DBMS Project 2

Francisco Lozano and Dragoljub Duric

May 19, 2020

1 Roll Up Operator

We implemented the rollup operator using two different approaches, the naive brute force approach and the optimized approach. The naive approach performed an aggregate for each possible key independently, while the optimized version reused the calculations from other rollups, thus reducing the amount of operations required. We used aggregateByKey as the spark function to aggregate the values of the RDD.

Implementation pseudocode:

```
def doNextRollup(rdd, groupingAttributeIndexes, aggAttributeIndex, aggType) = {
   // Takes RDD[(List, Value)] and returns,
   // depending on the aggType (SUM, AVG, etc.),
   // the correct aggregateByKey after removing
   // the last element of the current keys
   // I.E. (List("a","b","c") -> 35.4) => (List("a","b") -> 35.4)
}
def aggregateByKey(rdd, groupingAttributeIndexes, aggAttributeIndex, aggType) = {
   // Takes RDD[Rows] and returns,
   // depending on the aggType (SUM, AVG, etc.),
   // the correct aggregateByKey using
   // as key the grouping attributes
}
def naiveRollup(rdd, groupingAttributeIndexes, aggAttributeIndex, aggType) ={
   val allPossibleGroups =
      //All possible slices from [0, i)
      // from groupingAttributeIndexes
   val aggregates = allPossibleGroups.foreach(
      aggregateByKey(
         rdd,_,
         aggAttributeIndex,
         aggType))

   aggregates.reduce(_ ++ _)
}
def rollup(rdd, groupingAttributeIndexes, aggAttributeIndex, aggType) = {
   var current_aggregate = aggregateByKey(rdd, groupingAttributeIndexes, aggType)
   var rollups = List(current_rollup)

   while(/* Should do rollup */){
      current_aggregate = doNextRollup(current_rollup)
      rollups .::= current_aggregate
   }

   aggregates.reduce(_ ++ _)
}
```

1.1 Results

The results where unexpected (see Fig. 1), the performance was almost equivalent between both implementations even tho one was setup to reuse previous information, reducing the amount of operations required. This led us to two possible hypothesis, either some Spark optimizations happened during execution that we were not aware of, or there was some bottleneck unrelated to the amount of operations required, for example, the distribution of the rows of the RDD in the different partitions.

It is worth noticing that CPU utilization made the profiling results vary, so the values may vary from the true mean.

version	dataset	#attributes	avg time
O	big	2	21.8 s
		3	33.7 s
		4	38.1 s
		5	49.7 s
	medium	2	4.6 s
		3	5.1 s
		4	7.2 s
		5	9.4 s
		6	12.2 s
		7	14.3 s
		8	17.6 s
	small	2	1.3 s
		3	0.7 s
		4	0.8 s
		5	1.1 s
		6	1.3 s
		7	1.6 s
		8	2.0 s

version	dataset	#attributes	avg time
N	big	2	19.4 s
		3	29.2 s
		4	41.2 s
		5	52.9 s
	medium	2	3.2 s
		3	4.7 s
		4	6.4 s
		5	8.4 s
		6	10.6 s
		7	13.1 s
		8	14.9 s
	small	2	1.0 s
		3	0.8 s
		4	0.9 s
		5	1.3 s
		6	1.6 s
		7	1.7 s
		8	2.1 s

Figure 1: Performance of the rollup operators using rdd.count was usef to force the materialization of the rdd. O stands for Optimized, while N stands for Naive.

2 Theta Join

We implemented the algorithm described 1-Bucket-Theta described in this paper with minor modifications to adapt the implementation to Scala using Spark.

2.1 Results

# Part	avg Time
0	49.6 s
2	3.6 s
4	2.8 s
6	2.1 s
8	2.5 s
10	3.7 s
12	1.6 s
14	2.1 s
16	3.7 s
20	2.7 s

Figure 2: Performance of the theta-join operator. Left column are number of partitions, and the right column are the average time measurements obtained. Here, 0 partitions represents the brute force approach. The rest are applications of the theta-cube algorithm. Rdd.count was used to force the materialization of the rdd.

As we can see, the advantages of applying the algorithm are clear. Even tho the profiling results varied seconds between execution depending on the CPU utilization, we can observe it is much faster than the brute force approach, showing a speed up of around 90%, and that the best selection of number of partitions is not evident.

Note that when the resulting number of splits one of the datasets was not an integer, the number of splits was rounded up, and the extra blocks that exceeded the allowed number of partitions were distributed among the other partitions.