Michaël Spierer

Maxime Sierro

Project 2

Task 1

We will first take a look at the performance depending on the input size. We used subsets of lineorder_small of 1K and 3K rows (the complete one is about 6K rows). We averaged the runtimes of 500 independent executions for each case.

1K rows :	Naive : 29ms	Optimized : 33ms
3K rows :	Naive : 31ms	Optimized : 32ms
6K rows :	Naive : 30ms	Optimized : 32ms

The results are all around pretty disappointing. The differences are negligible and can mostly be attributed to external factors : even with the 500-average, we get slightly different results every time. A way larger input would be necessary to see major improvements in the optimized case. The good news is that the performance seems to scale well with the size of the input.

Now, we look at the impact of the number of grouping attributes, from 1 to 3. Once again, we averaged the runtimes of 500 independent executions for each case. This was done on the complete input set.

1 grouping attribute :	Naive : 15ms	Optimized : 18ms
2 grouping attributes :	Naive : 27ms	Optimized : 26ms
3 grouping attributes :	Naive : 30ms	Optimized : 32ms

Unfortunately, there are again no real differences between the naive and optimized cases. The increase in runtime with regards to the number of attributes was expected, since the number of groups grows exponentially in rollup. It is also heavily dependent on the input values : if for example a grouping attribute is always distinct, then no additional grouping attribute will make a difference with regards to the groupings and thus outputs.

Task 2

We take a look at the different execution times for different numbers of reducers. The two datasets each have 4K rows, and we averaged 500 independent runs for each case.

R = 1 :	142ms
R = 10 :	136ms
R = 100 :	127ms
R = 1'000 :	123ms
R = 4'000 :	121ms
R = 10'000 :	128ms
R = 100'000 :	128ms
R = 1'000'000 :	129ms
R = 1'000'000 :	134ms
R = 16'000'000 :	148ms

It looks like the "sweet spot" is around 4'000. Extreme values of r produce poor results, which makes sense since they don't prune as many comparisons. The two most important optimizations of our implementation quickly discard non qualifying regions and quickly recognize regions which are entirely made up of qualifying pairs. In both cases, individual comparisons inside these regions are not needed which speeds up the process. Intuitively, you want regions fitting those criterias to be as large as possible for this reason. Both very low and very high values of r minimize the number of prunings (low values of r result in regions that are too large to fit one of the 2 ideal conditions).

It is important to note that the "set-up" time is major. Since we wanted our code to work for any input size, we need to collect the two datasets and obtain their size in the ineq_join method which takes a significant time. Then, the sampling and sorting of values to compute the bounds is also major. Our experiments show that this set-up time is over 100ms. This makes the previously computed runtime differences more significant than they first appear.

Task 3

	ExactNN	LSH	LSH-Broadcast
q0	36ms	271ms	192ms
q1	12ms	162ms	180ms
q2	8ms	162ms	138ms

In order to test our implementation, we set 0.3 as the threshold for ExactNN and made combinations of ANDConstruction and ORConstruction.

We expected to see different results since LSH is supposed to take less time than ExactNN. One possible reason for the LSH runtimes going over 100ms is likely its setup, notably the construction of our lookup table. Another reason could be that the dataset is relatively small and therefore the optimized method shows less performance. The larger the dataset the more “value” there is in opting for the optimized method. We can expect ExactNN to be very slow on large datasets. The best expected algorithm is LSH-broadcast since we try to minimize the shuffling.

	Recall	precision
q0	0.9350	0.4061
q1	0.9209	0.3544
q2	0.9088	0.3027

The above times were obtained through local tests and not through the cluster due to multiple issues with spark-submit, hadoop and so on (one of us actually never managed to connect via ssh even though he had access to the VPN of EFPL). Hence, we had to do this testing with limited resources.