EPFL CS-422
Project II

Murat Topak

May 2019

In this project, I have implemented the cube operator, similarity join operator, and stratified sampling for approximate query processing. In all the following experiments, the spark-submit command given on the moodle is executed with executor-memory=10G, num-cores=4, executor-cores=4.

1 Cube Operator

Here the performance of the optimized and naive algorithm will be compared with respect to varying input size, the number of dimensions operated on, and the number of reducers.

First off, the optimized algorithm excels on big data set when the total number of cube groups is small, i.e. when the the number of returned rows is small. This might be the case for columns with fewer distinct values. Indeed, with fewer groups, each group consists of greater portion of data, and MRDataCube outperforms the naive algorithm. The argument for this is that the reducer of the naive algorithm has to compute the measure on a very big set while the optimized algorithm avoids that calculation by directly substituting * to each field. This is more easily observed when you run cube operator on "lo_shippriority, lo_shipmode" to aggregate some other field. These columns are picked because they contain few distinct values resulting in 16 returned rows regardless of the data set after cube is performed. The runtime for the naive on 3 datasets ordered from smallest to biggest is 1.00s, 2.90s, 16.44s whereas these are 1.41s, 2.11s, 4.48s for the optimized one with 4 reducers in both experiments. The difference is subtle for small and medium data sets as can be seen.

On the other hand, if the grouping set is "lo_orderkey, lo_shipmode", then we have lots of different groups in the cube result due to a key's presence. In this case, the returned number of rows increases with the input and for the naive the run time are 1.18s, 5.53s, 14.95s whereas it is 1.31s, 9.63s, 34.69s for the so called optimized algorithm. Here simply the naive algorithm beats its competent since there are a lot of cube groups (6 millions for the big data set) in the result. Hence we conclude the optimized algorithm performs best when there are reducer-unfriendly groups as in the first case.

That being said, increasing the number of reducers from 4 to 16 cuts the run time from 34.69s mentioned above to 19.02s for the optimized algorithm while it doesn't give any improvements for the naive one. As far as the number of dimensions is concerned, the runtime doubles each time a dimension is added so it is consistent with the theory which states cube operator is of $O(2^{|D|} n \log n)$

Remark: I have used * for null in cube operator output of spark, and also each column is separated by a comma in the string.

2 Similarity Join

The similarity join algorithm has only one parameter to play with, that is, the number of clusters. The idea is to define clusters so as not to compare unsimilar words. This is an efficient approach compared to simple cartesian of the words. In the following experiments, edit distance is specified as 2. For 1k dataset, I got around 25s for 10 clusters while cartesian performed in 40s. The number of clusters is essentially the number of reducers in my case, so for example cutting it to 1 increases the runtime to even worse than cartesian. Then to see the effect of the input size, I ran 2k dataset and this time join operator completed in 85s, and cartesian in 140s with 20 word clusters. For the 5k dataset, these times are 795s, 1067s. These indicate of course run time is in the order of $O(n^2)$, n being the input size. However, the randomized algorithm still proves better than the naive.

3 Stratified Sampling

My implementation samples within the specified confidence interval and margin of error with respect to l_extendedprice column. I wanted to fix it like that since for the most of the queries it is that column we are interested in, and also for some of the queries there doesn't even exist a numeric column to select, so going through variance calculations for string data etc. would complicate the process in those cases too. Therefore for each query the query column set is pre-defined in the sampler. Then we sample for each query and then sort them by size to take as much as possible from the samples with respect to budget given. For the row size, I use the Spark API such as IntegerType.defaultSize, TimestampType.defaultSize, etc.

Some query column sets are actually subsets of some other column sets, but for simplicity I do not make use of such supersets. Each query therefore has its own QCS. For some of the QCSs, stratified sampling is effective. This really depends on the grouping list, and the values those columns in the grouping list contain. Small QCSs with not so many groups is one good use case for instance.

The executor makes use of the session object given so it creates a temp table and runs the query on the temp tables thanks to Spark SQL API. Depending on whether a sample exists, the temp table for lineitem is created from the sample or the original table. The return value of the executor is an RDD as I saw on moodle.