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.

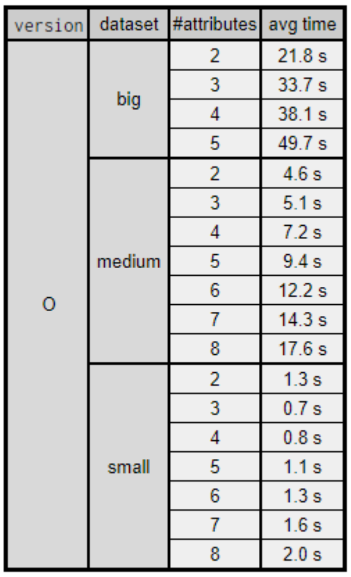
Implementation pseudocode:

```
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(_ ++ _)
}
```

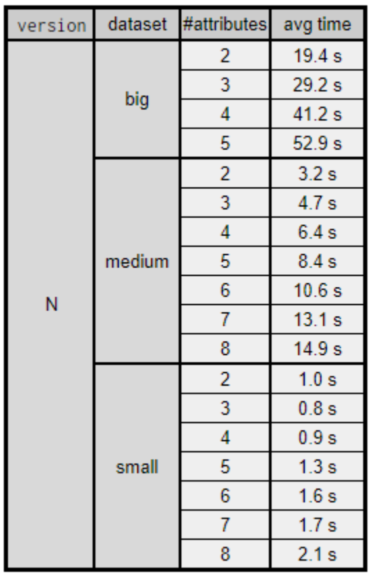
1.1 Results

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 be different by a couple of seconds 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



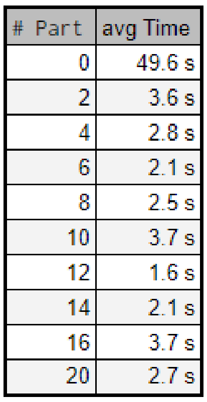
version	dataset	#attributes	avg time
N	big	2	19.4 s
		3	29.2 s
		4	41.2 s
		5	52.9 s
	medium	2	3.2 s
		3	4.7 s
		4	6.4 s
		5	8.4 s
		6	10.6 s
		7	13.1 s
		8	14.9 s
	small	2	1.0 s
		3	0.8 s
		4	0.9 s
		5	1.3 s
		6	1.6 s
		7	1.7 s
		8	2.1 s

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.

3 Locality Sensitive Hashing

In this task we implemented Locality Sensitive Hashing algorithm, both partitioned and broadcast-based.

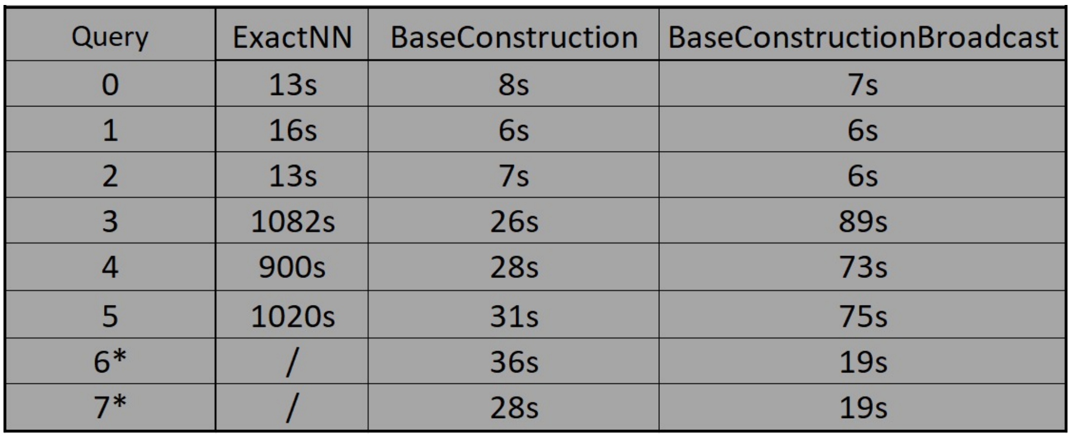
3.1 Results

While testing, we noticed that using of AndConstruction improves precision, and makes recall worse. And that can be justified with fact that AndConstruction will keep only movies which occurs in all Constructions used inside it. And with such procedure it is obvious that movies which remain after it are with high probability also inside exact solution. On the other hand, we noticed that using of OrConstruction improves recall, and makes precision worse. That is justified with fact that OrConstruction will keep all movies which occurs in any of Construction used inside it. With such procedure we can with high probability assume that all similar movies present in exact solution are as well in output of OrConstruction.

We compared execution times for ExactNN, BaseConstruction and BaseConstructionBroadcast. In order to maximize both precision and recall, we used combination of AndConstructions and OrConstructions. First, we created five different ORConstruction each with 5 BaseConstruction/BaseConstructionBroadcast, and then we created AndConstruction from previously computed OrConstructions. That can be written as:

$AND(OR(B,B,B,B,B),OR(B,B,B,B,B),OR(B,B,B,B,B),OR(B,B,B,B,B),OR(B,B,B,B,B))$

where B represents BaseConstruction(partitioned LSH) or BaseConstructionBroadcast(broadcast-based LSH). Hereafter this construction will be called $ANDOR$.



Query	ExactNN	BaseConstruction	BaseConstructionBroadcast
0	13s	8s	7s
1	16s	6s	6s
2	13s	7s	6s
3	1082s	26s	89s
4	900s	28s	73s
5	1020s	31s	75s
6*	/	36s	19s
7*	/	28s	19s

Figure 3: Execution times are shown with respect to different implementation on Nearest Neighbour. ExactNN represents brute force approach(exact solution). BaseContruction/BaseConstructionBroadcast represents $ANDOR$ construction with usage of BaceConstruction/BaseConstructionBroadcast.
For queries 6 and 7, we cannot execute brute force approach, since it requires too much time. For other two implementation we did not use $ANDOR$ construction in order to get reasonable execution times, so we used only one BaceConstruction/BaseConstructionBroadcast.

It is obvious that brute force approach is polynomially slower than Locality Sensitive Hashing. And that it becomes unusable with bigger data sets. The difference between other two implementation is in most cases comparable, with small differences which can be justified with different implementations of two approaches.

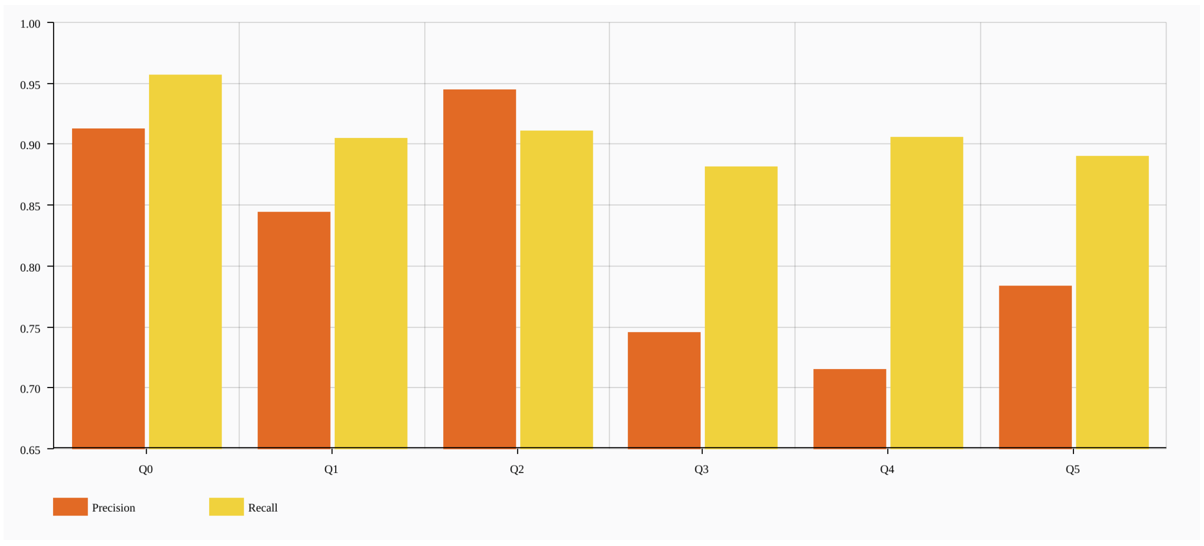


Figure 4: Precision and Recall for $ANDOR$ construction with BaseContruction

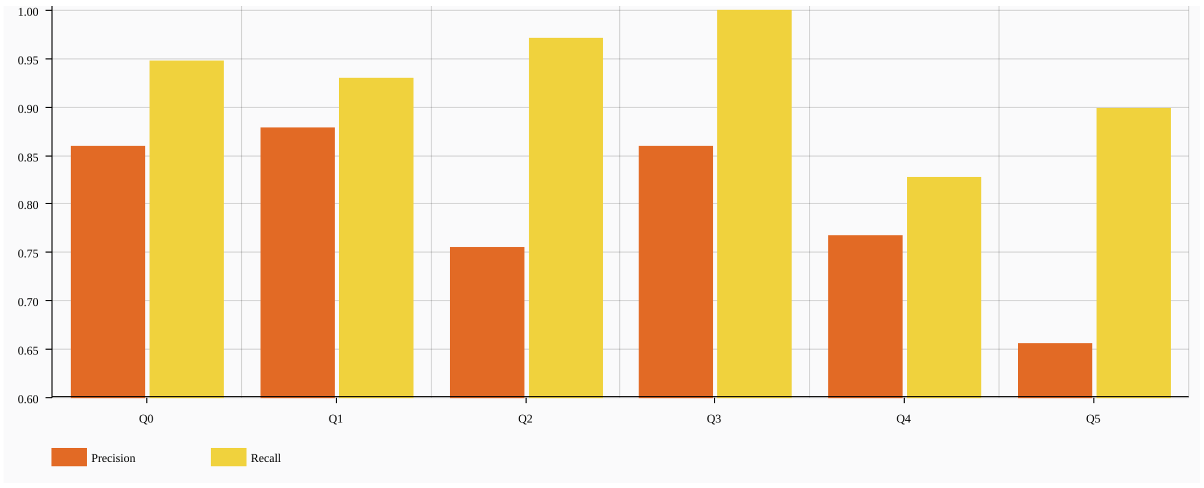


Figure 5: Precision and Recall for $ANDOR$ construction with BaseContructionBroadcast

As show in previous figures Precision and Recall are in most cases comparable, with exception of query 0 in which BaseConstruction get better results, and query 3 in which BaseConstruction get better results. But this is justified with randomization in execution of previously mentioned implementations.

To conclude, in case of small data set and small query (queries 0, 1 and 3), best approach will be to use ExactNN since it returns exact solution, and execution time difference is not huge. With medium sized data set (queries 3, 4 and 5) better approach is to use Locality Sensitive Hashing since on average both precision and recall are bigger than 80%, and time savings are significant. Even more in case of alternative uses of same implementations (medicine for example) it is even possible to get precision or recall very close to 1, with just using AndConstruction or OrConstruction respectively. In the last case, big data set and large queries (6 and 7) only solution is to use Locality Sensitive Hashing in order to get results in reasonable amount of time.