CS-422 Database Systems
Project II

Wentao Feng 293486, Zhechen Su 294180

Task 1:

If not specify in each test, the default setting is 10 reducers, 3 attributes and medium dataset.

1. Input size (number of tuples)
We chang the data size and keep other parameters same in this task. Small data size is 518KB, Medium size is 56.4MB and Big size is 588MB.

Query:

SELECT lo_suppkey, lo_shipmode, lo_orderdate, SUM (lo_supplycost)
FROM LINEORDER
CUBE BY lo_suppkey, lo_shipmode, lo_orderdate

Result:

Input size	Naive	Two-phase
Small	915ms	962ms
Medium	6621ms	6698ms
Big	52708ms	55664ms

Observation:
The result shows that under 10 reducers and 3 attributes condition, Naïve mode always performs a little better than two-phase algorithm, but their time difference is less than 7%. Also, their increase rates are quite similar. In two-phase algorithm, the first phase requires more time to load and calculate. If it does not run in a distributed setup, it is possible that extra first phase operations make two-phase algorithm no advantages than the simple one.

2. Number of CUBE attributes
We chang the number of attributes and keep other parameters same in this task. The attributes are selected of deleted from the blue part in query.

Query:

SELECT lo_suppkey, lo_shipmode, lo_orderdate, SUM (lo_supplycost)
FROM LINEORDER
CUBE BY lo_suppkey, lo_shipmode, lo_orderdate

Result:

# of attributes	Naive	Two-phase
1	2683ms	1643ms
2	3006ms	1994ms
3	6698ms	6621ms
4	9555ms	10168ms
5	18308ms	19308ms
6	107100ms	111708ms

Observation:
When the query has less than 3 attributes, two-phase algorithm works much faster than naïve one. But it is surpassed by naïve algorithm more than 3 attributes. According to two-phase algorithm, the first reduce phase can decrease the data size significantly and can increase the overall performance. It is true at beginning. However, this happens may only in low attributes situation, because if the query is grouped by lots of attributes, the unique key amount become really large making data size cannot decrease obviously in first phase. Therefore, the large attributes number makes two-phase algorithm perform like naïve one.

3. Number of reducers
In this task, we change the number of reducers from 2 to 20 in medium size dataset.

Query:

SELECT lo_suppkey, lo_shipmode, lo_orderdate, SUM (lo_supplycost)
FROM LINEORDER
CUBE BY lo_suppkey, lo_shipmode, lo_orderdate

Result:

# of reducers	Naive	Two-phase
2	5673ms	8518ms
6	5152ms	7251ms
10	6621ms	6698ms
20	5457ms	7071ms

Observation:
The time does not change significantly with respect to reducer number. Two-phase performance is improved with increment of reducers in some degree, but it is still slower than naïve. In our test environment, number of reducers only affects the number of threads operations instead of parallel operations. So, two-phases need extra calculation. but it still consumes similar time. Thus, we can expect that two-phase algorithm would run much better on multiple machines environment.

Task 2:

After implementing the cluster join algorithm, we conduct a series of tests with various data size, number of anchors. The distance threshold is 2 in all tests.

1. the number of anchors
In this section, we keep the data size constant at 3000 rows and change the number of anchors for observing difference.

# of anchors	Similarity	Cartesian	Same result
4	46.33s	75.84s	Yes
16	42.03s	75.84s	Yes
64	77.62s	75.84s	Yes
128	91.42s	75.84s	Yes

Observation:
The results are same as our expectation. When the number of anchors increases, the algorithms will consume much time on finding the cluster centers and the outer partition of each points. Due to the slight deceasing between 4 anchors and 16 anchors, we additionally test 8 anchors and prove that the joining time has a downward trending in a small range. That is because precise partition can reduce useless edit distance judgment within the cluster.

2. the data size
In this section, we test 3 different dataset sizes and keep the number of anchors at 16.

# of rows	Similarity	Cartesian	Same result
1k	6.39s	8.97s	Yes
3k	42.03s	75.84s	Yes
5k	116.10s	231.22s	Yes
8k	298.47s	649.39s	Yes

Observation:
The results are same as our expectation. The performance gap between similarity and cartesian product increases when the dataset size rises. The computational complexity of cartesian product is exponential, which is $O(N^2)$ and the counterpart of similarity join is $O(N)$. Therefore, when N increases, the gap is enlarged.

Conclusion: Overall, the similarity join performs much better than cartesian product, especially in large scale database. However, there is a minor flaw in our implementation. From Ye Wang's paper [1], SimilarityMapper algorithm will not have insufficient computation if the inner list only keeps inner points. But our results are always below the true value. So, we modify the algorithm and check the redundancy at the output. In this way, the computation cost increases about 50%

Task 3:

Instruction:
1). In Description.scala, we change the var sampleDescription from 'Any' to 'List[(Int, Seq[String])]' and var samples from 'List[RDD[]]' to 'List[RDD[Row]]'. Making them specific to get rid of type error in Executor.scala
2). All of our queries' aggregation results are multiply by the total_size/sample_size, which can estimate true value properly.

This task, we implemented a program to create samples according to confidence and error requirements and then choose the most suitable sample to estimate coming queries. The main steps are:

1. In order to figure out how many samples we should generate and what are their QCS, we calculate all combinations of a QCS and get their strata numbers, trying to find the minimum total sample size which could cover maximum columns. But the strata count of one set could not be larger than 15000 or it is likely to have numerous strata having only 1 item inside. And the SRS sampling method has to take all of them to keep sample complete, which is often against storage budget. Therefore, the pipeline could be: for each query, iterate all possible columns combinations, find one has the largest columns set with its group count less than 15000. If the column set is already in our sample, use this as the query's sample. Otherwise, take the column set as a new sample and add to our list.

   Table 1 shows the results of samples we use and all of them has really small count comparing to total size. It means our samples could stratify more properly and take more general information. Table 2 is sample allocation for each query. Red columns name means they are covered by our sample.

Sample	Columns	Count	Ratio
S1	["l_shipdate", "l_returnflag", "l_linestatus"]	3817	0.06
S2	["l_suppkey"]	10000	0.17
S3	["l_discount", "l_quantity", "l_shipmode"]	3850	0.06

Table 1: Sample columns set and its count

sql_num	QCS	Sample
1	l_shipdate, l_returnflag, l_linestatus	S1
3	l_orderkey, l_shipdate	S1
5	l_orderkey, l_suppkey	S2
6	l_shipdate, l_discount, l_quantity	S3
7	l_orderkey, l_shipdate	S1
9	l_suppkey, l_partkey, l_orderkey	S2
10	l_orderkey, l_returnflag	S1
11		S1
12	l_shipmode, l_commitdate, l_receiptdate	S3
17	l_partkey, l_quantity	S3

18	l_orderkey, l_quantity	S3
19	l_quantity, l_shipmode, l_shipinstruct, l_partkey	S3
20	l_quantity, l_partkey, l_suppkey, l_shipdate	S3

Table 2: QCS of each query and sample allocation

2. Having the number and columns sets of samples, we then tried to calculate sample size K within upper bound: storage budget and lower bound Int 1. Firstly, for each sample, K is assigned as half sum of upper bound and lower bound. Then calculate z_{alpha}*sqrt(v/n) using formula in moodle. Compare this result with error limitation sum(l_extendedprice)*e. If result is less than limitation, we assign upper_bound = K, else assign lower_bound = K. Go back to first step until upper_bound is lower than lower_bound and now the K is the optimal sample size. Every time we have a new sample, there is a need to ensure K is less than memory budget. Hence, the sampler part is done, and we get a list of samples with description of sample ratio and QCS.
3. For executor, we employ Sparksession.sql to execute the query and replace parameters according to its needs. Because we allocate suitable sample before, we here only need to replace lineitem table by the specific sample we assigned. Then, multiply aggregated result with inverse ratio and execute sql sentence.

Reference:

[1] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. In Proceedings of KDD, 2013.