Statistic Collection During Think Time in Modin

Project 4: Connor McMahon and Jonathan Shi

Problem

Modin is a drop-in replacement for Pandas that allows for convenient parallelization of dataframe transformations across multiple cores. Traditional database systems collect data distribution statistics to improve query performance. Modin executes all operations *eagerly* (a query is run without any changes as soon as a user submits it), so it currently does not support such optimizations. This project adds support for *lazy execution* in Modin, allowing for query planning optimizations and background collection of statistics.

Background

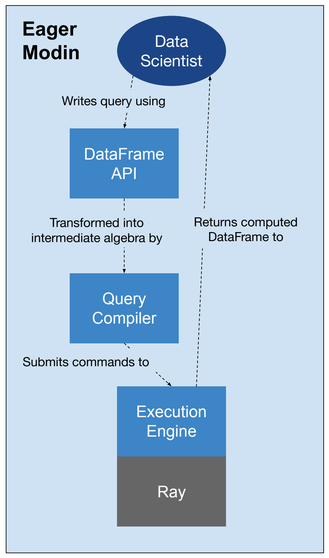
In Pandas and Modin, users can apply various transformations to manipulate *dataframes* (2-dimensional table structures similar to SQL tables), and often do so interactively in Jupyter Notebook while prototyping algorithms.

Query planners can take advantage of statistics about the distribution of data to reorder operations to improve performance, usually by minimizing the size of tables produced by intermediate computations. In relational database systems, where data typically persists for long periods of time, database administrators will periodically run commands that update these statistics. By contrast, data in Jupyter notebooks is short-lived and iterated over frequently. However, this context also presents a unique opportunity: users often spend time inspecting the results of short snippets of code; we can use this *think time* to run statistics collection in the background without impacting performance from the user's perspective.

Solution

To implement statistics collection during "think time", we wrote a Jupyter notebook extension that utilizes the IPython kernel's event manager to register callbacks before and after each cell is executed. Whenever a dataframe is instantiated, it is queued up for statistics collection. After a cell finishes running, the Jupyter extension spawns a thread to collect statistics, and then suspends it when the next cell is started to free up the CPU.

In order to use these statistics in the first place to perform query optimizations, we also implemented a minimal lazy execution engine in Modin. To this end, we extended Modin's QueryCompiler class, which transforms user-facing dataframe operations into actions on a distributed system backend like Ray.



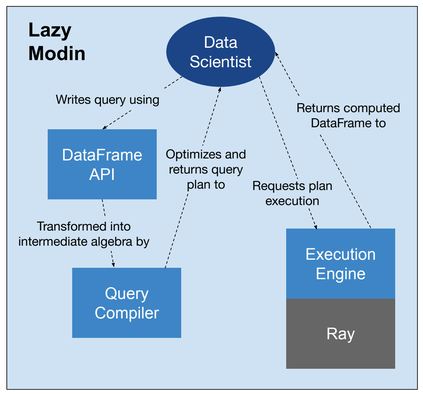


Figure 1: Differences in high-level architecture between eager and lazy Modin.

Our custom QueryCompiler avoids immediately dispatching commands to Ray and instead returns a QueryPlan object to the user. An actual dataframe is produced only when the user invokes the .execute() method of the QueryPlan, which gives us a chance to reorder operations.

Results

All benchmarks were run on data from the New York City Taxi and Limousine Commission's trip records for July 2021, duplicated several times to increase dataset size. All tests were run on AWS r5.x2large instances. (8 CPUs, 64GB RAM).

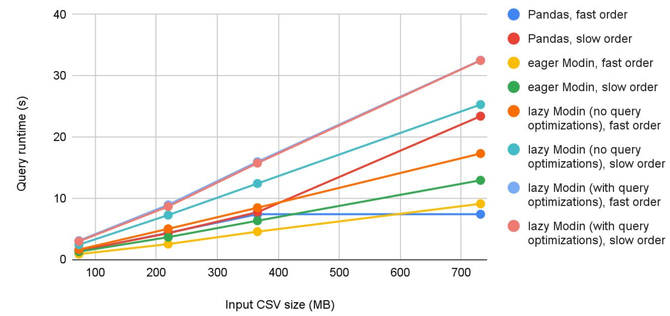


Figure 2 (above): Benchmarks of filtering NaN values for two different columns of the same dataframe. The "slow" tests filter fewer values in the first operation.

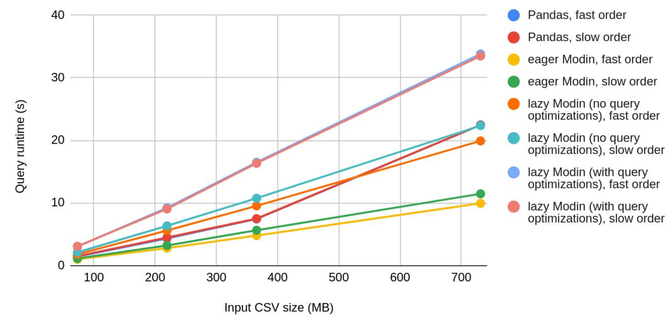


Figure 3: Reordering of two redundant filtering operations (filtering all values in a column >100, then those >200). The "slow" tests filter values >100 first.

There are still significant performance issues due to memory overhead from histogram storage to be worked out, though the query optimizer is able to eliminate the difference between fast and slow orders by reordering operations. Before our final report, we also plan to implement a few more query optimizations, such as reordering dataframe joins based on cardinality. We also intend to simulate a typical Jupyter workload, and show that statistics collection never exceeds think time.