CS-422 Database Systems

Project II

Mike Bardet, Diana Petrescu

Task 1 Implementation of a CUBE operator

Input size

Number of reducers: 10

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

	Execution time			
	Input size	**Small**	**Medium**	**Big**
oAlg	Two-phase	0.92s	10.28s	134.55s
	Naive	0.40s	4.85s	163.85s

The naïve approach performs better for the small and medium dataset. This may be because of the reduce_by_key performed in the Two-phase algorithm. If there is too much distinct tuple compared to the total size, then the reduce_by_key don't simplify anything and will in fact take time for no improvement. On the other side, for the big dataset, there is certainly enough identical tuples for the given attributes such that the reduce_by_key makes sense.

Number of CUBE attributes

Number of reducers: 10
Input size : medium

```
SELECT {attributes}, SUM (lo_supplycost)
FROM LINEORDER
CUBE BY {attributes}
```

	Execution time			
	Number of attributes	**2**	**3**	**4**
oAlg	Two-phase	2.62s	10.28s	26.46s
	Naive	3.00s	4.85s	21.22s

Here the results are not very clear, the two-phase algorithm does not seem to perform better. This may be because we used the medium dataset. Computing 4 attributes on the big dataset is a bit too much. Also, the choice of the attributes may have an influence. For example if there are a lot of distinct values for a given attribute, or very few.

Number of reducers

Input size : medium

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

	Execution time				
	Number of reducers	**5**	**10**	**15**	**20**
oAlg	Two-phase	10.86	10.28s	12.35s	11.1s
	Naive	5.95	4.85s	5.91s	5.69s

Since we are doing the experiments on a laptop, the number of reducers will not have any effect beyond the number of tasks that the laptop can execute in parallel. From 10 reducers, the tasks are simply queued, so there is no point to increase the reducers further.

Task 2 Implementation of a similarity join operator

Algorithm and Implementation

For this task we've implemented a cluster join. We followed the steps mentioned in the paper https://dl.acm.org/citation.cfm?doid=2732977.2732981. First, we extracted the anchor points. We compared them one with each other because, for example, if we have a large number of anchor points compared to the number of records, two anchor points could be at a distance < *dthreshold*.
Then, we assigned each remaining point to its closest anchor point and we kept track of the ones that were at a distance < *dthreshold* from the anchor points. In the next step we calculated for every record its possible belonging to outer partition of the anchor points. Then we've taken the combinations of all the pairs by home partition.
However, in order to reduce the number of similarity checks, we've paid particular attention to two main things concerning the outer partition points.
First of all, we didn't want to recompute the distance between two points belonging to the same outer partition but neither of them belonging to the home partition (because we already computed them, for example, if they both belong to another home partition or if one is in the outer partition of an anchor and the other is in the home partition of another anchor). So,

we've set a flag for the outer partition points and instead of taking the combination of pairs, we just compare each outer partition point to the points in the home partition of the same anchor.
Moreover, we've also paid attention to send the intersections of outer and home partitions to only one of the two partitions based on a hashing function (for example, if a record q1 is in the Home partition of the anchor point A1 and in the outer partition of the anchor point A2, and a record q2 is in the home partition of A2 and in the outer partition of A1, we send only q1 to A2 or q2 to A1 but not both).

Results

Method/Size	*1K*	*4K*	*10K*
4 anchors	12.542296917	203.722175767	-
64 anchors	14.006953181	196.108837608	-
256 anchors	15.942031852	183.534034281	1080.864933345
Cartesian	13.608893986	206.559920597	1993.416296197

These results are obtained by running locally the cluster join (with different numbers of anchor points) vs the Cartesian product of all pairs of records.

We can notice that the cluster join indeed prove to have better results than comparing all the pairs one another (Cartesian). This is especially true with 10K where the cluster join was almost 2 times faster. As we increase the size, the gap is even more significant. This is normal as, for example, if we need to compare 1'000 records, the Cartesian method need 1'000'000 comparisons (exponential increase with the size) whereas the cluster join with 4 anchors needs approximately 250*250*4 = 250'000 comparisons.

However, as we can see with the 1K, we need to choose carefully the number of anchors. If we take too many anchors, the overhead due to the algorithm and our implementation (the fact that we first distribute the points to closest anchor, that we compare the anchors together, etc.) can alleviate the gain of performance due to clustering. It can be worse than just doing a cartesian product.

Task 3 Approximate query processing

Algorithm and Implementation

For this task there are a few important thing to notice.

First, for the choice of the QCS, we did some analysis on the attributes distribution between the queries we've implemented. Here is what we found and chose: A query depends on 6 attributes but we've judged that taking 6 attributes for the QCS is too voluminous (specially because it is the case for a single query). So, we've chosen first to take the combination of the 5 attributes that appear most frequently, namely: *l_shipdate*, *l_orderkey*, *l_suppkey*, *l_quantity*, *l_partkey* (which can be used to answer 6 queries). Moreover, there are 5 queries

left that can not be answered yet (plus the query number 12 with 6 attributes). So, we've taken also the best combinations of 4 attributes to answer those queries, namely: (*l_returnflag*, *l_orderkey*, *l_suppkey*, *l_partkey*) and (*l_shipdate*, *l_returnflag*, *l_linestatus*, *l_orderkey*). All of those together allow to answer 9 out of 11 (+1) queries if the storage space permits it. That's why we think it is a judicious choice. We could also have added as many QCS (and samples) as we want to cover more precisely the queries but we've decided to keep those one in order to have a good storage usage.

Moreover for the implementation of the Sampler. In order to choose the best set of samples given a storage budget and accuracy requirements, we've done the following way. First, we've chosen the first hardcoded QC. We grouped lineitem based on those attributes. We then selected a random *K* (between 1 and lineitem.count). Then, using an iterative process, we have calculated the K *minimal* value that satisfies the error and confidence interval requirements (cf. S. Lohr: Sampling Design and Analysis) for all the groups. Then, we sampled with the *minimal K* and, if we had enough storage budget left, we added it to the result and repeated the steps for all QCs.

Finally, we've implemented all queries in executor. However, we've kept in the code the whole table *lineitem* in order to be sure to obtain the right results (with any parameters used for testing). However, in query 1 code, we've mentioned how we should use the sample method. More precisely, as explained in the article, we need to choose the superset with the smallest number of attributes that our query needs or the whole table if no superset exists. To go further, we could have implemented the test on (memory) subsets and the choice of the sample with the highest selectivity (without guarantee).