Operator Implementation in Spark

Eric Jollès, Robin Mamie (group 0)
CS-422: Database Systems (Spring 2020)
École Polytechnique Fédérale de Lausanne, Switzerland
2020–05–19

I. Introduction

The task of this project is to discover different operators using Apache Spark and evaluate their performances against existing and/or brute-force implementations. We implement a ROLLUP operator, a theta join operator, and finally locality-sensitive hashing for near-neighbor computation.

All times were taken on the 10-node cluster on IC Cluster, using 25 GB of driver memory, 10 GB of executor memory, 4 executor cores with 16 executors in total.

II. Implementation

A. ROLLUP operator

In this section, we compare 2 different ROLLUP operator implementations:

1) A naive ROLLUP implementation.
2) An optimization of the naive ROLLUP implementation using parent/child relations.

We evaluate their performances over 3 different input sizes and 6 numbers of attributes.

The optimized implementation was originally predicted to have better results than the naive one, since it reuses results that were already computed. However, as can be seen on figures 1, 2, and 3, this is not the results we get with our implementation. Indeed, the optimized version does not scale well with the number of attributes compared to the naive implementation, which is exactly the problem this implementation is supposed to solve. It also gets worse for bigger inputs.

This is explained by the fact that the optimized implementation has to *wait* on previous GROUP BYs to finish before using their results to compute the rest. The different computation stages – for each group of attributes – can thus not be executed in parallel, which is the case in the naive ROLLUP operator. We see that the trade-off parallelization/using precomputed data is clearly skewed in favor of parallel execution on the cluster, which is why the "optimized" version is much slower than the naive one.

B. Theta join

We test the theta join operator using different numbers of reducers, which we call `r`, as in the project description. The greater the `r`, the more fine-grained the horizontal and vertical zones are. Therefore, a greater `r` – up to a certain point – means fewer comparisons are executed, which reduces the execution time. This is true until the number of reducers `r` is greater than the number of inputs (see figures 4 and 5) or bottlenecked by the number of executors used during computation (see figures 6 and 7).

We can see on the graphs that the *sweet spot* of the amount of partitions/registers shifts to a higher number the bigger the input, up to a certain point. We also observe that our implementation is always faster than the canonical way of computing the join, even for big inputs. As the size of the input grows, a small `r` yields less performing results and explodes in computation time. This is explained by the overhead of the method, which explodes when inputs are big and the number of partitions is small.

C. Near Neighbor query processing

In this section, we compare 3 different near-neighbor (NN) query processing implementations:

1) An exact NN operator, which compares all attributes between all pairs of elements from the first and the second datasets to guess the nearest neighbor (given a certain threshold).
2) A combination of locality-sensitive hash functions (LSH), which use MinHash signatures, to approximate the exact NN computation.
3) A combination of LSH operators with prior broadcasting of the MinHash signature of the provided data.

The Exact NN operator corresponds to a function executing the cartesian product between the original data and the query, given a certain Jaccard threshold – which is 0.3 in our tests. The LSH implementation first combines the data having the same MinHash signature and then associates the results with the MinHash signature of the query.

1) Time analysis: The number of computation is $\mathcal{O}(NM)$ in exact NN where N and M are the number of tuples of the first and second dataset respectively. In the LSH implementation, we compute once the MinHash of each element for each dataset, and we then join them. Since the complexity of the MinHash algorithm is $\mathcal{O}(\text{number of different keywords})$, we have at the end an global complexity of $\mathcal{O}((N+M) \times \text{number of different keywords})$.

Thus, we can observe that the cartesian and LSH implementations have, *broadly speaking*, similar results (exact NN is still slower) for a small dataset (see figure 8). However, when the datasets are bigger, the number of operations explodes for exact NN but slowly grows for LSH, as expected (see figures 9 and 10). The broadcast implementation of the LSH algorithm evaluates queries faster for the small and medium datasets than the one without broadcasting, since the work is performed upstream. However, evaluations on the big dataset

TABLE I
PRECISION AND RECALL OF DIFFERENT LSH CONSTRUCTIONS, FOR QUERIES 0 TO 2 ON THE SMALL DATASET. A THRESHOLD OF 0.3 IS USED ON THE EXACT NN COMPUTATION.

Construction		Q0	Q1	Q2
Base	Recall	91.9%	90.8%	91.8%
	Precision	69.9%	67.7%	68.5%
AND, 8×Base	Recall	87.2%	86.7%	88.8%
	Precision	98.1%	96.7%	97.4%
OR, 8×Base	Recall	96.8%	95.2%	96.9%
	Precision	41.2%	41.0%	41.9%
AND, 8×OR, each 8×Base	Recall	94.0%	92.6%	93.3%
	Precision	78.1%	75.8%	74.9%

does not confirm this trend – exact NN has timed out, we cannot compare the *base* constructions times with it. The fact that the exact NN times out is perfectly logical, since it is far more complex than a simple *base* construction.

2) Performance analysis: We can see in table I that the *base* construction has poor results in terms of precision. In order to improve these results, we try different constructions mixing *OR* and *AND* constructions. As stated in the lecture *Compression & Privacy*, *AND* constructions reduce the number of false positives (*FP*), but increase the one of false negatives (*FN*). *OR* constructions reduce the number of false negatives (*FN*), but increase the one of false positives (*FP*). The combination of the two constructions reduces the number of false answers.

We test several constructions, as can be seen in table I. Choosing between theses types of constructions induces a trade-off between precision and recall, which are defined as follows (with *TP* = true positives):

$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{Recall} = \frac{TP}{TP + FN}$$

We can see that when we combine multiple *base* constructions using an *AND* construction we reduce the recall since we have more false negatives but we increase our precision, having fewer false positives.

We observe the opposite phenomenon for the combination of multiple *base* constructions using an *OR* construction. Indeed, we increase our recall since we have fewer false negatives and we reduce our precision, having more false positives.

We finally got a better recall and a better precision with a composition of *OR* and *AND* constructions, which makes sense since we have fewer false positives and negatives.

To satisfy the given requirements, query 0 can use an *AND* construction combining 2 *base* constructions (recall of 89.5% and precision of 91.9%), query 1 an *AND* construction combining 13 *base* constructions (recall of 72.3% and precision of 99.0%), and finally query 2 can simply use a *base* construction (as seen in table I)[1].

[1]Requirements are: for query 0, recall of 83%, precision of 70%, for query 1, recall of 70%, precision of 98%, and for query 2, recall of 90%, precision of 45%.

ROLLUP GRAPHS

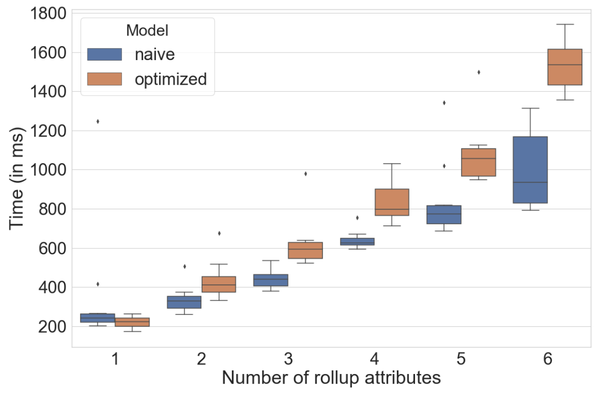

Fig. 1. Execution times of the the ROLLUP operator with the small dataset and different numbers of grouping attributes.

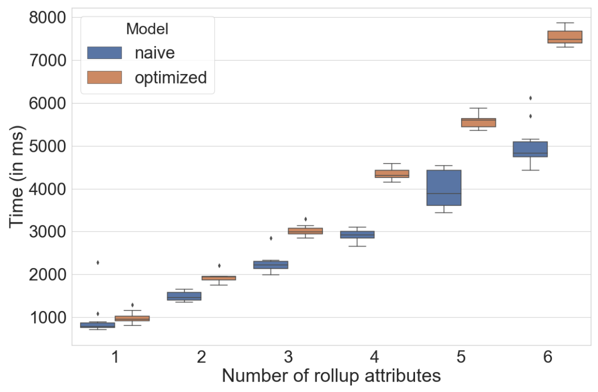

Fig. 2. Execution times of the the ROLLUP operator with the medium dataset and different numbers of grouping attributes.

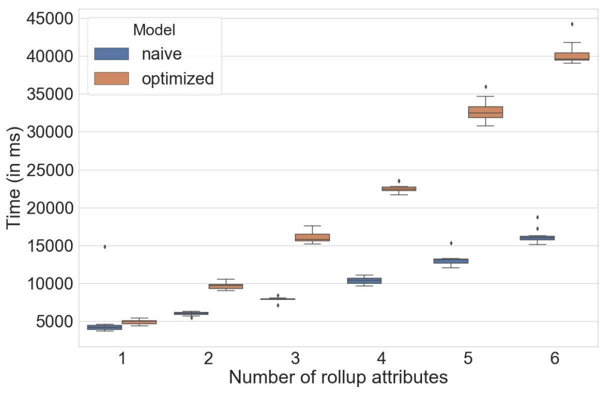

Fig. 3. Execution times of the the ROLLUP operator with the big dataset and different numbers of grouping attributes.

THETA-JOIN GRAPHS

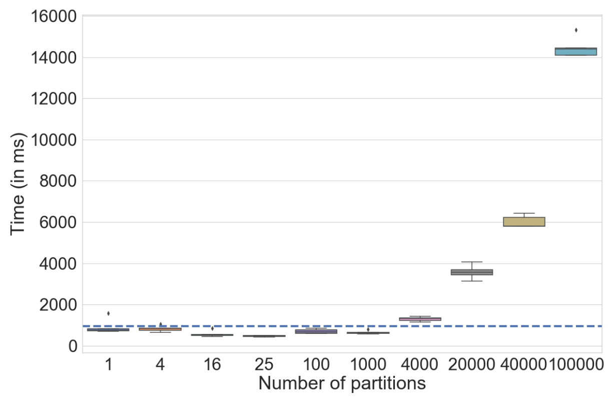

Fig. 4. Execution times of the theta join operator with a dataset of size 1000. The blue dashed line corresponds to the cartesian based theta join for the same dataset (standard deviation of 24.1 ms).

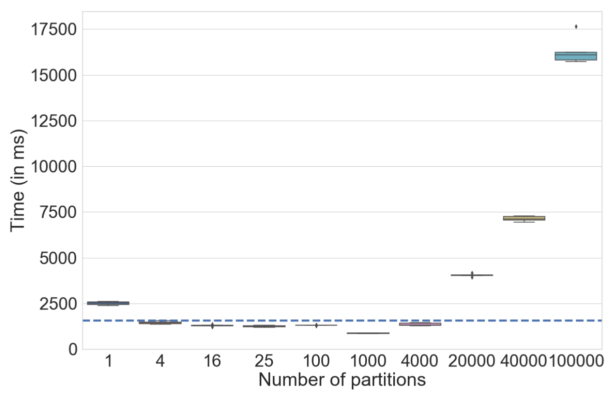

Fig. 5. Execution times of the theta join operator with a dataset of size 4000. The blue dashed line corresponds to the cartesian based theta join for the same dataset (standard deviation of 104.1 ms).

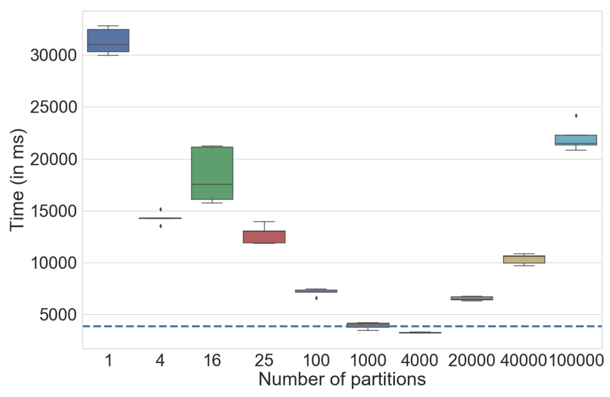

Fig. 6. Execution times of the theta join operator with a dataset of size 16000. The blue dashed line corresponds to the cartesian based theta join for the same dataset (standard deviation of 91.3 ms).

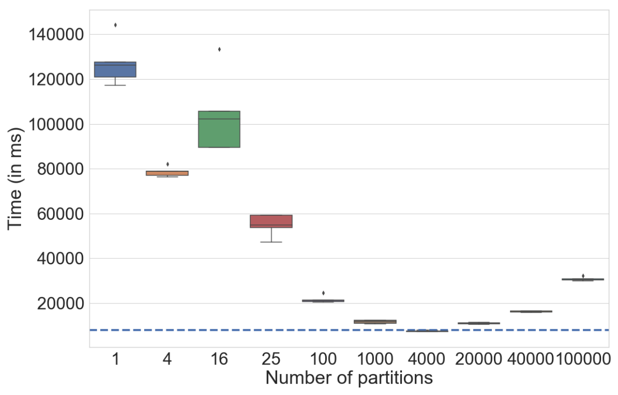

Fig. 7. Execution times of the theta join operator with a dataset of size 32000. The blue dashed line corresponds to the cartesian based theta join for the same dataset (standard deviation of 334.4 ms).

LSH GRAPHS

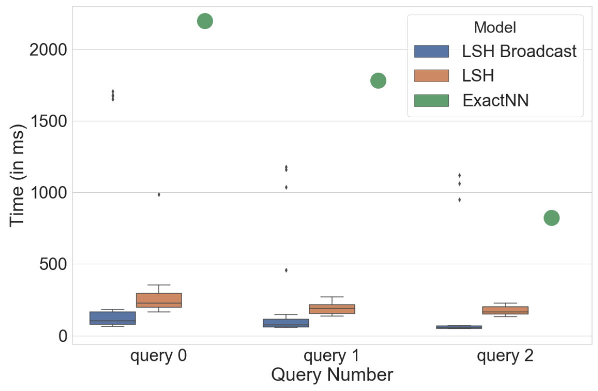

Fig. 8. Evaluation times of near-neighbor operators with the small dataset. Small boxes are redrawn as points so that their color is visible.

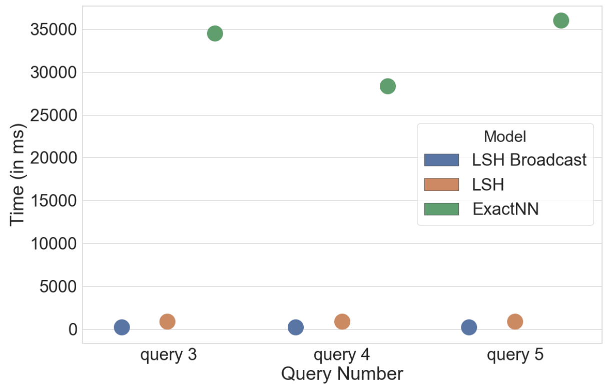

Fig. 9. Evaluation times of near-neighbor operators with the medium dataset. Small boxes are redrawn as points so that their color is visible.

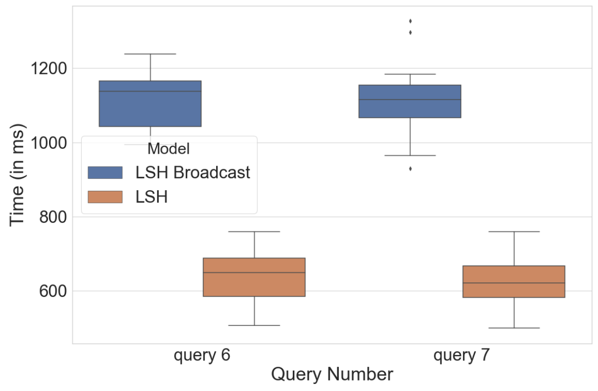

Fig. 10. Evaluation times of near-neighbor operators with the big dataset. Small boxes are redrawn as points so that their color is visible.