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 somewhat expectable, I think. One thing that surprised me though was that ScoreStudent performed did decently well compared to the other programs especially with the largest scoreset. It does have some optimization like doing the 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 (at least here, it

performed better with all score sets). 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.