EPFL CS-422
Project I: Data Layouts and Execution Models

Murat Topak

March 2019

In this project, different data layout and execution models have been implemented to observe their performance on 3 types of queries mentioned in Task 3.

For the SELECT type of queries, the best performance was given by **column-at-a-time execution model with column-storage**. Although early-materialization was doing fine with not so selective predicates, late-materialization was the winner for me when there were more predicates, cascaded one after the other. Indeed, with more columns and more conditions, we know from theory that delaying the tuple construction as much as possible performs better due to caching mechanisms. With more conditions, I was able to reap the benefits of late-materialization.

For the JOIN type of queries, the best performance was given by **tuple-at-a-time engine with row-data layout**. This seems reasonable since join operates on all the columns and not just a particular one when emitting the results. Hence, although column-at-a-time engine with column-storage was promising, it wasn't the best-performing. Late materialization was effective but too much effort put into stitching the columns so it didn't pay off.

The last type of queries with the aggregate functions is the one use case where **column-at-a-time model with column storage** excels. This makes sense since aggregates operate on one column to compute an average, min, max, etc. so a column can be effectively read into memory while caching is utilized at the same.

At this point, I think the reason PAXStore, being a hybrid solution to storage problem, didn't work out as expected is the fact that tuple-at-a-time engine retrieves the data row by row from the storage, which leads to an overhead of tuple construction from the column-stored data in the mini pages. There is also time spent to calculate the page number and offset in the mini-page given the row number so tuple-at-a-time engine doesn't make use of the hybridity of the PAXStore. Also, vector-at-a-time engine implementation has been inefficient for me since each time it had to create a vector of N tuples to pass from the column-stored data.