Big Data 4

Assessed Exercise 2 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

In a similar fashion to the previous exercise, there are three task classes that launch jobs. However, based on feedback received, we have implemented the second task using only one MapReduce job. The helper class exists to get the milliseconds from a timestamp string and allows sorting entries in a Map, used in Task 2 Mapper to only emit the Top K items; we have implemented a file printer class to aid in outputting to screen, for both sequential output files as well as files where the concatenated contents are not ordered. We use a modified version of the Key-Value class from the first exercise to help in storing the top K elements for task 2 within the reducer.

To ensure that the reducers receive all the mapper output for an article id in sorted order, we have a custom partitioner, based on the assumption that every reducer should handle an equal split of the key space.

Optimisation

KeyOnlyFilter

Because HBase is column-oriented and we don't need any of the non-key columns, the first optimisation is to only read the key column, which contains exactly the information we need.

setTimeRange

All the queries limit the records of interest within a time range so only the relevant values are received by the mappers. We could have further optimised this by creating an index on the timestamp, but that would have required a pass through the whole dataset which would have taken longer than reading the data directly. This would be a valuable option if the data is to be analyzed multiple times.

VLongWritable

After analyzing the data set, we observed that the largest article id is 15,071,250, which can fit in 3 bytes, while Java stores a long variable in 8 bytes. We initially set out to find a data structure that would be able to transmit this in a more efficient manner. We first discovered BytesWritable and figured that it is the optimal solution. Soon after though we discovered VLongWritable and decided to use this as it would automatically parse the long values to and from bytes. Using this, we are able to reduce every single long value transmitted from 8 bytes to

3 bytes or less (disregarding the overhead of the actual object), effectively saving a minimum of 5 bytes per value.

Minimal output between mappers and reducers

In order to minimize the amount of information processed by the reducers (and partitioner) and to optimize the network transfer, tasks execute all processing in memory, aggregate the results and only output the minimum necessary information that guarantees the results can be calculated correctly. This also eliminates the need for a combiner, as it executes on pairs received from local mapper output.

Taking advantage of the sorted nature of article id

For all tasks, we take advantage of the sorted nature of the article ids. We aggregate the results for the current article id in memory and output only once, when the article id changes. This is possible due to the fact that HBase returns consecutive revisions for every article.

Reducers

We tried to experimentally discover the optimal value for the number of reducers but the cluster was too unstable to make any meaningful observations. For this reason, we decided to go with the maximum number of reducers available in the cluster (16).

Number of MapReduce Jobs

We have refactored the code for all exercises, and are now using only one job for each query. This is opposed to our first implementation, where for the second task, we chained together two MapReduce jobs and were using a temporary HDFS directory to store intermediate results.

Partitioners

In order to ensure that all mapper output for an article id reaches the same reducer and the article ids are received by the reducers in a sorted sequence, we have implemented a partitioner. This is optimized to send all mapper output with article ids within a range to a certain partitioner. This is done by splitting the maximum value of the article id by the number of reducers, assigning equal key ranges to all of them. This is not perfect however as the article ids are not equally distributed across the key space.

Date as long

For the 3rd task, the dates are only turned into their string representation in the final step of the reducer. When transmitted over the network, they are transmitted as long values as part of a string, which is more efficient than transmitting them as their string representation. An even more optimal solution would have been to create a special data structure implementing writable, which would have stored the revision ids in an array, rather than strings.

Challenges and Lessons Learned

The main issues we encountered were with cluster instability during the allocated time slot. Because of this, we are unable to comment a lot on performance of HBase compared to HDFS. The same query on the same input values had varyingly large execution time differences, sometimes even doubling the execution time.

One of the team members was also unable to access the cluster during the time slot.

We improved our knowledge on how to perform optimisations and we now better understand how their use makes a huge difference in query performance. We believe that filtering input for HBase is easier than doing so for HDFS, requiring only a few lines of code and little background research required to accomplish the task for the NoSQL implementation.

Experimental Results

The code was run on the cluster as per the instructions, using the same parameters used for the first assessed exercise. Each task was run 10 times.

Execution Time

Because of the cluster's instability, the tasks show a large standard deviation, therefore the minimum and maximum execution times for all tasks are included. The aggregated results are detailed below.

Task	Average (s)	Standard Deviation (s)	Minimum (s)	Maximum (s)
Task 1	78	28.27	46	120
Task 2 - 100	103	32.92	62	150
Task 2 - 1000	97	14.86	74	119
Task 3	81	24.21	57	120

Number of Bytes Read (from HDFS/HBase)

The requirement asks for the number of bytes read from HDFS, which in this case is irrelevant as we are reading from HBase. For the sake of completeness, the number of bytes read from HDFS was 7,840 for all tasks, with a standard deviation of 0.

We however consider that the number of bytes read from HBase is relevant, especially for the comparison between using HDFS and HBase. Therefore, you can find them below:

Task	Bytes Read
Task 1	12,560,688
Task 2 - 100	4,095,119,347
Task 2 - 1000	4,095,119,347
Task 3	1,841,184,885

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

Total Bytes Transferred over the Network

Below you can see the total number of bytes transferred over the network. This is the sum of Mapper Output, Reducer Shuffle Bytes and HDFS Written Bytes.

Task	Total Bytes Transferred
Task 1	1,161,185
Task 2 - 100	117,977
Task 2 - 1000	864,356
Task 3	76,534,584

The results of all the test runs were identical, which means that the standard deviation is 0 for all the tasks.

MapReduce over HDFS vs. MapReduce over HBase

Below you can find some metrics comparing the performance of the two runs.

Execution Time

It is easy to see (below) that the cluster was stable when running over HDFS but was unstable when running over HBase from the huge difference in the standard deviation. This means that comparing the averages of the two runs doesn't provide much information. We will assume that when stable, the cluster would be able to produce an average execution time closer to the minimum one, so we use that for comparison as well.

Task	HDFS Average (s)	HBase Average (s)	HBase Minimum (s)	HBase Ave. faster than HDFS Ave.	HBase Min. faster than HDFS Ave.
Task 1	96	78	46	19.17%	52.08%
Task 2 - 100	107	103	62	3.93%	42.06%
Task 2 - 1000	107	97	74	8.97%	30.84%
Task 3	108	81	57	24.72%	47.22%

As you can see from the above, running the same queries on HBase is consistently faster than running them on HDFS. In the optimal conditions, HBase would be able to produce the results at least 30 - 50% faster than HDFS.

Execution Time - Standard Deviation

To demonstrate the stability difference between the runs, the table below shows a comparison of the standard deviations.

Task	HDFS Standard Deviation (s)	HBase Standard Deviation (s)	HBase SD larger than HDFS SD (%)
Task 1	3.79	28.27	745.79%
Task 2 - 100	2.52	32.92	1306.22%
Task 2 - 1000	1.53	14.86	971.00%
Task 3	3.06	24.21	791.25%

Number of Bytes Read

Task	HDFS	HBase	HBase / HDFS
Task 1	31,273,800,537	12,560,688	0.04%
Task 2 - 100	31,273,800,537	4,095,119,347	13.09%
Task 2 - 1000	31,273,800,537	4,095,119,347	13.09%
Task 3	31,273,800,537	1,841,184,885	5.89%

Because in HBase we were able to only read the records we needed, without having to read the entire data set, we managed to dramatically reduce the amount of data read.

Total Bytes Transferred over the Network

Task	HDFS	HBase	HBase / HDFS
Task 1	908,686	1,161,185	127.79%
Task 2 - 100	153,260,992	117,977	0.08%
Task 2 - 1000	153,286,547	864,356	0.56%
Task 3	280,130,529	76,534,584	27.32%

As can be seen from the above, there are massive differences between the two runs. This can be attributed to our naive implementation of the first exercise, as an optimal solution should be able to achieve equal values regardless of whether the data is read from HDFS or HBase.