CS-422: Database Systems

Project 2

Manuel Leone 307640, Edoardo Debenedetti 313740

Note about the test executions: performances evaluations of our implementations for the three tasks have been done with a fixed Spark configuration on the cluster. We used such layout because we found it had a good work balance across reducer, with no I/O nor out-of-memory errors.

The configuration we used is: `--executor-memory 16g --executor-cores 8 --num-executors 4`

For each test run, three warm-up executions have been executed before evaluations of the implementations' performances and the RDDs containing the datasets have been cached with a call of `.cache()` on them. To trigger the transformations execution in Spark, we printed the result's elements *count*. Every test result has been averaged over **5 runs** and all the plots are presented with error bars given by the standard deviation of each group of measurements.

1 Task 1: ROLLUP Operator

We tested the ROLLUP implementations using the `lineorder{small, medium, big}` datasets, using, for each dataset, 1/3, 1/2 and the entire dataset. For every dataset size, we tested ROLLUP from 0 to 8 indices, using, for the grouping, the least granular columns (i.e. those with the least number of unique values) as most external ones and the most granular columns as the most inner ones. The columns we selected are, in order, `lo_suppkey`, `lo_orderpriority`, `lo_shipmode`, `lo_linenumber`, `lo_tax`, `lo_discount`, `lo_custkey`, `lo_partkey`, `lo_orderkey`, with indices `[4, 6, 16, 1, 14, 11, 2, 3]` respectively.

The number of unique values for each attribute of the lineorder relation (in the small dataset) are presented in Table 1.

Attribute	Number of unique values
lo_suppkey	2
lo_orderpriority	5
lo_shipmode	7
lo_linenumber	7
lo_tax	9
lo_discount	11
lo_custkey	20
lo_partkey	200
lo_orderkey	1500

Table 1: Number of unique values for each attribute of the lineorder relation (based on the small dataset)

The results obtained by the tests run on the entire datasets can be found in Figure 1. We first make a comparison between the general trend of the performances of the naive implementation and those of the optimized one on the different data sets. We then move to an analysis of the execution time given different number of indices to group.

1.1 Naive VS Optimized

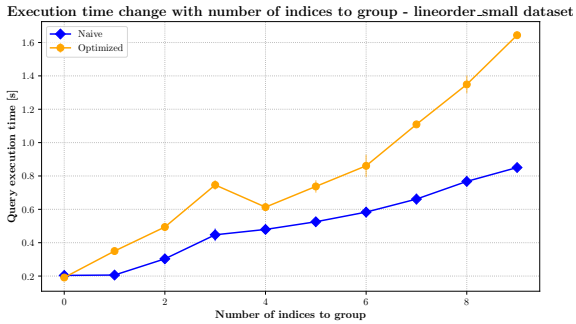
We can observe that, while the **naive implementation outperforms the optimized** one for all the indices with the **small dataset**, the **opposite** happens with the **larger ones** (notably, in the large dataset the difference is huge). In order to explain this, we will first explain how our optimized implementation work. We first compute the aggregate of the innermost level and we then compute all the others using a tail-recursive function (thus optimized into a loop by Scala compiler) keeping an RDD with the latest aggregated values (used to compute the aggregate in the following level) and another RDD containing the data aggregated in the levels before the latest one (used as accumulator). **In the optimized version we then have 3 RDDs**: one for the current aggregation, one for the latest aggregation and one for all the aggregations done before the latest one. On the other hand, **in the naive implementation we only have 2 RDDs**: one for the current

aggregation and one for the aggregations done before. Then the fact that the naive implementation outperforms the optimized one in the small dataset could be explained by the **overhead due to the additional RDD** used in the optimized implementation.

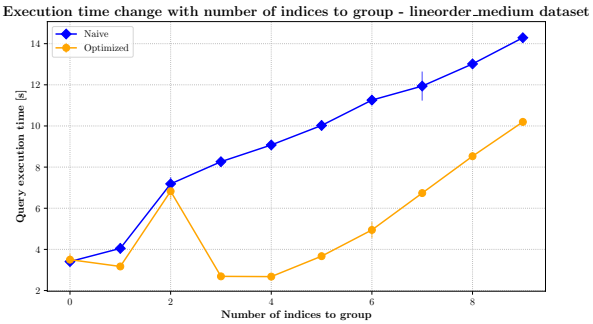
1.2 Performances given different numbers of indices

Let us first observe that, unsurprisingly, the naive implementation always takes more time if more columns have to be grouped. The only exception is in the large dataset with 4 indices, but both measurements have a large standard deviation (maybe due to the cluster being more loaded during the execution of these test queries), so we can expect the real trend to be consistent with the other measurements.

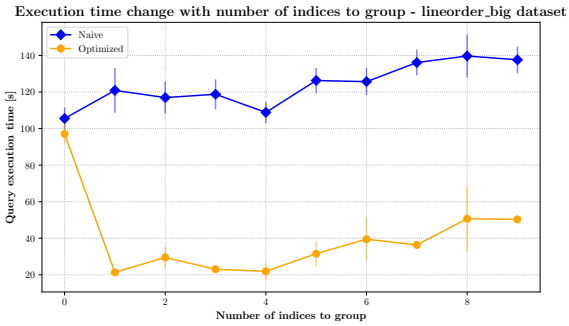
Regarding the optimized implementation, instead, we have a couple of noteworthy exceptions to the increasing trend: when we ROLLUP 3 columns in the small dataset, 2 columns with the small medium dataset, and no columns in the large dataset. Regarding the small dataset, we have a small spike when rolling up 3 attributes (`lo_suppkey`, `lo_orderpriority`, `lo_shipmode`). Given our implementation, we can't find a reasonable explanation for this, since, theoretically, each new ROLLUP query on n columns should do *at least* the work done by the query on $n-1$ columns and then the time should grow with the number of grouped attributes. Thus, we assume that these spikes are due a temporary reduction of the resources assigned by the resource manager.



(a) `lineorder_small` Dataset



(b) `lineorder_medium` Dataset



(c) `lineorder_large` Dataset

Figure 1: Execution time changing with number of partitions with different sized datasets averaged over 5 runs.

2 Task 2: ThetaJoin Operator

The ThetaJoin operator has been analyzed by using the two datasets `taxA4K.csv` and `taxB4K.csv`, extracting also subsets of dimensions 1K, 2K, 3K and 4K (full dataset), where the dataset A is always used as operator's left-side (i.e., the queries are of the form $A < B$ or $A > B$).

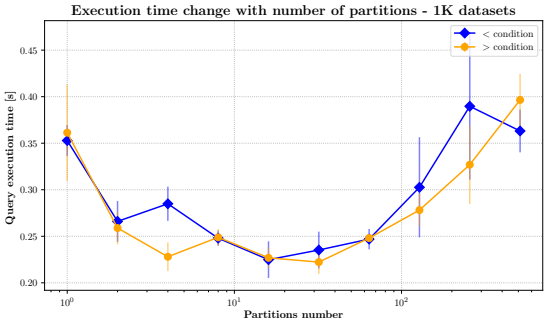
For the `attrIndex1` and `attrIndex2` we tried all the possible combinations of columns 1 and 2 (as we were restricted to integer values). In the end, we selected `attrIndex1 = attrIndex2` $= 2$, because these two present the highest number of different values, so they are more suitable for the task and because the analysis of the results on the cluster showed these two columns have the highest query response time among the combinations.

Finally, the number of partitions was *exponentially increased* with powers of two, specifically we stressed the implementations with $1, 2, 4, 8, 16, 32, 64, 128, 256$ and 512 partitions.

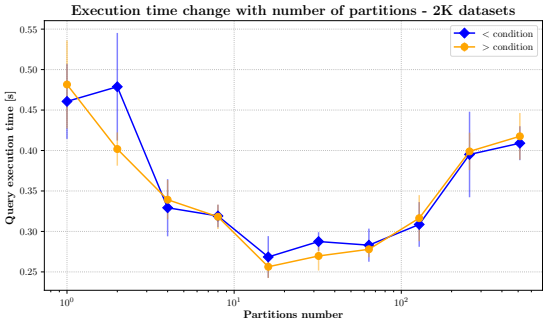
The executions times vs the increasing number of partitions for each dataset are presented in Figure 2, while a summary of the best results is shown in Table 2.

A similar trend for all the datasets dimensions can be recognized in all plots: having a **too low or too high** partitions number makes the Bucket-theta algorithm perform **worse**, while the best settle in the **middle** of the plot, moving towards the right when increasing the dataset dimension. Also, the standard deviations are the lowest when the algorithm performs the best, proving that a correct partitions number makes the algorithm also more **stable**.

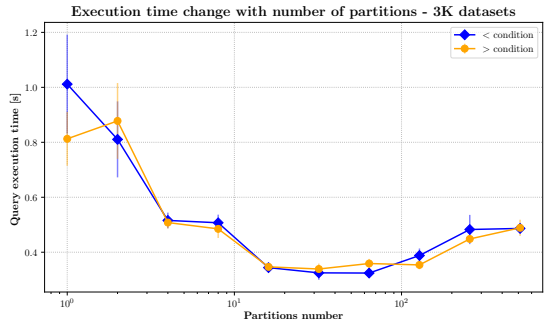
The motivation of this behavior can be found in the relation between the number of partitions and the pruning stage of the algorithm. When a low number of partitions is used the algorithm **doesn't make any pruning** with high probability, so it runs the comparisons an all partitions and doesn't prune anything within the partitions. When we increase the number of partitions the algorithm starts to prune more and more, reaching the **perfect working point** somewhere between 16 and 64 partitions (depending on the dataset). From now on the increase may worsen the query timing, because even if the number of pruned partitions increases, the used ones probably get so **smaller** that the need to **coordinate** their executions (through shuffling, mapping and reducing) increases the execution time.



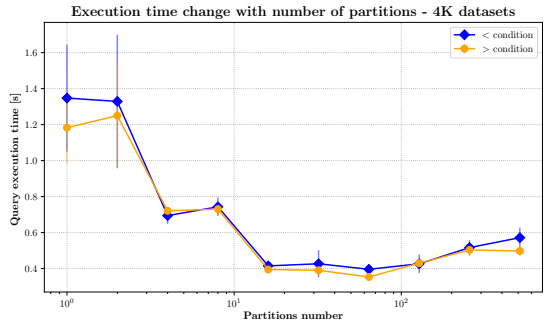
(a) 1K Datasets



(b) 2K Datasets



(c) 3K Datasets



(d) 4K Datasets

Figure 2: Execution time changing with number of partitions with different sized datasets averaged over 5 runs.

Dataset	Condition	Partitions	μ (s)	σ (s)
1 **K**	<	16	0.225	0.019
	>	32	0.222	0.013
2 **K**	<	16	0.268	0.025
	>	16	0.256	0.014
3 **K**	<	64	0.324	0.018
	>	32	0.339	0.019
4 **K**	<	64	0.396	0.009
	>	64	0.353	0.015

Table 2: Best query time and standard deviation for each dataset over 5 runs.

3 Task 3: Near Neighbor Query Processing

Task 3.4: Composite Construction queries performances

The results of precision and recall obtained with the composite queries we built can be found in Table 3: there we present the mean and the standard deviation of precision and recall over 100 separate runs of the composite constructions. We also show the required precision and recall for each query and the percentage of the runs which satisfied the requirements. We now discuss how we have chosen composite constructions for every query.

Query	Precision				Recall			
	Required	Above required	μ	σ	Required	Above required	μ	σ
0	0.7	100%	0.87	0.02	0.83	94%	0.85	0.01
1	0.98	98%	0.99	0.02	0.7	100%	0.72	0.01
2	0.45	100%	0.52	0.02	0.9	100%	0.94	0.01

Table 3: Mean and standard deviation of precision and recall of 100 separate runs of composite constructions.

Query 0 Since, in this query, the required precision and recall are moderately high, we need a composite construction that allows both low false positives and low false negatives. However, because of the fact that the required recall is higher than the required precision, we should avoid false negatives more than false positives. Thus we chose and **AND+OR** composite construction. We found that $r = 2$ and $b = 2$, with a total of 4 base constructions required (the minimal amount required for this kind of construction), were enough to obtain sufficiently accurate results, with only 6 out of 100 runs having a recall lower than the required one.

Query 1 In this query, instead, the required recall is smaller than the required precision (which is, indeed, very large). We should then avoid false positives as much as possible, while we can admit some false negatives. Thus, we chose an **AND** construction, with $r = 3$ (with a total of 3 base constructions). It was enough to obtain the accurate results reported: 98 runs out of 100 had results above the required ones.

Query 2 Finally, in this query the required precision is remarkably smaller than the required recall. We then chose an **OR** construction since it allows few false negatives, while being more prone to false positives. With $b = 3$ (with a total of 3 base constructions) we had accurate enough results, with 100 out of 100 runs with both precision and recall being above the required values.

Task 3.5: Base Constructions queries performances

The performance evaluations for the two Base Construction types (Partitioned and Broadcast) has been performed both in terms of speed performances and goodness of the results with respect to the ExactNN implementation.

As required, queries 0 to 2 has been executed using `lsh-corpus-small.csv`, queries 3 to 5 with the dataset `lsh-corpus-medium.csv` and queries 6 and 7 with `lsh-corpus-large.csv`. For the goodness evaluation, a distance threshold of **0.3** has been used for all queries. Table 4 presents the different queries/corpuses dimensions, to better understand the changes in performance over the different queries.

We now proceed to analyze the results of this analysis.

Query	Corpus	Query Size	Corpus Size
0		319	
1	`lsh-corpus-small.csv`	336	3249
2		327	
3		3193	
4	`lsh-corpus-medium.csv`	3176	32526
5		3276	
6	`lsh-corpus-large.csv`	32287	323187
7		32348	

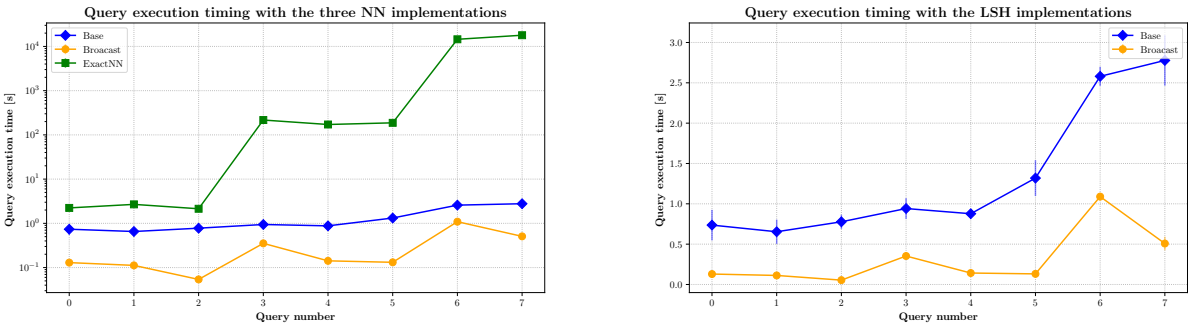
Table 4: Queries and corpuses dimensions (expressed as number of rows in the dataset).

Timing evaluation

The execution time of each implementation has been computed only considering the call of `.eval(queryRDD)`, without considering the time needed to **instantiate the class and performing the pre-processing** on the corpus. However, this time represents only a small overhead (probably slightly more pronounced in the Broadcast implementation due to the broadcasting that must be performed), so the relationship between the different implementations are still valid even without considering it.

The execution time per query is presented in Figure 3. Figure 3a clearly shows how the two LSH implementations **outperform** the ExactNN in execution time, as they have an execution time fixed between 0.1-1 s, while ExactNN has a step increase in its timing (going from few seconds for the first queries, to half an hour for the medium ones, up to 4 hours to complete the most complex queries).

Figure 3b zooms instead on the two LSH implementation, to better compare them. Once again the results are as expected, with the Broadcast implementation that performs better than the partitioned one in all the queries. The performance improvement is due to the introduction of **broadcast variables** in place of the joins used by the Base implementation. Thanks to the broadcasted maps the executors can perform all the Min-hash computation **locally**, avoiding **excessive shuffles** and data transfers among the different partitions. Also, broadcast's standard deviation is lower for all queries, meaning that this implementation is far more **stable** than the partitioned one (once again, thanks to the shuffling reduction).



(a) ExactNN, Base and Broadcast implementations

(b) Zoom on Base and Broadcast implementations

Figure 3: Execution time changing for the different queries averaged over 5 runs (with only exception of queries $6-7$ ExactNN implementation, in which only a single run is performed).

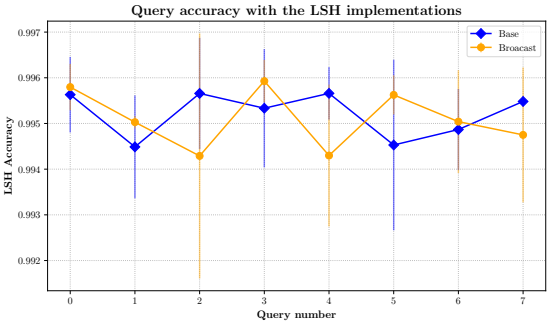
Goodness evaluation

To evaluate how similar our results are with respect to the correct result (ExactNN output), we used *accuracy, precision* and *recall*, as these are common metrics used in information retrieval context. Moreover, we computed the *average distance* of each query movie from its predicted neighbors both for ExactNN (`exact_distance`) and the LSH implementations (`lsh_distance`), then we measured the pointwise difference `exact_distance - lsh_distance`. This metric enables to understand if the LSH implementation is more oriented toward producing **False Positive** (**FP**) or **False Negative** (**FN**) for each point. Specifically, if `exact_distance - lsh_distance` is positive this means that the LSH is more prone to produce elements which are **below** the exact distance threshold (FP), while in the opposite case it **hasn't considered some neighbor** (FN), that otherwise would

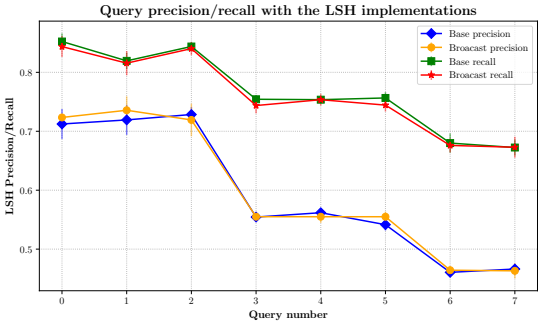
lower the average distance. Finally, an average over these differences is computed to see what is the trend over the full query.

Figure 4 shows the results for the given metrics, which trends are as expected. For the accuracy (Figure 4a), both LSH implementations present an **high accuracy** (at least above 0.9) for all queries. These high results are due to the high percentage of True Negative (TN) for each element in the dataset (and for this reason, accuracy usually is not used in an Information Retrieval context, such as the one we are analyzing), which shifts up the accuracy value. The high standard deviation is due to the **random nature of LSH**: losing a TP or gaining a FP/FN at each run dramatically changes the overall accuracy over the dataset.

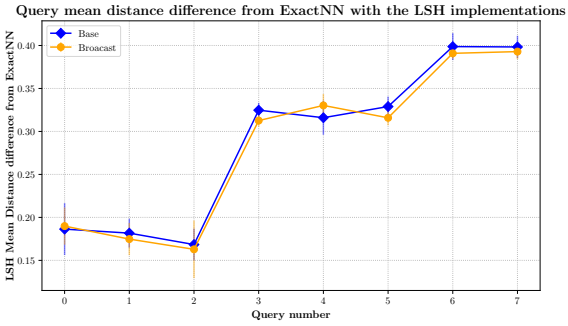
The other metrics enable a better evaluation than accuracy. Figure 4b shows the precision and recall values for all queries. Both metrics go down as the size of query/corpus increases because it's difficult for the LSH to find the correct set of neighbors, and both implementations show the **same trend** (as expected, as the operations they perform are the same, but only done in a different way). The recall is always **higher** than the precision, and the rationale behind that is easily found by analyzing the mean distance difference between exact and LSH implementation (Figure 4c). The average distance `exact_distance - lsh_distance` over the dataset is always **positive**, this means that the LSH implementations always produce a higher percentage of **FP**, which decrease the precision while it doesn't touch the recall. Hence, LSH is really good in finding positive examples, but it's *too good*, in the sense that even if it finds the correct neighbors, it fills the result with many unwanted elements that are below the threshold. For this reason, a AND construct may be useful for increase performance (and it is, as we have seen in the analysis of query 1). To conclude the analysis of this plot, we can see that the distance difference **increases** as the query/corpus sizes increases, so the LSH finds way more FP (and consequently precision decreases).



(a) LSH implementations' accuracy



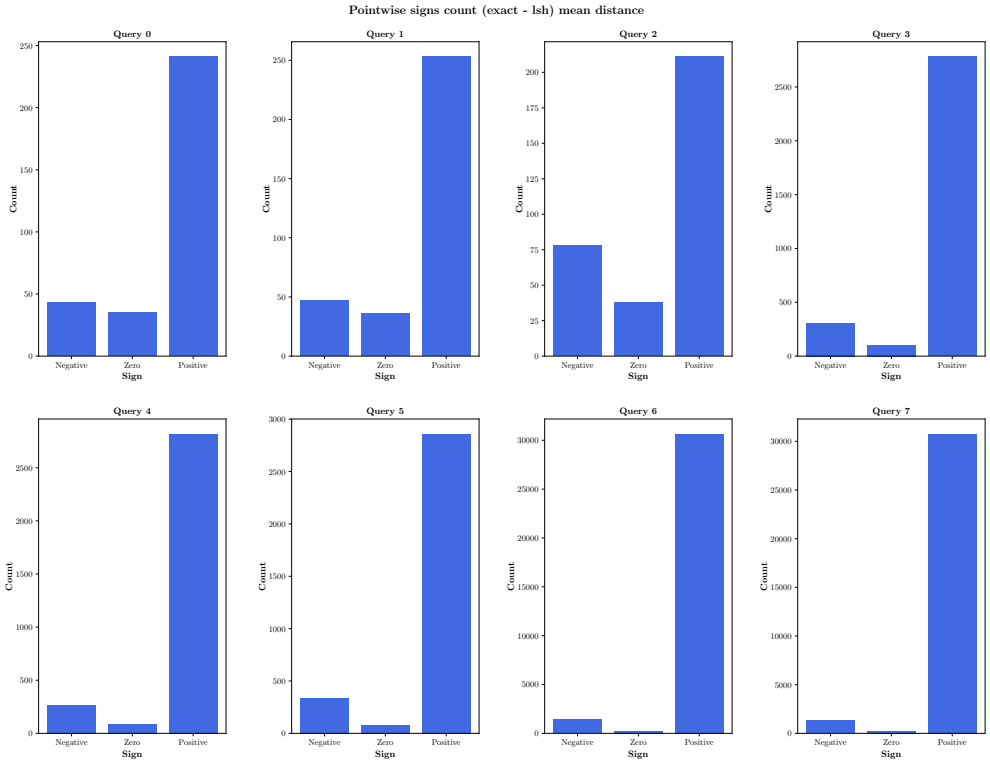
(b) LSH implementations' precision and recall



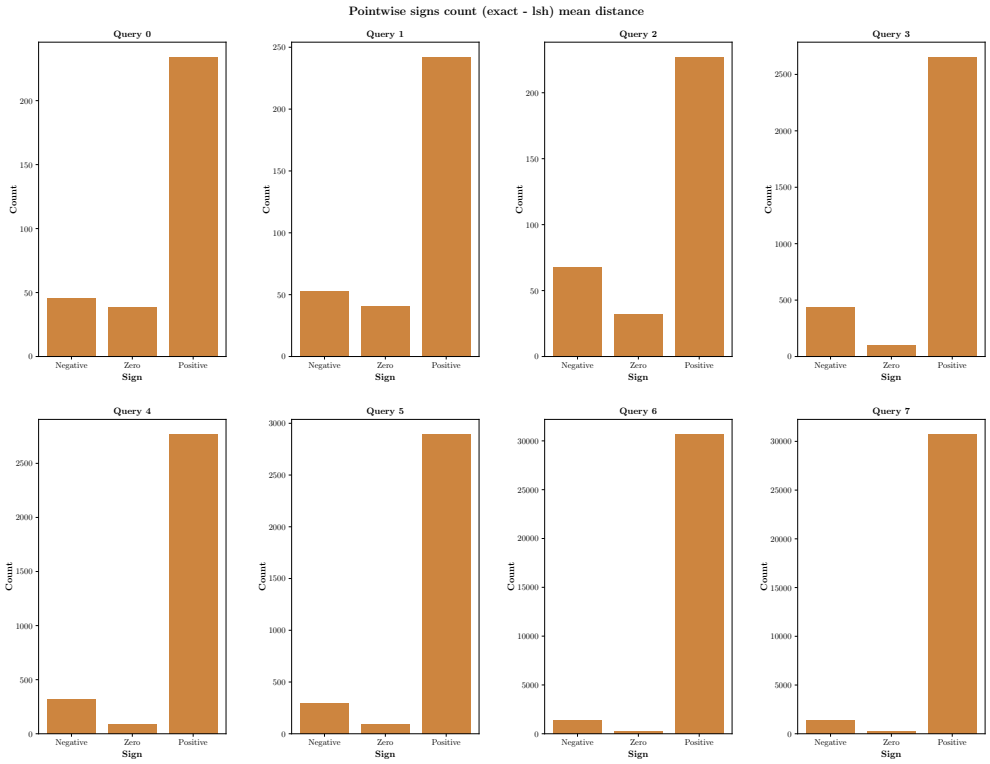
(c) LSH implementations' mean distance difference with respect to exactNN, computed as `exact_distance - lsh_distance`

Figure 4: Performance evaluation for the different queries averaged over 5 runs.

Finally, Figure 5 shows the count of elements for which the pointwise `exact_distance - lsh_distance` is positive, negative or zero. Once again the trend is the same for both partitioned and broadcasted construction (confirming the equivalence of the implementations). Also, the number of query points for which the distance is positive is **way higher** than the negative or zero counts, going from being 5 times higher than their sums for the queries 0 to 5, up to **10 times higher** for the last two.



(a) Base construction



(b) Broadcast construction

Figure 5: Count of query points for which the difference of the mean `exact_distance - lsh_distance` is positive, negative or zero, for all queries, for both construction.