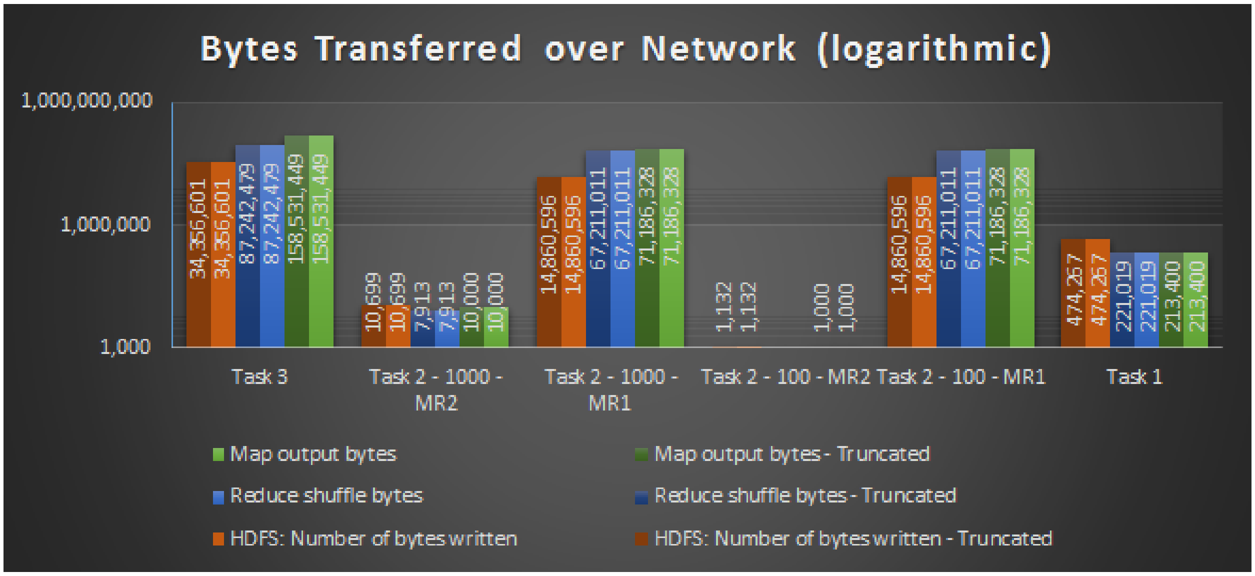
The formula we use is Mapper Output + Reducer Shuffle Bytes + HDFS Written Bytes. We assume that all Mapper Input is data-local (i.e. the mapper is running on the same HDFS node where the corresponding split is).

You can see the details both on a linear and logarithmic scale, as well as the breakdown.




Total Bytes Transferred over Network (logarithmic)
Task 1
908,686
908,686
Task 2 - 100 - MR1
153,257,935
153,257,935
Task 2 - 100 - MR2
3,057
3,057
Task 2 - 1000 - MR1
153,257,935
153,257,935
Task 2 - 1000 - MR2
28,612
28,612
Task 3
280,130,529
280,130,529
1,000
1,000,000
1,000,000,000
🟩 Total Bytes Transferred
🟦 Total Bytes Transferred - Truncated


Bytes Transferred over Network
180,000,000
160,000,000
140,000,000
120,000,000
100,000,000
80,000,000
60,000,000
40,000,000
20,000,000
0
34,356,601
34,356,601
87,242,479
87,242,479
158,531,449
158,531,449
10,699
10,699
7,913
7,913
10,000
10,000
14,860,596
14,860,596
67,211,011
67,211,011
71,186,328
71,186,328
1,132
1,132
925
925
1,000
1,000
14,860,596
14,860,596
67,211,011
67,211,011
71,186,328
71,186,328
474,267
474,267
221,019
221,019
213,400
213,400
Task 3
Task 2 - 1000 - MR2
Task 2 - 1000 - MR1
Task 2 - 100 - MR2
Task 2 - 100 - MR1
Task 1
🟩 Map output bytes
🟩 Map output bytes - Truncated
🟦 Reduce shuffle bytes
🟦 Reduce shuffle bytes - Truncated
🟧 HDFS: Number of bytes written
🟫 HDFS: Number of bytes written - Truncated



Number of Bytes Read from HDFS

The number of bytes read from HDFS was the same for all tasks:

- 31,273,800,537 bytes (29.126 GB) when running on the full data set
- 1,024,541,285 bytes (0.954 GB) when running on the truncated data set

This shows that the truncated data set was just 3.276% of the original data set while preserving all the required information.

The standard deviation is 0 as the input is identical for all task runs.