Mika Huttunen

013879011

1.

Result sizes (= that of generated part-r-00000 files) for each score and student file combination. Notice that these are in different order than as in Questions material.

|  |  |
| --- | --- |
| Data combination | Result size |
| “score\_3000.txt” and “student\_30000000.txt” | 164 bytes |
| “score\_3000000.txt” and “student\_30000000.txt” | 114246 bytes |
| “score\_30000000.txt” and “student\_30000000.txt” | 1681827 bytes |

Below you can find the running times for each program and data combination.

Program (1) stands for “generic” reduce-side join solution. Program (2) is using

distributed cache for storing the score data, and does joining on map side.

Program (3) uses bloom filter to reduce the amount of network usage because

It allows many student rows to be ignored on map phase.

|  |  |  |  |
| --- | --- | --- | --- |
| Data combination | Program (1) | Program (2) | Program (3) |
| “score\_3000.txt” and “student\_30000000.txt” | 2min 11s | 0min 22s | 1min 06s |
| “score\_3000000.txt” and “student\_30000000.txt” | 2min 17s | 0min 41s | 1min 25s |
| “score\_30000000.txt” and “student\_30000000.txt” | 2min 56s | Runs out of memory, or doesn’t finish in reasonable time | 2min 25s |

Questions:

1. I used two ukko nodes (ukko006 and ukko007) for running all three programs. ukko006 was used as the namenode and ukko007 as the only datanode.

I tried to use more datanodes, but the single one in usage ended up locking /hadoop/mydata/datanode directory for itself. Thus when I tried to run the script for starting another datanode, I could see from the logs that starting this another datanode failed (due to the lock).

Nevertheless, as can be seen from the running times above, it wasn’t much of a deal to only use one in this case. So I didn’t bother trying to do anything else with it either.

1. Source codes can be found in the attached zip file. There are single Java classes for each program. ScoreStudent.java stands for Program (1), ScoreStudentDistributedCache for Program (2), and ScoreStudentBloomFilter for Program (3).

The performances of the different programs were quite expectable. I initially expected program (3) to perform much better than program (1) with the large scoreset, but after working also with exercise 2, and getting more familiar with MapReduce, I realized all bloom filter really avoids in that case is doing a bit less I/O, and sorting. But as the sizes of scores and students are equal, the running times don’t differ that much either.

ScoreStudent has some optimization in terms of filtering on mapping side. This results in the reduce function only handling either single records from either mapper (which are ignored), or pairs which are written as a result to the file when received. *Also worth mentioning that* *ScoreStudentBloomFilter uses the exact same reduce functionality.*

Program (2) (with distributed cache) performed really well with both small (3000 scores) and relatively large (3 million scores) datasets. It stores the scores in a cache file making the scores are reachable for the mappers. The map function does the joining and passes the pretty much the final result to the reduce function.

When I tried running the same program with the large scoreset (of 30 million rows), the program either crashed due to running out of memory, or seemed to take extremely long to finish. This is also quite expectable because the file itself is around the size of 1GB, and copying that over to the local file system of the other node can take quite a while. Perhaps using ukko006 as both the namenode and as a single datanode could have helped in this case.

Program (3) implements bloom filter by in the main function first going through the scoreset, and storing student IDs of scores (which satisfy the required values for courses 1 and 2 for joining) in a globally accessable BloomFilter object. *This is possibly copied to the datanodes for processing as well.*

The scores are handled the same way in their own mapper as in program (1), but mapper for students is different as it filters out students whose IDs aren’t stored in the bit array of the BloomFilter. The reduce functionality is the same as in program (1) too like also mentioned before.

The program (3) seems to outmatch program (1) with most inputs. It doesn’t do as good as program (2) unless we use fairly large datasets (which is also expectable).

As a sidenote, I also tried to do some optimization with initialization of the

BloomFilter based on the scoreset the program is given. Later on I figured out that it

I could have probably used a higher value for p (= allowed probability of false

positives) as “false positive students” are ignored on the reduce side anyhow if they

happen to appear. This holds because if BloomFilter errornously notices that a

student with id X is belonged to the scores data, the reduce function will see that,

but not a score related to it and the student row is thus just ignored.

1. First of all, the construction time of the bloom filter is also included in the running time of program (3). For 3000 scores it’s around 20 milliseconds, for 3 million scores around 1.75 seconds, and for 30 million scores around 20.60 seconds. These running times were gotten from Java code under the main() function of the program.

2.

|  |  |  |  |
| --- | --- | --- | --- |
| Data combination | Join threshold θ | Result size | Running time (ms) |
| Small dataset | 1 | 3 | 10123 |
| Small dataset | 2 | 7 | 42143 |
| Big dataset | 1 | NA | NA |
| Big dataset | 2 | NA | NA |

a)

I tried various solutions in implementing the task. Nonetheless, it seems none of them is good enough for processing the large datasets. Perhaps a better cluster configuration, and actually using more than two nodes for the processing could have made a difference.

EditDistanceV1 was my initial solution which first in main function stores the words of table1.txt in global file system, and has a two-phase map-reduce procedure for processing the words of table2.txt. It should finally output the correct solution with result sizes of 3 and 7 for the small dataset.

EditDistanceV2 is similar to V1 with an exception that it only needs one phase map-reduce procedure in processing words of table2.txt, and gives the same output as V1. Here in the main function, we compute 2-grams for each word of table1.txt and store them in the global file system.

Map workers compute also 2-grams for each processed word. Next they loop through the globally stored table1’s grams, see if the grams of two words overlap enough and if they do, computes edit distances between these two words. The words are outputted in reduce worker.

EditDistanceV3 has a different approach which tries to keep the amount of data stored in global file system as small enough. It stores only the alphabet (= only characters that appear in small and big datasets), and threshold in the fs.

Here we have two map functions of which one processes table1 and the other table2. The one processing table1 simply outputs the processed word as both key and value with a note in value field that the word belonged to table1.

The other map function computes all “nearest neighbors” of same length to each word. Here the notion of a neighbor is the edit distance. How this works is that if the threshold is 1, the map worker outputs all strings of the same length as the original word (= 12) that can be formed with a single edit distance operation, in this case substitution. If the threshold is 2, it returns all words of length 12 that can be gotten with one or two substitutions. In both cases the map worker also outputs the word itself (= threshold 0).

The point here is that the amount of words formed this way is high, but perhaps yet somewhat reasonable. For example for string “0123456789abc”, the amount of nearest neighbors for edit distance 1 is 205 and for edit distance 2, 21149.

Eventually the map function outputs all the computed nearest neighbors as keys, and the original word as the value.

In reduce phase we write all the <“key”, “neighbours’ original word”> for each processed value if the map function processing table1 also output a key-value-pair to the same key.

Duplicates are removed storing the nearest neighbours in Java’s HashSet.

I still tried something else and wrote EditDistanceV4. This one uses two phase map-reduce procedure where the second phase is reserved for removing the duplicate entries outputted by the first reducer. This was apparently something similar to what was suggested in the exercise session, so I thought it might be a correct solution in processing the big dataset.

Here I first process both tables with a similar map worker. It computes 2-grams for each word, and outputs different combinations of the 2-grams along with the word and table notion. The amount of outputted combinations for each word depends on the used threshold, and the map workers output n-q+1-tq different combinations. Here n is the length of words (12) and q the length of grams (2).

By combinatorics this means that when threshold is 1, for each processed word by either map worker, the worker outputs 220 different keys (= gram combinations separated by commas) along with the word itself. For threshold 2 this amount is 792.

Next we have a single reduce function where for each received 2-grams combination, and a list of values (= words from both tables which satisfy that combination), it stores the values in two lists (one for both tables). Next as brute force solution, it computes the edit distance for each word from list of table1 to each word from list of table2. If the edit distance is at most the threshold, the word pair is written as middle output in the global file system.

This middle output contains lots of duplicate entries and thus the functionality does yet another map-reduce procedure to simply remove the duplicate entries.

All these solutions were good enough in computing the solutions for the small datasets with my own local computer. In Ukko however, only EditDistanceV3 and EditDistanceV4 successfully processed the small dataset with threshold 1. And only EditDistanceV4 successfully processed the small dataset with threshold 2.

The problem with V1 and V2 was that the heap size got too large, and thus the executions failed. V3 on Ukko (with a small dataset) failed to an IOException after 22min since starting execution. Map-phase was then 97% done.

V4 on the other hand performed decently well in Ukko with a small dataset. These execution times are also shown in the table above; around 10s with threshold 1 and 42s with threshold 2.

b)

Performance analysis of the different solutions is somewhat described above.

EditDistanceV4 ended up being the best solution (especially for threshold > 1) because it stores only a little bit of data in global file system that the map workers need. And in this case, the amount of outputted 2-grams combinations (in first map phase) is also much smaller than the amount of nearest neighbors outputted in V3’s map phase.

When threshold = 1, I can’t say whether V4 really outperforms V3 or not. The total running time for small dataset with V3 in Ukko was 16306 ms while for V4, the same was 10123 ms. Locally they performed for me in a similar way; the running time for V3 with threshold 1 was around 10000 ms while for V4, that was around 9000 ms. For threshold 2 however, the running time for V3 locally was around 15 minutes and for V4, only 30000 ms (30 s).

For the program to run fast, we should especially try minimize the amount of I/O and sorting work that happens between actual map and reduce phases.

For exercise 1, we managed to do this with the largest student and score datasets by filtering out scores and students based on (score1, score2) and birth year correspondingly. Bloom filter was also used for filtering out students who aren’t mentioned in the score dataset (with “satisfied” (score 1, score 2)). Similar bloom filter could have been used for filtering out scores beforehand as well. That would have possibly improved performance of working with large score and student sets in problem of exercise 1.

In exercise 2, we’ll have to use separate map workers for processing the table1 and table2. Otherwise we would have no way of identifying, which words belonged to which file. The main issue here is that we have also no way of knowing, which words are such that there exists another word in the other text file which is similar to this one in edit distance.

To avoid I/O of sending all text strings from both text files to all reducers (and not just doing brute-force implementation), we have to somehow decrease the amount of reducers where each word from both files should be sent. This was tried in program V3 with nearest neighbors computation, and with 2-grams with “Threshold for overlap q-gram” lemma introduced in the lecture slides.

Final outputs of exercise 2 with small datasets:

Threshold 1:

4cf4fd7g1e3d 4cf4fd7g1e3d  
635b684afmbf 635b684afabf  
df972g7c81d1 df972g7c81d2

Threshold 2:

492agc3cg74c 92agc3cg74ca  
4cf4fd7g1e3d 4cf4fd7g1e3d  
635b684afmbf 635b684afabf  
6eb8gbe176b9 6eb8gbe1760a  
d0cc40200880 d0ca4020a880  
df972g7c81d1 df972g7c81d2  
g51db0698e14 51db0698e14a