Leveraging User Think Time to Enable Query Optimizations

Jonathan Shi and Connor McMahon
University of California, Berkeley
{jhshi,mcmahon}@berkeley.edu

ABSTRACT

Pandas is a popular open source data analysis tool built for the Python programming language, and has gained much popularity due to its ease of use. Queries in Pandas are performed in a single-threaded, eager fashion, meaning that operations are executed as soon as they are issued, and control is not returned back to the user until the execution is complete. Modin is a framework that mimic's the Pandas API, and lets users parallelize Pandas operations without substantial changes to their codebase: they simply need to change one import statement to take advantage of Modin's optimizations. To improve Modin's performance even further, we look to traditional database systems which collect data distribution statistics to improve query performance. Modin is frequently used in Jupyter notebooks, an interactive environment in which users execute a set of queries, then spend time analyzing the intermediate results to determine the next set of operations to perform. We can leverage this time that users spend analyzing their data (referred to as "think time") to collect statistics that will enable query optimizations. This project adds support for lazy execution in Modin, allowing for query planning optimizations and background collection of statistics, and lays the groundwork for realizing opportunistic execution in Modin.

1 INTRODUCTION

Pandas is a widely-used Python library that allows users to easily manipulate and visualize data. One of the most popular components of the Pandas API is the dataframe, a 2-dimensional table-like structure (analogous to a SQL table).

Modin [14] is a drop-in replacement for Pandas that addresses two of its major weaknesses:

1) Pandas does not natively support parallelizing computations on very large DataFrames. Modin allows computations to be split up and scheduled on multiple cores or machines.

2) Pandas has a very large API, and many operations are redundant. Modin provides a concise set of basic operators that higher-level computations are reduced to, and dataflow analyses then make these computations as efficient as possible.

In traditional database literature, statistics about the distribution of a dataset (often in the form of histograms) can be used to optimize query planning by providing a better estimate of how long an operation takes to execute [16]. Modin does not yet support these optimizations, and our goal is to construct such histograms during think time to enable this.

Within Modin and Pandas, users apply various transformations to manipulate dataframes, and usually do so interactively in Jupyter notebooks [3] while prototyping an algorithm. Users spend a decent amount of time reading the results of intermediate computations to verify the success and accurancy of a computation, and to help them decide which steps to take next. This time spent reading intermediate computations is known in the literature as *think time* [19], and leaves the CPU idle and open for use for other computations. In the context of a database system like Modin, the unused CPU cycles during think time open up the opportunity to run statistics collection and query optimizations at no additional runtime cost to the user.

In the initial publication of Modin, Petersohn et al. leave the idea of performing statistics collection and query optimization during user think time as an idea for future work [14], and Xin et al. later measure the potential for significant performance enhancements that arise from leveraging think time for this purpose [19]. These prior works also outline the paradigm of *opportunistic evaluation*, where computations are deferred to run in the background of user activity, and prioritized when a user needs to see their results.

Our work takes the first step in realizing an opportunistic evaluation framework by implementing statistics collection and query optimization during think time, within a subset of the Modin API.

In concrete terms, this work has three primary goals:

(1) Statistics collection and query optimization should be run in the background of an interactive environment at no additional cost to responsiveness.
(2) Query optimizations should allow possibly sub-optimal code written by users to execute in approximately the same amount of time as an equivalent hand-optimized query.
(3) Statistics-based query optimization should allow queries to run faster than those of baseline Modin, which does not support any form of statistics collection.

2 BACKGROUND

DataFrames. The core unit of data in Modin in Pandas is the *DataFrame*, a simple 2-dimensional table-like structure. DataFrames enable Python programmers and others who are unfamiliar with database systems to easily perform analysis on large data sets. They provide a more intuitive way of working with data compared to SQL, as they do not require familiarity with the schema to perform queries. Moreover, DataFrames also provide an easier debugging process than that of SQL queries because users can easily flatten out operations when trying to analyze intermediate steps, whereas SQL requires the users to write whole new queries to do the same [18]. Additionally, DataFrames support both relational and linear algebra operations, whereas SQL is only made to support relational operations.

Pandas supports a wide array of relational and linear algebra operations. These operations were developed by dozens of contributors who have expanded the Pandas DataFrame API to support more than 200 operations [2]. Many of these operations are redundant and therefore create an unnecessary burden for users who

CS262A, Graduate Computer Systems, UC Berkeley
2021.

must sift through this large API to determine which operation will best suit their needs.

This massive set of operators makes optimizing all of the API calls an arduous and redundant task. Modin reduces the redundancy of optimizing the operators by consolidating operators that perform the same operations together, and representing them in terms of a simpler DataFrame algebra, which we further elaborate on in Section 4.

Although most relational operators are easily parallelizable, most Pandas operators are executed by a single thread. To remedy this, Modin provides three levels of parallelization to support different operations: row-based, column-based, and block-based partitioning. Preliminary analysis compared the performance of Modin to pandas using datasets ranging from 20 GB to 250 GB with up to 1.6 billion rows on the following operations: map, group-by (n), group-by(1), and transpose operations. Modin performed 12x faster for map operations, 19x faster for group-by(n) operations, and 30x faster for group-by(1) operations compared to pandas. Pandas was unable to perform the transpose on the smallest dataset after two hours, but Modin was able to transpose even the largest dataset in the study in less than 200s [14], thus demonstrating the effectiveness of the distributed DataFrame design.

The importance of responsiveness. Prior work has shown that it is crucial for systems to provide feedback to users as quickly as possible. Even a small additional delay of 500ms can have substantial negative affects on a user's ability to efficiently interact with data [11]. Consequently, it is crucial to find additional means of optimizations to ensure that improve the performance of user code without sacrificing interactivity and response times.

Traditional database optimizations. 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 [16]. We discuss some such optimizations in Section 5.3.

Opportunistic execution. In contrast to relational databases, where tables persist for a long time and are infrequently updated, data in Jupyter notebooks (where Modin and Pandas operations usually take place) is inherently short-lived and iterated over frequently. Jupyter notebooks are split up into execution units ("cells") which typically contain only a few lines of code. After executing each cell, users tend to spend a considerable amount of time analyzing the results of the cell to verify the computations produced the expected outcome and to determine the next steps they should take. Xin et al. empirically examined the distribution of user think times, and based on a corpus of 210 notebooks from an intermediate-level data science course, found the 75th percentile of think time to be 23 seconds [19]–a staggeringly large figure compared to the time scales at which modern CPUs operate.

The possibilities engendered by this considerable amount of free CPU time forms the basis of what Petersohn et al. refer to as the *opportunistic evaluation* paradigm [14]. This comes in contrast to an *eager evaluation* system, where computations are performed as soon as they are issued, as well as a *lazy evaluation* system, where all computations are deferred to be manifested only when the user explicitly requests the result. Under an opportunistic evaluation framework, most computations are initially left deferred, but computations are scheduled and run in the background instead of waiting for an explicit request by the user. In the case of Modin running in a Jupyter notebook, think time would serve as the "background" during which queries are run.

Pandas and Modin both execute all operations eagerly, meaning that queries are executed as soon as the user runs the Python code that define them. Control does not return to the user until the query has finished executing. Other DataFrame-like systems use lazy execution [6, 7] in which the query is not executed until its value is needed. This allows for the prioritization of queries that are needed sooner which speeds up the response time to the user.

Our work modifies Modin to fall under a mix of lazy and optimistic evaluation behaviors. Statistics collection and query optimization are run during think time, thus representing opportunistic execution, whereas the results of user queries are returned as DataFrame "future" rather than a concrete object, thus representing lazy execution. Further details are laid out in Section 5.

3 USER WORKFLOW

To demonstrate what a typical user's workflow might look like, consider a data scientist examining a dataset of New York City taxicab data [5] (also used in our benchmarks). In this example, a user wants to find the average time length of taxi rides in a given dataset. The user's likely behavior is as follows:

(1) The user will first read the `.csv` file containing the trip information for the given month
(2) The user will inspect the contents of the first few rows of the data. This inspection leads the user to realize that some of the drop-off locations and pick-up locations are missing (represented in the CSV as `NaN`).
(3) These entries cannot be used to compute the average trip distance, so the user will filter them out of the data set.
(4) The user will again inspect the data. It appears that the computations filtered out the data as expected, so the user can proceed to calculating the average ride time.

A snippet of Python code executing this workflow in Jupyter is presented in Figure 1.

This example code presents two instances of think time. In the first instance of think time (between cell B and cell C), statistics could have been computed on the `tripdata` DataFrame, which would enable a query optimizer to later reorder the commands in cell C to provide faster feedback to the user. For example, if there were more `NaN` pick-up locations (represented by the `PULocationID` column) compared to drop-off locations (`DOLocationID`), then it would have been faster to filter out the `NaN` pickup locations before filtering out the `NaN` drop-off locations since the first filter produces a smaller intermediate result. In the second instance of think time (after cell C), statistics can be collected on the newly computed DataFrame to enable query optimizations on later computations.

```
# Cell A: Read CSV into DataFrame
tripdata = pd.read_csv("fhv_tripdata_2021-07.csv")
```

```
# B: Inspect DataFrame to sanity check data
tripdata.head()
```

	dispatching_base_num	pickup_datetime	dropoff_datetime	PULocationID	DOLocationID
0	B00014	2021-07-01 00:31:02	2021-07-01 01:10:00	NaN	NaN
1	B00037	2021-07-01 00:16:15	2021-07-01 00:24:33	NaN	71.0
2	B00037	2021-07-01 00:39:00	2021-07-01 00:45:31	NaN	188.0
3	B00037	2021-07-01 00:55:26	2021-07-01 01:09:41	NaN	89.0
4	B00037	2021-07-01 00:05:22	2021-07-01 00:27:11	NaN	17.0

```
# C: Remove NaN values, inspect result
dolocation = tripdata["DOLocationID"]
tripdata = tripdata[dolocation.notna()]
pulocation = tripdata["PULocationID"]
tripdata = tripdata[pulocation.notna()]
tripdata.head()
```

	dispatching_base_num	pickup_datetime	dropoff_datetime	PULocationID	DOLocationID
22	B00254	2021-07-01 00:30:20	2021-07-01 01:13:08	87.0	265.0
23	B00254	2021-07-01 00:15:32	2021-07-01 01:08:28	13.0	265.0
24	B00254	2021-07-01 00:04:01	2021-07-01 00:16:48	13.0	170.0
25	B00254	2021-07-01 00:30:47	2021-07-01 00:52:44	230.0	51.0
26	B00254	2021-07-01 00:19:58	2021-07-01 00:37:09	125.0	265.0

```
# D: Perform calculation
(tripdata["dropoff_datetime"] - tripdata["pickup_datetime"]).mean()
```

```
Timedelta('0 days 04:18:22.283310007')
```

Figure 1: An example workflow in Jupyter notebook.

4 AUGMENTING MODIN'S ARCHITECTURE

Our project replaces Modin's eager query compiler with a lazy one, with further additions to the API to perform statistics collection and query optimization. The total changes affected 702 lines of code–a nontrivial amount, but still relatively small compared to the 30,000 LOC of the entire Modin codebase [14]. In this section, we describe the architecture and data model of the original, eager version of Modin, and how it differs from our implementation (Section 4.1). We also discuss our changes to the Modin DataFrame API to support for lazy evaluation and statistics collection (Section 4.2), and conclude by describing how our changes fit in to the Jupyter data science workflow (Section 4.3).

4.1 Modin High-Level Architecture

Fundamentally, the Modin library is designed to bridge the gap between the DataFrame API familiar to data scientists, and execution engines like Ray and Dask that allow for parallel execution of DataFrame operations. To this end, Modin DataFrames expose a user-facing abstraction almost identical to that of their Pandas counterparts, and a query compiler then translates all high-level query operations to an underlying DataFrame "algebra." This algebra is a set of fundamental operators, including Selection (eliminating rows), Join (combining two frames), and Map (applying a function to every row) [14]. This simpler collection of operators, compared

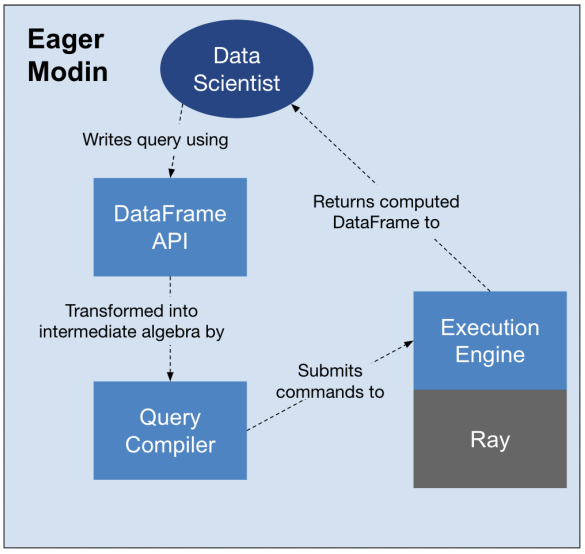

Figure 2: Simplified architecture diagram for eager Modin.

to the vastness of the Pandas API surface, makes query planning and optimizations easier to reason about. Note that some of the algebra operators outlined in the original Modin paper [14] are absent or changed in implementation; these differences are described by Petersohn in [13].

The data underlying the DataFrames themselves are internally partitioned to the parallel execution framework (generally Ray or Dask), and the intermediate DataFrame algebra operations are then translated to a sequence of commands issued to the execution framework. This flow of data is represented in Figure 2.

Our changes to Modin replace the existing eager query compiler with a lazy one. Consequently, instead of user code blocking until the execution engine has produced the desired DataFrame, our version of Modin returns a query plan (possibly optimized) to the user. It is then up to the user to request its execution by explicitly invoking the query plan's `execute` method. This architecture is depicted in Figure 3.

Importantly, the Modin-internal representation of all DataFrames is immutable; destructive functions in the API surface are actually implemented as copy-on-write operations under the hood. This means our lazy framework has no trouble handling in-place updates.

In theory, we could also have chosen to make execution lazy at the level of the execution engine, as it is the component of the system directly responsible for the actual calculations being performed. In fact, computations here are already "deferred" in a sense, as they must enter a scheduling queue before execution, at which point operations may be reordered and optimized. However, implementing optimizations and lazy evaluation in the execution engine would have required too many API changes (since there would be no easy way for a user to request immediate execution of a particular plan), and lose semantic information about operations that exists at the level of the query compiler.

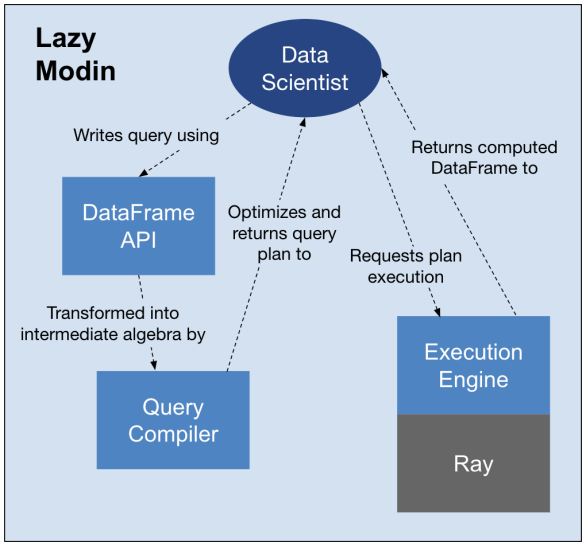

Figure 3: Simplified architecture diagram for lazy Modin.

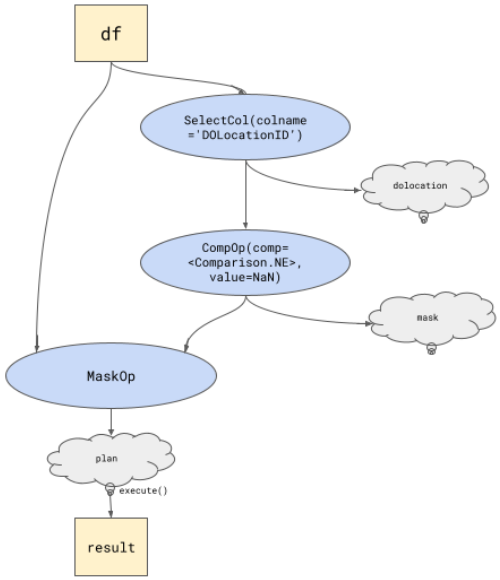

Figure 4: Algebra representation of code to remove all `NaN` values from a DataFrame. Intermediate query plans are shown in gray, operators in blue, and concrete DataFrames in yellow.

4.2 Modifying the Modin Algebra and API

In the interest of simplicity, our lazy query compiler supports a modified subset of Modin's internal algebra rather than strictly corresponding to the original operations. These modified operators, and their correspondence in the original Modin algebra, are outlined in Figure 5. Each of our operators is translated into execution engine commands in essentially the same way as the corresponding operator in original Modin. Though we did not evaluate what portion of the DataFrame API these operators cover, our experiments anecdotally show that they encode a fair amount of computations that might be performed by a data scientist.

Though our modifications to Modin support all the operators described in Figure 5, we did not implement optimizations for all of them (see Section 5.3 for further discussion).

As an example of the relationship between the DataFrame API and our internal algebra, Figure 4 gives a translation of the following snippet of DataFrame code from our benchmarks, which filters out all values in the dataset where the value of the field `DOLocationID` is `NaN`:

```
df = pd.read_csv("fhv_tripdata_2021-07.csv")
dolocation = df["DOLocation_ID"]
mask = dolocation.notna()
plan = df[mask]
result = plan.execute()
```

As seen in the sample code snippet, the manner in which users interact with DataFrames in our lazy framework is mostly the same as that in eager Modin. The only difference is the returned `QueryPlan` instead of a computed DataFrame, which requires the user to call `execute` (and optionally, `optimize`) to complete the query. This `QueryPlan` can also be conceptualized as a DataFrame "future": it is not yet a DataFrame, but a user may use it to produce one.

4.3 Statistics Collection API

To enable statistics collection, we maintain a queue of DataFrames that are created by the user, either directly through the `pd.DataFrame` constructor, indirectly through `pd.read_csv`, or as the final result of a sequence of computations. We add a line of code to the DataFrame constructor that pushes it to the statistics queue whenever it is called in such a context.

We expose a `stats_manager` singleton object to the API that is tasked with draining this queue; calling its `compute_next` method will compute histograms for the oldest DataFrame still in the queue, `compute_all` will synchronously compute histograms for every single DataFrame in the queue, and `clear_all` will remove all previously computed histograms, cached query plans, and queued DataFrames. The details of our histogram representation are discussed in Section 5.2. `compute_next` and `compute_all` may also optionally begin performing query optimizations in addition to histogram generation.

User-facing query plan objects (returned by the lazy query compiler) come equipped with an `optimize` method, which will compute and memoize an optimized version of the query plan based on stored statistics; these optimizations are discussed in Section 5.3. If statistics for a DataFrame have not yet been generated, calling `optimize` will not modify the query plan (other literature describes the alternative of assuming all data is uniformly distributed [16]).

5 IMPLEMENTATION

This section describes our modifications to Modin and Jupyter to enable statistics collection and query optimization during think time.

Lazy Modin Operator	Description	Original Modin Operator
Compare	Compares each element in a DataFrame to a given scalar value, setting the corresponding element of the resulting DataFrame to `true` if the comparison is true.	Special case of Map
Filter	Replaces all rows of a DataFrame where some condition is `false`.	Filter
SelectCol	Chooses the column of a DataFrame with a given name.	Special case of Mask
Mask	Removes all rows of a DataFrame where some condition is `false`.	Special case of Mask
InnerJoin	Combines the columsn of two DataFrames based on some common key.	Special case of Join
Mean	Aggregation function that computes the arithmetic mean of every column in the DataFrame.	Special case combination of Map and Reduce

Figure 5: Modified subset of the Modin DataFrame algebra.

5.1 DataFrames and Query Plans

As alluded to earlier in Section 4.2, the internal representation of Modin DataFrames remains largely unchanged under our frameworks. The primary difference is our lazy query compiler returns a `QueryPlan` object to users rather than a computed DataFrame.

In Modin, every DataFrame object has a unique associated `QueryCompiler`; we further add a `QueryPlan` to each `QueryCompiler` to represent the sequence of operations built up so far. Together, the `QueryCompiler` and `QueryPlan` represent the sequence of algebraic operations necessary to complete a given query.

A `QueryPlan` is represented as a tuple (immutable list) of DataFrame algebra operations. Whenever the user adds a new operation to a query via the DataFrame API, a new `QueryCompiler` object is created, and the algebra equivalent of the user's desired operation is added to this new query compiler's `QueryPlan`. When the user calls the `QueryPlan`'s `execute` method, we iterate over its list of operations and apply them in sequence to the input DataFrame. Operators like Mask, which combine two DataFrames together, store a reference to a second `QueryCompiler` object containing the plan for computing the second DataFrame argument.

5.2 Statistics

All Modin-internal DataFrames are queued for statistics collection at the end of their creation. When a DataFrame is popped of the statistics queue, we compute the following statistics for each numeric (`int` or `float`) column of the DataFrame:

- The maximum and minimum values of the column.
- Boundaries for n even-width bins, together spanning the minimum and maximum values.
- Count histograms with n bins; each bin contains the number of total elements within its range.
- Distinct count histograms with n bins; each bin contains the number of distinct elements within its range.
- The number of `NaN` items in the column (these do not counted in any of the above bins).
- The total number of elements in the column.

We arbitrarily choose $n = 10$ bins in our implementation.

As these histograms are much smaller than ordinary DataFrames (they have only 1 row and about 30 columns), we store them as vanilla Pandas DataFrames to prevent accidentally recursively invoking statistics collection on a newly-constructed histogram DataFrame. These histogram DataFrames are stored in a global dictionary internal to Modin, which maps (DataFrame, column name) pairs to their corresponding histogram.

The user is not allowed to access these computed histograms directly, as all statistics collection is arbitrated through the `stats_manager` singleton object. Histograms are later read when a `QueryPlan`'s `optimize` method is invoked. Finally, if the user no longer has use for computed histograms, they may call the `stats_manager.clear_all` method to remove all previously computed histograms, cached query plans, and queued DataFrames.

Note that since the Modin representation of DataFrames is immutable (as mutability is emulated through copy-on-write and a layer of indirection), we need not worry about a DataFrame's distribution changing. Its computed statistics are guaranteed to be accurate for as long as it exists, though they may no longer be necessary if the DataFrame has been replaced with a modified copy of itself.

5.3 Optimizations

Here, we detail the query optimizations implemented in our system.

Query plan memoization and intermediate value caching. In Jupyter Notebooks workflows, it is not uncommon for the same operations to be performed multiple times, whether as a sub-query of another, larger algorithm, or as an attempt by a user to confirm the result of a computation. Our creation of an explicit algebra-level `QueryPlan` lets us recognize when two plans are identical, so we choose to aggressively cache the computed DataFrame results of all `QueryPlans`, even those only involved in intermediate computations. We discuss the consequences of this choice in our evaluation (Section 6.3). Note that this optimization is performed whether or not statistic collection is activated.

Operator reordering. We implement parts of a cardinality-based operator reordering algorithm, as outlined by Abraham and Silberschatz [16]. We use histograms to estimate the number of elements in a DataFrame after a query algebra operator is applied; as a general heuristic, our goal is to minimize the number of elements present in intermediate computations. We examine several possible equivalent candidate query plans, assign each a cost based on the number of intermediate elements in each, and choose the candidate with the lowest cost.

Our primary reordering rule seeks to swap the order of Mask and Filter operations within a query plan, as those operations

affect the number of rows in their results. The cardinality of a Mask or Filter is determined by iterating through the operation's parent query plan, identifying the earliest Comparison present, and estimating the number of elements for which this Comparison would return true. Suppose this comparison is against some non-NaN value v, assumed to be within the min/max of the column's range (size estimates are trivial if v is NaN or out of bounds). Let lb and ub be the lower and upper bounds of the bin containing v. The comparison's cardinality estimates are given by the following pseudo-formulas, adapted from Abraham and Silberschatz [16]:

- Equal to v: (total number of elements in the bin containing v) / (distinct number of elements in the bin containing v)
- Less than or equal to v: (number of elements in all bins less than lb) + $(\frac{v-lb}{ub-lb})$
- Inequality, less than, greater than, etc.: follow from complements of above formulas

More robust query plan rewrite rules could also be implemented, given some time. See our evaluation (Section 6) for further discussion.

5.4 Jupyter Extension

Jupyter notebook supports hundreds of kernels in dozens of languages. Developers using Modin most likely use the default IPython kernel to execute their code. It is for this reason that we chose to implement background statistics collection using the IPython kernel [12]. We wrote an IPython extension that utilizes IPython's event manager to register two callback functions: one before each cell is executed and one after each cell is executed.

Recall that DataFrames are queued for statistics collection as soon as they are created. After a cell finishes running, the statistics queue will have been populated with DataFrames created by that cell's code. Our plugin's callback function then spawns a thread to collect statistics by repeatedly calling `stats_manager.compute_next`, which attempts to collect statistics for the next queued DataFrame. When a user requests for another cell to begin execution, the main thread sets a variable to indicate that control should return to the main thread to execute the users commands, thus breaking the loop in the aforementioned callback function. The statistics collection callback checks this exit condition on each iteration of the loop (after completing statistics computation for a single DataFrame), causing the thread to terminate as soon as the exit condition is true. This provides a basic unit of computation that simplifies the process of determining what work has already been completed when the thread begins execution again.

Although our implementation of `compute_next` is not preemptable and may therefore not immediately return to the user to begin executing the next cell, this delay is so small that it should not be noticeable nor affect a user's interactivity with the data. However, we may also choose to schedule background query optimizations alongside histogram generation. This would have a larger potential impact on interactivity; we examine both cases in our evaluation (Section 6).

6 EXPERIMENTAL EVALUATION

Recall the goals of this project, as stated in Section 1:

(1) Statistics collection and query optimization should be run in the background of an interactive environment at no additional cost to responsiveness.
(2) Query optimizations should allow possibly sub-optimal code written by users to execute in approximately the same amount of time as a hand-optimized query.
(3) Statistics-based query optimization should allow queries to run faster than those of baseline Modin, which does not support any form of statistics collection.

The first criteria is difficult to measure; though existing work ([19]) has examined and simulated think time distributions, think time can be dependent on the size and nature of the data involved. Though we have no precise metrics for think time in our tests, we are still able to make an informed guess about our system's efficacy by breaking down the execution times of each component in the query optimization pipeline: data ingestion, query plan creation, statistics collection, query plan optimization, and query plan execution. We examine the runtimes of each of these components.

The second and third criteria are easily tested by our more general benchmarks.

6.1 Benchmarks

Each test was run on three different configurations:

(1) Baseline (eager) Modin
(2) Lazy Modin without query optimizations
(3) Lazy Modin with statistics collection and query optimizations

All our benchmarks come from queries run on the New York City Taxi and Limousine Commission's "For Hire Vehicle Trip" data from July 2021 [5], which is a CSV file of approximately 1.2 million rows, or 70 MiB on disk. To test our system against larger datasets, we duplicated the rows of the CSV file as many as 15 times, yielding file sizes up to 1 GiB. This is the same approach taken by Petersohn et al. in their original evaluation of Modin [14].

To test our optimizations, we constructed three different queries. Each query also has a "fast" and "slow" ordering, as described below.

(1) *Comparison of* NaN *values:* The query removes rows from the dataset where the dropoff location (column `DULocationID`) is NaN, then removes rows where the pickup location (column `POLocationID`) is NaN. There are more NaN values in the pickup location column, so performing the latter mask first is faster.
(2) *Redundant numeric comparison:* The query retains all rows where the dropoff location ID (an ordinal number) is greater than 100, then retains rows where the dropoff ID is greater than 200. Since the first operation is redundant, performing the latter comparison first is faster.
(3) *Workflow simulation:* This query simulates a more realistic sequence of operations that a data scientist might perform. The query first removes rows where the dropoff location is NaN, then retains rows where the pickup ID is greater than 100, and finally selects the pickup ID column and computes its mean. Once again, the second comparison (comparing pickup ID > 100) would yield fewer values in the final DataFrame, so performing it first is most efficient.

We examine these benchmarks given by the user in both the "fast" and "slow" query orders; our expectation is that lazy Modin with query optimizations enabled would allow both orders to run in approximately the same amount of time, in keeping with criteria (2).

In addition to testing these query orders, each benchmark was also further split into three different categories:

(1) *Full query tests:* These tests timed the entirety of a query from start to finish, including statistics collection and query optimization when enabled.
(2) *Component tests:* As mentioned earlier, since we cannot easily benchmark our system directly against think time metrics, we must measure the performance of each component of the query optimization pipeline. These components are data ingestion (`pd.read_csv`), query plan creation, statistics collection, query plan optimization, and query plan execution; each component is run independently in a separate test for each query (due to limitations in the pytest benchmarking framework, we could not modularly test every portion in sequence and had to run them as separate tests).
(3) *Double tests:* These tests run a query twice, where statistics and query plans computed after the first run can potentially be reused on the second.

6.1.1 Hardware. All benchmarks were run on an Amazon EC2 x1.16xlarge (64 vCPUs, 976 GB RAM) Spot Instance on us-east-1.

6.2 Results

6.2.1 Full Query Tests. The median runtimes of each full query tests are given in Figure 6; the bars within each category represent the number of times the input CSV has been duplicated. Lazy Modin with statistic collection disabled performs comparably to the baseline Modin in the full "workflow" benchmark, but begins to fall behind on the comparison and `NaN` microbenchmarks. Lazy Modin with optimizations manages to fare even worse, with the runtime for the 15-duplicate (1 GB) dataset consistently at least doubling that of Modin. However, we do see that the difference in runtime between the "fast" and "slow" version of each query, especially in the `NaN` ordering benchmark, is made almost nonexistent by query optimizations.

6.2.2 Double Query Tests. The median runtimes of each "double query" test, where the same query is repeated twice, is given in Figure 7. Our expectation is that the cached query plans in the second run of each query would improve the speed of computation, which appears to hold true given the evidence. Across the board, lazy Modin with optimizations sees significant improvement relative to the other configurations, though largely remains the slowest.

6.2.3 Query Component Tests. To identify the root cause for the relative slowness of our "optimized" version of Modin, we examine the results of the component tests to see where most runtime is spent. The most telling example is the runtime breakdown from the 15-duplicate comparison benchmark, shown in Figure 8. The longest operation is statistics collection by an order of magnitude, with CSV reads clocking in a respsectable second, and all other phases nowhere in the same neighborhood. Such a singly large component is likely responsible for the entirety of the speed difference between our version of Modin and the baseline.

We discuss the causes of statistic collection's high runtime in the next section. Regardless, this system shows potential for performance gains in practice because most of the difference in runtime is attributable to statistics collection, an optional component that we can defer to the background by opportunistic execution; at the very least, we demonstrate that optimizations allow a poorly ordered query submitted by a user to perform as well as a hand-optimized one.

6.2.4 Statistics Collection Runtime. Finally, we examine the relationship between input dataset size and statistics collection runtime, as given in Figure 9.

All these measurements (except for the 15-duplicate one) fall within the 23 second 75th percentile of think time identified by Xin et al. [19], and are adequate for situations where a user is pondering the outputs of their code. However, these runtimes (on the order of whole seconds) are unacceptable for interactive situations where a user runs one cell directly after the other, as this imposes a significant runtime overhead.

Analysis of flame graphs from our tests reveals that the aberrant runtime of statistics collection stems primarily from calls to the `to_pandas` method of DataFrame object. This method turns a Modin DataFrame into a Pandas one, which is an expensive operation on large DataFrames because it requires Modin to retrieve the distributed physical representation of the underlying data and send its logical representation back to Pandas. This function call is used in statistic collection code because we need to iterate over the DataFrame to compute histograms, and there is no clearly idiomatic way to do so in the underlying Modin representation.

We would have spent more effort investigating a more efficient alternative to conversion to Pandas, but our initial statistics collection benchmarking function was written incorrectly and reported deceptive runtime results. Our test cases used the pytest-benchmark framework [4], which measures the runtime of a function `f` when the programmer calls `benchmark(f)` within a test case. Pytest-benchmark will automatically call `f` multiple times within a single instance of `benchmark(f)`, so as to produce a more reliable average measurement; however, it does not rerun any setup functions. In our case, we read a new DataFrame from a CSV, then directly benchmarked the runtime of the `stats_manager.compute_next` function, which computes statistics for the oldest DataFrame not yet serviced by the statistics queue. The first call to `compute_next` behaves as expected, producing a histogram for the new DataFrame. Unfortunately, since no new DataFrames are created in between benchmarking iterations of `compute_next`, the remaining iterations are run while the statistics queue was empty, and the benchmark thus silently reported very low runtimes that actually resulted from not performing any computations at all. The results given in this report are from a fixed version of the test case, but we recognized this error too late to sufficiently debug the problem.

6.3 Challenges

In addition to the misleading statistics runtime benchmarks, our initial benchmark suite consistently ran out of memory (on a VM with 64GB RAM we used while prototyping, not the x1.16xlarge

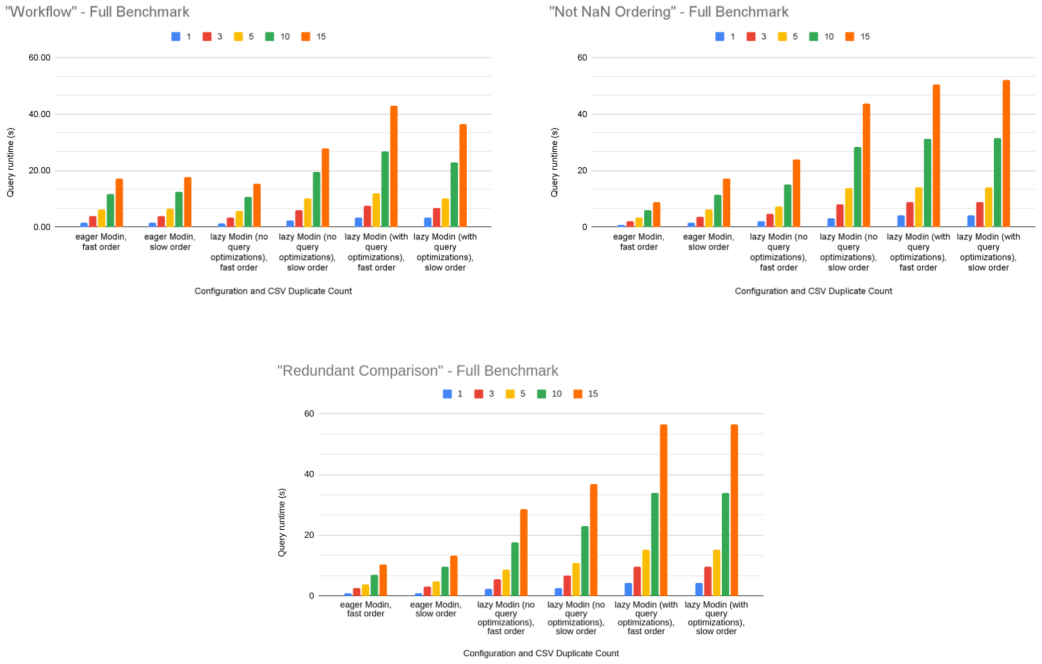

Figure 6: Full query benchmark runtimes.

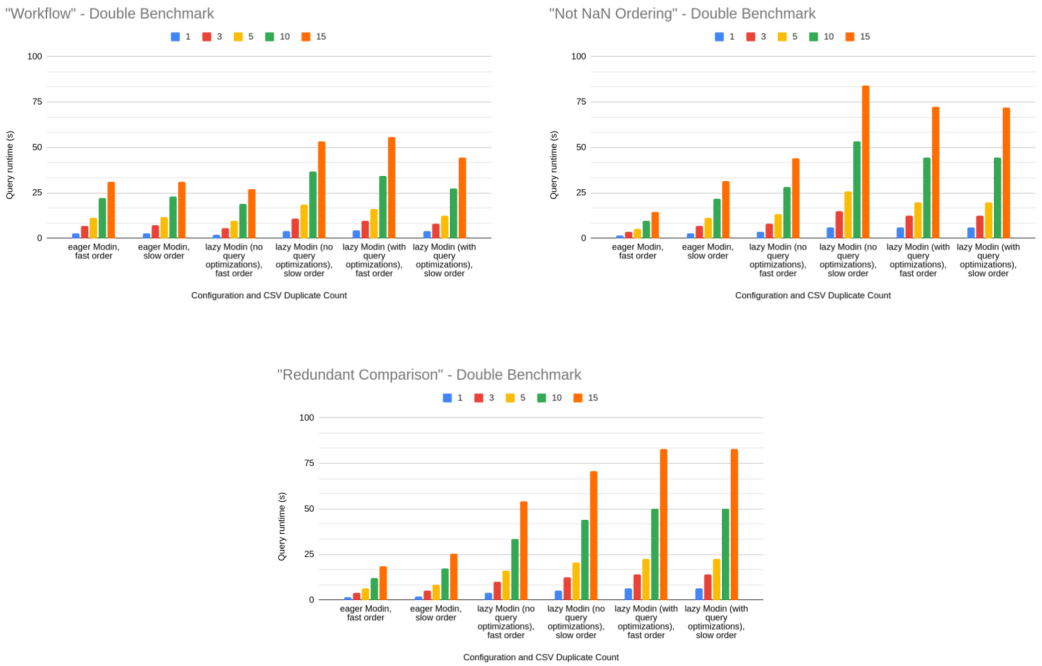

Figure 7: Double query benchmark runtimes.

	`read_csv`	Statistic Collection	Plan Construction	Query Optimization	Query Execution
Baseline	2.86+E0	N/A	N/A	N/A	7.55
No statistics	2.91+E0	N/A	3.18E-04	N/A	2.49E-05
With statistics + optimizations	2.90+E0	2.80E+01	3.24E-04	1.18E-05	1.90E-05

Figure 8: Runtimes of components of the optimization pipeline, in seconds.

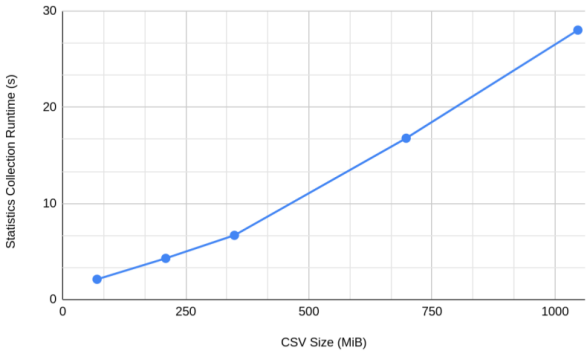

Figure 9: Statistics collection runtime as a function of input CSV size.

AWS monsters) if it ran too long, and caused Ray to begin spilling massive amounts of data to disk. This caused massive performance penalties, and unfairly slowed tests that were run later (generally those on larger duplicate counts) by forcing them to swap to disk instead of using only RAM. We later discovered that our extremely aggressive memoization of the results of previous query plans was causing a memory leak; the global dictionary acting as the cache of plan results was not reset between test cases, and as new DataFrames were being created, these old values were becoming obsolete. Our solution was to clear this cache between runs of test cases, which greatly stabilized memory usage and improved our test results compared to the benchmarks we produced for our poster presentation. However, this does leave a problem for users who might inadvertently grow the plan cache by performing operations without realizing the need to clear it; future work should either give the cache a fixed size, or dynamically drop cached query plans whose DataFrames are no longer being referenced elsewhere.

Implementing query optimizations also proved inordinately difficult for us, as neither author had taken a formal databases course prior to this class. We also realized too late that our choice of representation for `QueryPlan` structures made applying rewrite rules and transformations difficult: recall from Section 5 that we represented `QueryPlans` as lists of operators, with DAG nodes made implicit by fields of individual operators pointing to other `QueryCompiler` instances. A more sensible representation would be a recursive expression tree common in compiler literature; this tree structure is also strongly suggested by Abraham and Silberschatz [16].

7 RELATED WORK

To our knowledge, there does not exist another database system that uses think time to perform computation; Petersohn et al. refer to the notion of opportunistic execution as a novel one [14].

Besides Modin itself, other DataFrame frameworks have also presented unique ways of leveraging parallelism.

Spark [6] is one of the longest-lived pieces of software in this area, as it allows users to issue parallel queries to fully distributed datasets. Like our work, Spark executes all computations lazily, as plans are stored in a DAG representation before being dispatched. Previous work, such as Skew Join optimizations in Spark that ensure work remains evenly distributed [17], and adaptive query processing frameworks like Cuttlefish [9], focus primarily on optimizations that are either run in-line with queries, or explicitly requested by the user.

In 2019, Koalas [1] was first introduced as an open source Python package that implements the Pandas data frame API on top of Apache Spark to enable data scientists to easily scale their workloads. Since its release, Koalas has been widely adopted by data scientists and was recently merged into PySpark. Similar to Modin, Pandas on Spark enables users to easily improve the performance of their workloads without needing to understand the underlying system framework; users simply need to change one import statement. The performance of the Pandas API on Spark is capable of scaling linearly with the number of nodes in the cluster with certain data sets. Additionally, the pandas API on Spark is able to outperform native pandas even on a single machine due to Spark engine optimizations like multithreading and the Spark SQL Catalyst Optimizer [10].

Hagedorn et al. propose Grizzly, another drop-in replacement for Pandas that translates Pandas API calls to SQL expressions in order to improve the performance of queries [8]. The motivation for the creation of Grizzly stems from the fact that Pandas requires that data be stored in files that must be loaded into the program for processing; however, most companies store their data in database management systems (DBMS). Experimental analysis based on the TPC-H benchmark dataset with a scale factor of SF100 revealed that Pandas queries took significantly longer than Grizzly queries, with some queries taking up to almost 200x longer. Additionally, Pandas consumed 11 GB of RAM to perform the test while Grizzly only took up 0.33 GB of RAM. This difference stems from Pandas storing the data in memory while Grizzly interfaces directly with the database system and stores very little data in memory.

8 CONCLUSION

As the field of data science has grown in recent years, so has the scale of the data it studies. Just as the end of Moore's Law signaled a switch to massive parallelism in hardware designs, the inability of software responsiveness to keep up with the growing size and

complexity of data science workloads has forced more creative solutions into existence. The Modin framework's vision of an opportunistic DataFrame system allows users to take advantage of parallel hardware while working with large datasets, without having to spend pointless time waiting for queries to complete. Our work in implementing lazy evaluation in Modin lays the foundation for a fully opportunistic framework, with query optimizations and statistics collection run in the background as a starting point.

8.1 Future Work

To minimize the potential impact of statistics collection routines exceeding think time and causing latency for the user, the callbacks invoked by our Jupyter notebook extension must somehow be interruptable, and ideally could be resumed at the next instance of think time. The *continuation* proramming model [15] would easily allow for this, though it would require existing optimization algorithms and state representation to adapt accordingly.

Our lazy query compiler also does not yet support the entire DataFrame API surface that eager Modin does; however, the simple nature of Modin's internal DataFrame algebra (as described by Petersohn in [13]) means extending lazy evaluation to the majority of the DataFrame API would not be too difficult.

Further query optimizations can also be implemented during think time, including those based on traditional reordering rules (as established in Abraham and Silberschatz [16]), as well as those like Skew Join [17] that are specific to distributed database representation. The unique environment of the Jupyter notebook potentially allows for these operations to occur essentially for "free," without using CPU cycles that would otherwise be dedicated to other computations.

Finally, the most important (and perhaps most impactful) extension of our work would be the implementation of a fully-fledged opportunistic execution framework as laid out in [14]. Our framework only schedules optional operations (statistics collection and query optimization) during think time, and leaves users to explicitly request which query plans they want executed. This ergonomics of this system leave something to be desired, and it is difficult for users to know how to optimally schedule execution of their queries to maximize processor utilization. To this end, a truly opportunistic system would schedule query executions during think time, and potentially even learn dynamically what queries it is expected to prioritize for its users. A simple prototype where a Jupyter notebook plugin schedules queries executions is already possible with minimal modifications to our project, but as alluded to earlier, our framework does not yet allow the preemption of operations issued during think time. A system built around continuations, where operations executing during think time can be preempted for higher-priority ones, could drastically cut down on the amount of idle time users spend waiting for cells to execute.

REFERENCES

[1] 2020. Koalas: pandas API on Apache Spark. https://koalas.readthedocs.io/en/latest/.

[2] 2021. Pandas API Reference. https://pandas.pydata.org/pandas-docs/stable/reference/index.html

[3] 2021. Project Jupyter. https://jupyter.org/

[4] 2021. pytest-benchmark 3.4.1. https://pypi.org/project/pytest-benchmark/

[5] 2021. TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[6] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[7] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling.* https://dask.org

[8] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting pandas in a box. *The Conference on Innovative Data Systems Research (CIDR)* (2021).

[9] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A lightweight primitive for adaptive query processing. *arXiv preprint arXiv:1802.09180* (2018).

[10] Hyukjin Kwon and Xinrong Meng. 2021. Pandas API on Upcoming Apache Spark™ 3.2. https://databricks.com/blog/2021/10/04/pandas-api-on-upcoming-apache-spark-3-2.html.

[11] Zhicheng Liu and Jeffrey Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2122–2131. https://doi.org/10.1109/TVCG.2014.2346452

[12] Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (May 2007), 21–29. https://doi.org/10.1109/MCSE.2007.53

[13] Devin Petersohn. 2021. *Dataframe Systems: Theory, Architecture, and Implementation.* Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-193.html

[14] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2033–2046. https://doi.org/10.14778/3407790.3407807

[15] John C Reynolds. 1993. The discoveries of continuations. *Lisp and symbolic computation* 6, 3-4 (1993), 233–247.

[16] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2011. *Database System Concepts* (6 ed.). McGraw Hill.

[17] Zhuo Tang, Wei Lv, Kenli Li, and Keqin Li. 2018. An intermediate data partition algorithm for skew mitigation in spark computing environment. *IEEE Transactions on Cloud Computing* (2018).

[18] Yifan Wu. 2020. Is a Dataframe Just a Table?. In *10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019) (OpenAccess Series in Informatics (OASIcs))*, Sarah Chasins, Elena L. Glassman, and Joshua Sunshine (Eds.), Vol. 76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:10. https://doi.org/10.4230/OASIcs.PLATEAU.2019.6

[19] Doris Xin, Devin Petersohn, Dixin Tang, Yifan Wu, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Enhancing the Interactivity of Dataframe Queries by Leveraging Think Time. *CoRR* abs/2103.02145 (2021). arXiv:2103.02145 https://arxiv.org/abs/2103.02145