# Database Systems: Project 2 Report

Marshall Cooper - Andrea Piccione

In this project, we implemented a two-phase CUBE operator on Spark RDDs as well as a similarity ClusterJoin. For our CUBE operator, we compared its performance in a variety of tests versus a more naïve MapReduce implementation with only a single phase. We tested over three sizes of the same data set and varied the number of reducers as well as the CUBE dimension |D| (*number of columns used for grouping)*. We intentionally chose CUBE columns to allow for some aggregation to occur, meaning that we did not, for example, choose any column that uniquely identifies the data set (*such as a primary key or comment field*). The following summarizes the results of testing:

Runtime (s) for CUBE algorithms on 6008 tuples, 16 reducers

| | Dimension \|D\| of CUBE | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| Naïve | 0.0767 | **0.0647** | **0.2106** | **0.3599** | **0.7863** | 2.6452 |
| 2-Phase | **0.0625** | 0.0694 | 0.3314 | 0.4537 | 0.8956 | **2.0827** |

Runtime (s) for 2 Dimension CUBE, 16 reducers

| | Number of tuples | | |
|---|---|---|---|
| | 6008 | 600572 | 6001171 |
| Naïve | **0.2849356** | **0.6053815** | 9.1548128 |
| 2-Phase | 0.3876628 | 0.6597307 | **6.1207095** |

Runtime (s) for 3 Dimension CUBE on 600572 tuples

| | Number of reducers | | | | |
|---|---|---|---|---|---|
| | 1 | 4 | 8 | 16 | 32 |
| Naïve | **8.3122** | **3.4269** | **2.2873** | **1.9909** | **1.8037** |
| 2-Phase | 23.788 | 10.378 | 4.64 | 3.1343 | 2.1077 |

The first interpretation that we make for these results is that in simple cases, with a low |D| and a small size data set, the naïve operator substantially outperforms the two-phase optimization. For the two-phase algorithm, there is an additional mapper side combine phase that is meant to lower the workload on reducers, but we believe that the extra execution of this phase, for small inputs, is not offset by the reduction in workload further down the stream. We observe that on larger data sets, the two-phase algorithm suffers less in performance than the naïve implementation. More drastically, we note that the two-phase algorithm deals substantially better with higher |D|. This appears to be the most drastic performance benefit for the two-phase algorithm, as even on the small data set, it handled larger CUBE dimensions more easily. Finally, we note that increasing the number of reducers seems to benefit both algorithms up to a point. For the case of the two-phase algorithm, which has multiple reduction steps, we see that having a small number of reducers is especially penalizing. Prior to testing, we had the intuition that the naïve algorithm would perform better on less computationally intensive test cases because it is a simpler algorithm with fewer steps. Ultimately we were surprised that the optimized version only outperformed the naïve algorithm when either the number of tuples or the dimension of the CUBE was very large. Nonetheless, this indicates that is a more scalable algorithm.

In the second task of this project, we implemented a similarity join operator using an optimized ClusterJoin algorithm [1]. We carried out a battery of performance tests that aim at assessing the validity and the scalability of the ClusterJoin algorithm in comparison with a reference implementation which takes advantage of the Cartesian product to perform similarity checks among each possible combination of strings in the chosen column of the table.

We performed one series of tests to see the influence the number of anchors had on the ClusterJoin, and additionally performed a series of tests to see how the ClusterJoin would compare versus the Cartesian product algorithm on differently sized samples of the input data. We sampled 10%, 20%, 50% and 100% of the original dataset (10k inputs) for testing. For all tests, the distance threshold was set to 2 as a constant. We made the choice to control this feature because it doesn't directly affect the time elapsed for the computation of the result.

ClusterJoin Runtime (s)

| | | Number of Anchors | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 4 | 8 | 16 | 32 |
| Input size | 1000 | 25.2808 | 24.7078 | **14.692** | 21.5589 | 23.3682 | 15.0509 |
| | 2000 | 50.6235 | 60.8314 | 53.2724 | 50.2225 | **46.8363** | 51.9872 |

Runtime (s) Comparison between ClusterJoin and Cartesian Product-based Join

| | Input size | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1k | | | 2k | | | 5k | 10k |
| | Number of Anchors | | | Number of Anchors | | | Anchors | Anchors |
| | 1 | 4 | 16 | 1 | 4 | 16 | 4 | 4 |
| ClusterJoin | 25.280752 | 14.692032 | 23.368197 | 50.204785 | 53.255414 | 53.463841 | **295.05756** | **1136.5164** |
| Cartesian | **13.99230057** | | | **46.83632267** | | | 358.26465 | 1201.0249 |

We can first notice that, as a general trend for low dimensional input sizes, increasing the number of anchors makes the ClusterJoin algorithm faster. This was an expected behavior predicted by the authors of the algorithm [1], who state: "for low dimension spatial data, having a larger number of anchors will likely make the “closest” anchor even closer, thus improving the effectiveness of our filtering procedure". On the other hand, that authors also state that "in high dimensional space, adding more anchors does not significantly reduce the average distance to the “closest” anchor." This is reflected by our observed results that show how the performance improvement becomes significantly less evident already by sampling 20% of the original input dataset.

Varying the size of the dataset provided us with an interesting overview of the scalability capabilities of our implementation. In fact, even though the Cartesian implementation evidently outperforms the ClusterJoin algorithm for small data sizes, the efficiency of the optimized algorithm is noticeable and can be appreciated for larger input sizes where it is definitely faster than computing the Cartesian product for each string pair.

Therefore, we believe these observation support the scalability of the ClusterJoin algorithm. We also noticed that the difference in performance between the algorithms, even on small data sets, was very small if a reasonable number of anchors was used.

In conclusion, both algorithms we implemented showed clear signs of increased scalability over their more simplistic counterparts. In the case of the CUBE operator, we were surprised how large the scale of parameters had to be in order for the optimized algorithm to outperform its counterpart. In the case of the ClusterJoin implementation, we were surprised that the ClusterJoin was proportionally competitive for even small inputs and outperformed its counterpart even for a relatively small (10k) selection of strings.

**Reference**

[1] Sarma, A., He, Y., Chaudhury, S. ClusterJoin: A Similarity Joins Framework using Map-Reduce, 2014