Enhancing the Interactivity of Dataframe Queries by Leveraging Think Time

Doris Xin, Devin Petersohn, Dixin Tang, Yifan Wu, Joseph E. Gonzalez,
Joseph M. Hellerstein, Anthony D. Joseph, Aditya G. Parameswaran
UC Berkeley

1 Introduction

During the course of Machine Learning (ML) model development, a critical first step is *data validation*, ensuring that the data meets acceptable standards necessary for input into ML training procedures. Data validation involves various sub-tasks, including *data preparation*: transforming the data into a structured form suitable for the desired end-goal, and *data cleaning:* inspecting and fixing potential sources of errors. These validation steps of data preparation and cleaning are essential even if the eventual goal is simply exploratory data analysis as opposed to ML model development—in both cases, the quality of the eventual end-result, be it models or insights, are highly tied to these steps. This data validation process is highly exploratory and iterative, as the data scientist often starts off with a limited understanding of the data content and quality. Data scientists therefore perform data validation through *incremental trial-and-error*, with the goals evolving over time: they make a change, inspect the result (often just a sample) to see if it has improved or "enriched" the dataset in some way, e.g., by removing outliers or filling in NULL values, expanding out a nested representation to a flat relational one, or pivoting to organize the dataset in a different manner more aligned with the analysis goals.

To support this iterative process of trial-and-error, data scientists often use powerful data analysis libraries such as Pandas [7] within computational notebooks, such as Jupyter or Google Colab [12, 1]. Pandas supports a rich set of incrementally specified operators atop a tolerant dataframe-based data model, drawn from relational algebra, linear algebra, and spreadsheets [14] embedded within a traditional imperative programming language, Python. While the use of dataframe libraries on computational notebooks is a powerful solution for data validation on small datasets, this approach starts to break down on larger datasets [14], with many operations requiring users to wait for unacceptably long periods, breaking flow. Currently, this challenge may be overcome by either switching to a *distributed dataframe system* (such as Dask [3] and Modin [6]), which introduces setup overhead and potential incompatibilities with the user's current workflow, or by users manually optimizing their queries, which is a daunting task as pandas has over 200 dataframe operations. We identify two key opportunities for improving the interactive user experience *without requiring changes to user behavior*:

- Users often do not want to inspect the entire results of every single step.
- Users spend time thinking about what action to perform next.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Unfortunately, at present, every cell (the unit of execution in a notebook) issued by the user is executed verbatim immediately, with the user waiting until execution is complete to begin their next step. Moreover, the system is idle during *think time*, i.e., when users are thinking about their next step or writing code. Fundamentally, *specification* (how the user writes the query) and *execution* (what the system executes) are tightly coupled.

In this paper, we outline our initial insights and results towards optimizing dataframe queries for interactive workloads by *decoupling specification and execution*. In particular, dataframe queries are not executed immediately, unless the user intends to inspect the results, but are deferred to be computed during think time. We distinguish operators that produce results that users inspect, what we call *interactions*, from those that do not. We can then use program slicing to quickly determine what code is *critical* in that it influences the interactions, i.e., what the user intends to see immediately, and what is *non-critical*, in that it can be computed in the background during think-time to speed up future interactions. For the critical portions, we further identify if it can be rewritten in ways that allows us to improve interactivity further. For example, identifying that users often only examine the first or last few rows/columns of the result allows us to compute this as part of the critical portion and defer the rest to the non-critical portion. For the non-critical portions, by deferring the execution of the non-critical portions, we can perform more holistic query planning and optimization. Moreover, we may also *speculatively* compute other results that may prove useful in subsequent processing. We call our framework *opportunistic evaluation*, preserving the benefits of eager evaluation (in that critical portions are prioritized), and lazy or deferred evaluation (in that non-critical portions are deferred for later computation). This paper builds on our prior vision [14], wherein we outline our first steps towards establishing a formal framework for reasoning about dataframe optimization systematically.

2 Background and Motivation

2.1 Key Concepts

Users author dataframe queries in Jupyter notebooks, comprising code cells and output from executing these code cells. Figure 1 shows an example notebook containing dataframe queries on the left. Each code cell contains one or more queries and sometimes ends with a query that outputs results. In this part of Figure 1, every cell ends in a query (namely, `df1.describe()`, `df1.head()`, and `df2.describe()`) that outputs results. Dataframe queries are comprised of operators such as `apply` (applying a user defined function on rows/columns), `describe` (compute and show summary statistics), and `head` (retrieve the top K rows of the dataframe). Operators such as `head` and `describe`, or simply the dataframe variable itself, are used for inspecting intermediate results. We call these operators ***interactions***. Users construct queries incrementally by introducing interactions to verify intermediate results. An interaction usually depends on only a subset of the operators specified before it. For example, `df1.describe()` in Figure 1 depends only on `df1 = pd.read_csv("small_file")` but not `df2 = pd.read_csv("LARGE_FILE")`. We call the set of dependencies of an interaction the ***interaction critical path***. To show the results of a particular interaction, the operators not on its interaction critical path do not need to be executed even if they were specified before the interaction.

After an interaction, users spend time inspecting the output and authoring new queries based on the output. We call the time between the display of the output and the submission of the next query ***think time***, during which the CPU is idle (assuming there are no other processes running on the same server) while the user inspects intermediate results and authors new queries. We propose ***opportunistic evaluation***, an optimization framework that leverages this think time to reduce interactive latency. In this framework, the execution of operators that are not on interaction critical paths, which we call *non-critical operators*, are deferred to being evaluated asynchronously during think time to speed up future interactions.

2.2 Motivating Scenarios and Example Optimizations

To better illustrate the optimization potential of opportunistic evaluation, we present two typical data analysis scenarios that could benefit from asynchronous execution of queries during think time to minimize interactive latency. While the user's program remains the same, we illustrate the modifications to the execution plan that highlights the transformations made.

2.2.1 Interaction-based Reordering

Consider a common workflow of analyzing multiple data files, shown on the left in Figure 1. The user, Sam, executes the first cell, which loads both of the files, and is forced to wait for *both* to finish loading before she can interact with *either* of the dataframes. To reduce the interactive latency (as perceived by the user), we could conceptually re-order the code to optimize for the immediate output. As shown on the right in Figure 1, the re-ordered program defers loading the large file to after the interaction, `df1.describe()`, obviating the need to wait for the large file to load into `df2` before Sam can start inspecting the content of the small file. To further reduce the interactive latency, the system could load `df2` while Sam is viewing the results of `df1.describe()`. This way, the time-consuming process of loading the large file is completed during Sam's think time, thus reducing the latency for interacting with `df2`.

```
1 df1 = pd.read_csv("small_file")
2 df2 = pd.read_csv("LARGE_FILE")
3 df1.describe()

  output

4 df1["col1"] = df1["col1"].apply(UDF1)
5 df1["col2"] = df1["col2"].apply(UDF2)
6 df1.head()

  output

7 df2.describe()

  output
```

```
  # user executes the first cell
1 df1 = pd.read_csv("small_file")
3 df1.describe()
  # output

  # execute the following in the background
  # while the user inspects the output above
2 df2 = pd.read_csv("LARGE_FILE")

  # user executes the second cell
  df1["col1"] = df1["col1"].apply(UDF1)
  df1["col2"] = df1["col2"].apply(UDF2)
  df1.head()
  # output

  # user executes the third cell
7 df2.describe()
  # output
```

Figure 1: Example program transformation involving operator reordering: (left) Original program where user has to wait for both files to load before viewing any; (right) Optimized program where the user can view the smaller file first while the other loads.

2.2.2 Prioritizing Partial Results

For any large dataframes, users can only inspect a handful of rows at a time. However the current evaluation mechanism requires *all* the rows to be evaluated. Expensive queries such as those involving user-defined functions (UDFs) could take a long time to fully compute, as shown on the left in Figure 2.

To reduce interactive latency, one can prioritize computation of only the portion of the dataframe inspected. This method is essentially an application of *predicate pushdown*, a standard technique from database query optimization. The right part of Figure 2 provides an example transformation for the particular operator, `groupby`. While the first cell prioritizes the computation of the inspected rows, the user may still need the result of the entire computation, which is scheduled to be computed later while Sam is still reading the result of the previous cell, `groupNow.head(10)`, i.e. the think time. A noteworthy attribute of dataframes is row and column equivalence [14], which means that predicate pushdown can also happen when projecting columns as well.

```
df = pd.read_csv("file")
groups = df.groupby("col").agg(expensiveUDF)
groups.head(10)
```

```
output
```

```
# user executes the code cell
df = pd.read_csv("file")
top10Groups = df["col"].unique()[:10]
groupsNow = df[df["col"].isin(top10Groups)].agg(expensiveUDF)
groupsNow.head(10)
# output

# execute the following in the background
# while the user inspects the output above
groups = df.groupby("col").agg(expensiveUDF)
```

Figure 2: Program transformation involving predicate pushdown. (left) Original program where the user has to wait for an expensive UDFs to fully compute; (right) Optimized program where the user can view a partial result sooner.

3 Assessment of Opportunities with Notebook Execution Traces

To assess the size of opportunity for our aforementioned optimizations to reduce interactive latency in computational notebooks, we evaluate two real world notebook corpora.

One corpus is collected from students in the **Data 100** class offered at UC Berkeley. Data 100 is an intermediate data science course offered at the undergraduate level, covering topics on tools and methods for data analysis and machine learning. This corpus contains 210 notebooks across four different assignments, complete with the *history* of cell execution content and completion times captured by instrumenting a custom Jupyter extension.

We also collected Jupyter notebooks from **Github** comprising a more diverse group of users than Data 100. Jupyter's IPython kernel stores the code corresponding to each individual cell executions in a local `history.sqlite` file[1]. We used 429 notebook execution histories that Macke et al. [13] scraped from Github that also contained pandas operations.

To assess optimization opportunities, we first quantify think time between cell executions, and then evaluate the prevalence of the code patterns discussed in Section 2.2.

3.1 Think-Time Opportunities

Our proposed opportunistic evaluation framework takes advantage of user think time to asynchronously process non-critical operators to reduce the latency of future interactions. To quantify think time, we measure the time lapsed between the completion of a cell execution and the start of the next cell execution using the timestamps in the cell execution and completion records, as collected by our Jupyter notebook extension. Note that the think time statistics are collected only on the Data 100 corpus, as the timestamp information is not available in the Github corpus. Figure 3 shows the distribution of think time intervals on the left, measured in seconds, between consecutive cell executions across all notebooks, while the right part of Figure 3 shows the distribution of the median think time intervals, measured in seconds, within each notebook. We removed automatic cell re-execution ("run all") from the dataset. We can see that while there are many cells that were executed quickly, there exist cells that had ample think time—the 75th percentile think time is 23 seconds.

3.2 Program Transformation Opportunities

Interaction-Based Reordering. To assess the opportunities to apply operator reordering to prioritize interactions, we evaluate the number of non-critical operators specified before each interaction. We use the operator DAG, to be described in Section 4.2, to determine the dependencies of an interaction and count the number of operators that

[1] `https://ipython.readthedocs.io/en/stable/api/generated/IPython.core.history.html`

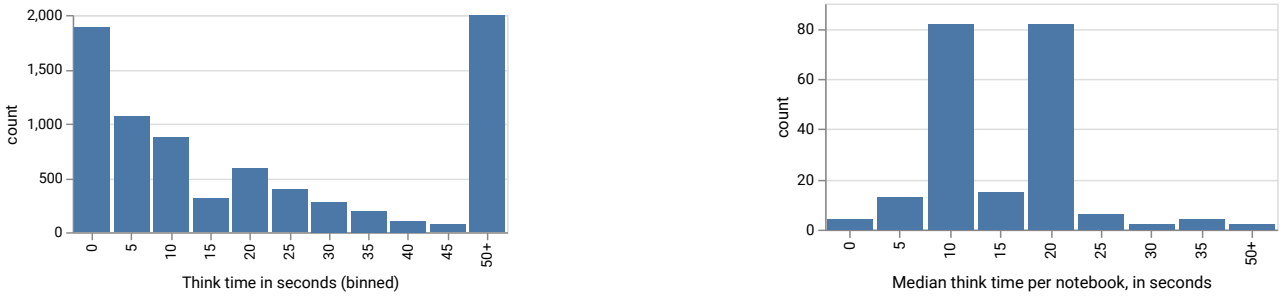

Figure 3: Think time the average "think time" between cell executions and the average think time per notebook. (left) Think time between cell executions; (right) Median think time per notebook across cells.

are not dependencies, i.e., non-critical operators, specified above the interaction. Figure 4 shows the distributions for the two datasets. In both cases, non-critical operators present a major opportunity: the Data 100 and Github corpus have, respectively, 54% and 42% interactions with non-critical operators.

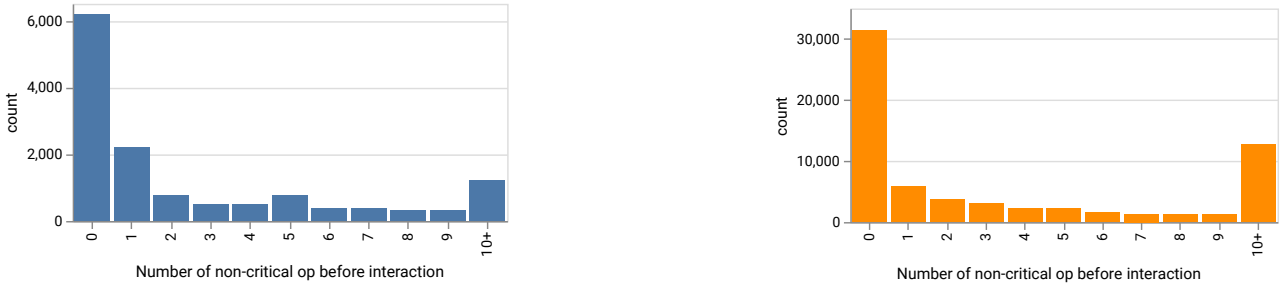

Figure 4: Number of non-critical operators before interactions. (left) Data 100: $\mu = 4$, $\sigma = 5$; (right) Github: $\mu = 7$, $\sigma = 11$

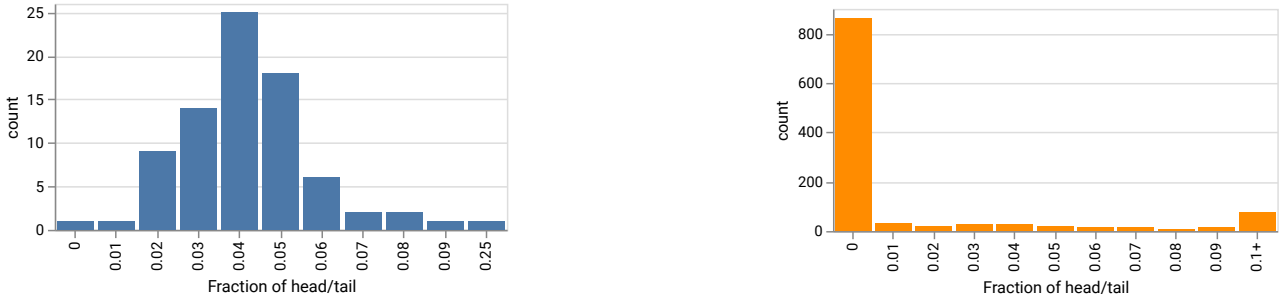

Figure 5: Stats for head/tail interactions used in each notebook. (left) Data 100: $\mu = 0.04$, $\sigma = 0.028$; (right) Github: $\mu = 0.11$, $\sigma = 0.21$

Prioritizing Partial Results. The optimization for prioritizing partial results via predicate pushdown can be applied effectively to many cases when predicates are involved in queries with multiple operators. The most common predicates in the dataframe setting are `head()` and `tail()`, which show the top and bottom K rows of the dataframe, respectively. Figure 5 show the distribution of the fraction of interactions that are either `head` or `tail` in each notebook. We see that partial results views are much more common in the GitHub dataset than Data 100. This could be due to the fact that users on GitHub tend to keep the cell output area short for better

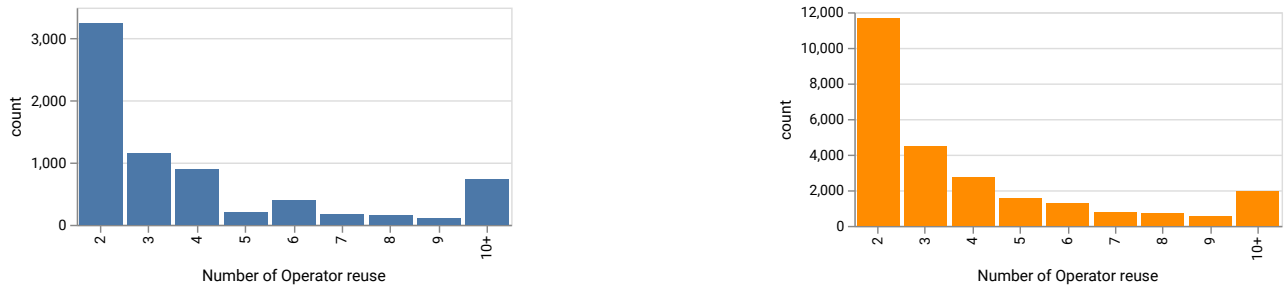

Figure 6: Distribution of number of operators that can benefit from reuse. (left) Data 100: $\mu = 5$, $\eta = 3$, $\sigma = 8$; (right) Github: $\mu = 7$, $\eta = 3$, $\sigma = 14$

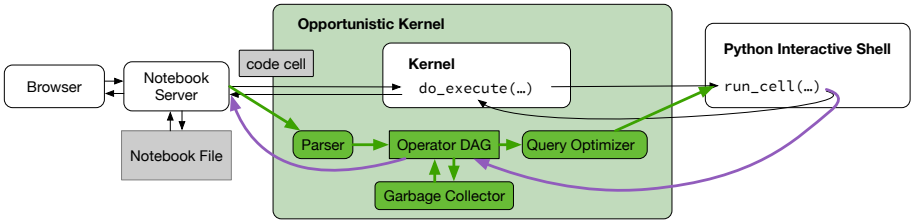

Figure 7: Opportunistic Evaluation Kernel Architecture.

rendering of the notebook by Github, but further studies are needed to corroborate this hypothesis. Lastly, partial views are not nearly as prevalent as non-critical operators before an interaction, accounting only for $< 20\%$ of the interactions.

Reuse of Intermediate Results. Since dataframe queries are incrementally constructed, with subsequent queries building on top of previous ones, another common query optimization technique that is applicable is caching these intermediate results. To assess the opportunities to speed up queries by caching, we evaluate the number of times an operator is shared by different interactions but not stored as a variable by the user. Ideally, we would also have the execution times of the individual operators, which is not possible without a full replay. We present an initial analysis that only assesses the *existence* of reuse opportunities, as shown in Figure 6. Both the Data 100 and Github datasets have a median of 3 operators that can benefit from reuse.

Of the types of optimizations explored, operator reordering appears to be the most common. Thus, we focus our initial explorations of opportunistic evaluation on operator reordering for asynchronous execution during think time, while supporting preemption to interrupt asynchronous execution and prioritize interaction.

4 System Architecture

In this section, we introduce the system architecture for implementing our opportunistic evaluation framework for dataframe query optimization within Jupyter notebooks. At a high level, we create a custom Jupyter Kernel to intercept dataframe queries in order to defer, schedule, and optimize them transparently. The query execution engine uses an operator DAG representation for scheduling and optimizing queries and caching results, and is responsible for scheduling asynchronous query executions during think time. When new interactions arrive, the execution of non-critical operators is preempted and partial results are cached to resume execution during the next think time. A garbage collector periodically uncaches results corresponding to the DAG nodes to avoid memory bloat based on the likelihood of reuse.

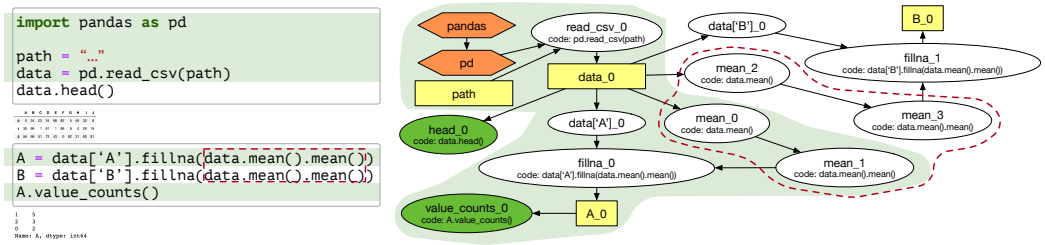

Figure 8: Example Code Snippet and Operator DAG.

4.1 Kernel Instrumentation

Figure 7 illustrates the round-trip communication between the Jupyter front-end and the Python interactive shell. The black arrows indicate how communication is routed normally in Jupyter, whereas the green and purple arrows indicate how we augment the Jupyter Kernel to enable opportunistic evaluation. First, when the code is passed from the front-end to the kernel, it is intercepted by the custom kernel that we created by wrapping the standard Jupyter kernel. As shown in the green box, the code is passed to a parser that generates a custom intermediate representation, the operator DAG. The operator DAG is then passed to the query optimizer to create a physical plan for the query to be executed. This plan in then passed to the Python interactive shell for execution. When the shell returns the result after execution, the result is intercepted by the custom kernel to augment the operator DAG with runtime statistics as well as partial results to be used by future queries, and the query results are passed back to the notebook server, as indicated by the purple arrows.

4.2 Intermediate Representation: Operator DAG

Figure 8 shows an example operator DAG constructed from the code snippet on the left. The orange hexagons are imports, yellow boxes are variables, ovals are operators, where green ovals are interactions. The operator DAG is automatically constructed by analyzing the abstract syntax tree of the code, in the parser component in Figure 7. We adopt the static single assignment form in our operator DAG node naming convention to avoid ambiguity of operator references, as the same operator can be invoked many times, either on the same or different dataframes. In the case that the operator DAG contains non-dataframe operators, we can simply project out the irrelevant operators by keeping only the nodes that are weakly connected to the `pandas` import node.

To see how the operator DAG can be used for optimization, consider two simple use cases:

Critical path identification. To identify the critical path to the interaction `A.value_counts()`, we can simply start at the corresponding node and traverse the DAG backwards to find all dependencies. Following this procedure, we would collect all nodes in the green region as the critical path to `A.value_counts()` (corresponding statements are highlighted in green on the left), slicing out the operators associated with the statement `B = data['B'].fillna(data.mean().mean())`, which does not need to be computed for the interaction.

Identifying repeated computation. Note that `data.mean().mean()` is a common subexpression in both `A` and `B`; recognizing this allows us to cache and reuse the result for `data.mean().mean()`, which is expensive since it requires visiting every element in the dataframe. We assume that operators are *idempotent*, i.e., calling the same operators on the same inputs would always produce the same results. Thus, descendants with identical code would contain the same results. Based on this assumption, we eliminate common subexpressions by starting at the root nodes and traversing the graph breadth first, merging any descendants with identical code. We then proceed to the descendants of the descendants and carry out the same procedure until the leaf nodes are reached. Following this procedure, we would merge `mean_0` with `mean_2` and `mean_1` with `mean_3` in the red dotted region in Figure 8.

We will discuss more optimizations in Section 5.

4.3 Operator Execution & Garbage Collector

When a notebook cell is executed, the opportunistic kernel first parses the code in the cell to add operators to the operator DAG described above. The DAG is then passed to the query optimizer, which will either immediately kick off the execution of interaction critical paths, if they are present in the DAG, or consider all the non-critical operators to determine what to execute next. We discuss optimizations for non-critical operators in Section 5.2.

After the last interaction is executed and the results are returned, the query optimizer will continue executing operators asynchronously until the entire DAG is executed. In the event that an interaction arrives while a non-critical operator is executing, we preempt the execution of the non-critical operator to avoid delaying the execution of the interaction critical path. We discuss optimizations for supporting effective preemption in Section 5.1.

While the kernel executes operators, a garbage collector (GC) is working in the background to uncache results in the operator DAG to control memory consumption. A GC event is triggered when memory consumption is above 80% of the maximum memory allocated to the kernel, at which point the GC inspects the operator DAG to uncache the set of operator results that are the least likely to speed up future queries. We discuss cache management in Section 5.2.

5 Optimization Framework

The opportunistic evaluation framework optimizes for interactive latency by deferring to think time the execution of operators that do not support interactions. The previous section describes how we use simple program analysis to identify the interaction critical path that must be executed to produce the results for an interaction. In this section, we discuss optimizations for minimizing the latency of a given interaction in Section 5.1 and optimizations for minimizing the latency of future interactions by leveraging think time in Section 5.2. We discuss how to model user behavior to anticipate future interactions in Section 5.3.

5.1 Optimizing Current Interactions

Given an interaction critical path, we can apply standard database optimizations for single queries to optimize interactive latency. For example, if the interaction operator is `head` (i.e., examining the first K rows), we can perform predicate pushdown to compute only part of the interaction critical path that leads to the top K rows in the final dataframe. The rest can be computed during think time in anticipation of future interactions.

The main challenge for optimizing interactive latency in opportunistic evaluation is the ability to effectively preempt the execution of non-critical operators. This preemption ensures that we avoid increasing the interactive latency due to irrelevant computation. The current implementation of various operators within pandas and other dataframe libraries often involves calling lower-level libraries that cannot be interrupted during their execution. In such cases, the only way to preempt non-critical operators is to abort their execution completely, potentially wasting a great deal of progress. We propose to overcome this challenge by partitioning the dataframe so that preemptions lead to, in the worst case scenario, only loss of the progress on the current partition.

Dataframe partitioning. Partitioning the dataframe in the opportunistic evaluation setting involves navigating the trade-off between the increase in future interactive latencies due to loss of progress during preemption and the reduction in operator latency due to missed holistic optimizations on the entire dataframe. In the setting where interactions are sparse, a single partition maximizes the benefit of holistic optimization while losing progress on the entire operator only occasionally due to preemption. On the other hand, if interactions are frequent and erratic, a large number of small partitions ensures progress checkpointing, at the expense of longer total execution time across all partitions. Thus, the optimal partitioning strategy is highly dependent on user behavior. We discuss how to model user behavior in Section 5.3.

Without a high-fidelity user interaction model, we can create unevenly sized partitions to handle the variability in the arrival rate of interactions. First, we create small partitions for the top and bottom K rows in the dataframe not only to handle the rapid succession of interactions but also to support partial-result queries involving `head` and `tail` that are prevalent in interactive dataframe workloads. Then, for the middle section of the dataframe, the partitions can reflect the distribution of think time such that the partition sizes are smaller at intervals where interactions are likely to be issued. For example, if the median think time is 20s and the operator's estimated execution time is 40s, it might be desirable to have smaller partitions after 50% of the rows have been processed.

The above strategy assumes sequential processing of every row in the dataframe. If, instead, the prevalent workload is working with a select subset of rows, then it is more effective to partition based on the value of the attributes that are commonly used for selection. Of course, partitioning is not necessary if computation started during think time does not block computation for supporting interactions.

Note that another important consideration in generating partial results is the selectivity of the underlying operators and whether they are blocking operators. For the former, we may need to employ a much larger partition simply to generate K results. For the latter, we may need to prioritize the generation of the aggregates corresponding to the groups in the top or bottom K (in the case of group-by), or to employ algorithms that prioritize the generation of the K first sorted results (in the case of sorting). In either case, the problem becomes a lot more challenging.

5.2 Optimizing Future Interactions Leveraging Think Time

Non-critical Operator scheduling. We now discuss scheduling non-critical operators. Recall that these operators are organized in a DAG built from queries. The job of our scheduler is to decide which *source* operators to execute. Source operators in the DAG are those whose precedent operators do not exist or are already executed. We assume an equal probability of users selecting any operator in the DAG to extend with an interaction.

The scheduler is optimized to reduce the interaction latency; we introduce the notion of an operator's *delivery cost* as the proxy for it. If an operator has not been executed yet, its delivery cost is the cost of executing the operator along with all of its unexecuted predecessors. Otherwise, the delivery cost is zero. Our scheduler prioritizes scheduling the source operator that can reduce the delivery cost across all operators the most. We define a utility function $U(s_i)$ to estimate the benefit of executing a source operator s_i. This function, for a node s_i is set to be the sum of the delivery cost for the source operator and all of its successors D_i:

$$U(s_i) = \sum_{j \in D_i} c_j \tag{3}$$

where c_j is the delivery cost for an operator j. Our scheduler chooses to execute the one with the highest $U(s_i)$. This metric prioritizes those operators that "influence" as many expensive downstream operators as possible.

Caching for reuse. When we are executing operators in the background, we store the result of each newly computed operator in memory. However, if the available memory (i.e., the memory budget) is not sufficient to store the new result, we need to recover enough memory by discarding materialized results of previously computed operators. If the discarded materialized results are needed by future operators, we will execute the corresponding operators to recompute them. Here, the optimization problem is to determine which materialized results should be discarded given the memory budget. Our system addresses this problem by systematically considering three aspects of a materialized result, denoted r_i: 1) the chance of r_i being reused, p_i, 2) the cost of recomputing the materialized result, k_i, and 3) the amount of memory it consumes, m_i. We estimate p_i by borrowing ideas from the LRU replacement algorithm. We maintain a counter T to indicate the last time any materialized result is reused and each materialized result is associated with a variable t_i that tracks the last time it is reused. If one materialized result r_i is reused, we increment the counter T by one and set t_i to T. We use the following formula to estimate p_i:

$$p_i = \frac{1}{T + 1 - t_i} \tag{4}$$

We see that the more recently a materialized result r_i is reused, the higher p_i is. We can use a cost model as in relational databases to estimate the recomputation cost k_i. We note that we do not always recompute an operator from scratch. Given that the other materialized results are in memory, our cost model estimates the recomputation cost by considering reusing existing materialized results. Therefore, we use the following utility function to decide which materialized result should be discarded.

$$O(r_i) = p_i \times \frac{m_i}{k_i} \tag{5}$$

Here, $\frac{m_i}{k_i}$ represents the amount of memory we can spare per unit of recomputation cost to pay. The lower $\frac{m_i}{k_i}$ is, the more likely we discard r_i. Finally, our algorithm will discard the r_i with the lowest $O(r_i)$ value.

Speculative materialization. Our system not only considers caching results generated by users' programs, but also speculatively materializes and caches results that go beyond what users specify, to be used by future operators. One scenario we observed is that users intend to explore the data by changing the value of a filter repeatedly. In this case, we can materialize the intermediate output results before we apply the filter and when users modify the filter, we can reuse the saved results without computing them from scratch. The downside of this approach is that it can increase the latency of computing an interaction when the think time is limited. Therefore, we enable this optimization only when users' predicted think time of writing a new operator is larger than the time of materializing the intermediate states.

5.3 Prediction of User Behavior

The accurate prediction of user behavior can greatly improve the efficacy of opportunistic evaluation. Specifically, we need to predict two types of user behavior: think time and future interactions. Section 3.1 described some preliminary statistics that can be used to construct a prior distribution for think time. As the system observes the user work, this distribution can be updated to better capture the behavior of the specific user, as we expect the distribution of think time to vary greatly based on the dataset, task, user expertise, and other idiosyncrasies. These workload characteristics can be factored into the think time model for more accurate prediction. This think time model can be used by the optimizer to decide the size of dataframe partitions to minimize progress loss due to preemption or to schedule non-critical operators whose expected execution times are compatible with the think time duration.

To predict future interactions, we can use the models from Yan et al. [17]. These models are trained on a large corpus of data science notebooks from Github. Since future interactions often build on existing operators, we can use the future interaction prediction model to estimate the probabilities of non-critical operators in the DAG leading to future interactions, which can be used by the scheduler to pick non-critical operators to execute next. Let p_j be the probability of the children of an operator j being an interaction. We can incorporate p_j into the utility function in Equation 3 to obtain the updated utility function:

$$U_p(s_i) = \sum_{j \in D_i} c_j \times p_j \tag{6}$$

Of course, the benefits of opportunistic evaluation can lead to modifications in user behavior. For example, without opportunistic evaluation, a conscientious user might self-optimize by avoiding specifying expensive non-critical operators before interactions, potentially at the cost of code readability. When self-optimization is no longer necessary when authoring queries, the user may choose to group similar operators for better code readability and maintenance, thus creating more opportunities for opportunistic evaluation optimizations.

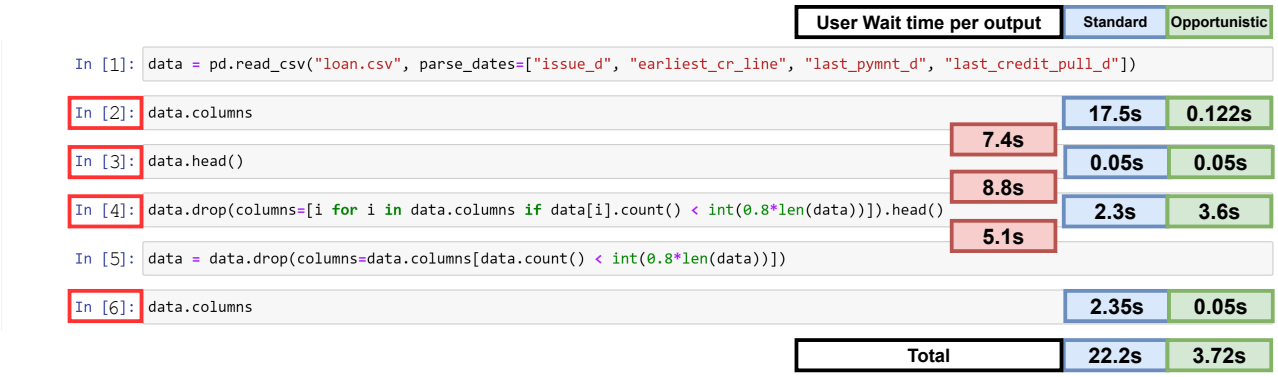

Figure 9: An example notebook. Cells that show an output are indicated with a red box.

6 Case Study

In this section, we evaluate how opportunistic evaluation will impact the end user through a case study. Figure 9 shows an excerpt from the original notebook, taken from a Kaggle competition (https://www.kaggle.com/c/home-credit-default-risk).

In this case study, the data scientist first read in the file, and was forced to immediately wait. Then, the user wanted to see the columns that exist in the dataset. This is often done when the data scientist first encounters a dataset. They therefore printed the first 5 lines with `data.head()`. This inspection is important for data validation: the data scientist wanted to ensure that the data was parsed correctly during data ingestion. After these two data validation steps, the data scientist noticed that there were a significant number of `null` values in multiple columns.

The cell labeled `In[4]` shows how the data scientist solved the `null` values problem: they decided to drop any column that does not have at least 80% of its values present. Notice that the data scientist first wanted to see what the results of the query would look like before they executed it, so they added a `.head()` to the end of the query that drops the columns. Likely this was done during debugging, where many different, but similar queries were attempted until the desired output was achieved. The query was then repeated to overwrite the `data` variable. An important note here is that the full dataset is lost at this point due to the overwriting of the `data` variable. The data scientist will need to reread the file if they want access to the full dataset again. After dropping columns with less than 80% of their values present, the data scientist double-checked their work by inspecting the `columns` of the overwritten `data` dataframe. Next, we evaluate the benefits of the opportunistic evaluation approach by determining the amount of synchronous wait time saved by leveraging think time.

To evaluate opportunistic evaluation in our case study, think time was injected into the notebook from the distribution presented in Figure 3. We found that the time that the hypothetical data scientist spent waiting on computation was almost none: the `read_csv` phase took 18.5 seconds originally, but since the output of the `columns` and `head` were prioritized, they were displayed almost immediately (122ms). The data scientist then looked at the two outputs from `columns` and `head` for a combined 16.2 seconds. This means the data scientist synchronously waited on the `read_csv` for approximately 1.3 seconds. Next, the user had to wait another 2.3 seconds for the columns with less than 80% of their values present to be dropped. Without opportunistic evaluation, the user would have to pay this time twice, once to see the first 5 lines with `head` and again to see the `data.columns` output in cell `In[6]`.

7 Related Work

Recently, researchers and developers have begun to turn their attention to the optimization, usability, and scalability of dataframes as the community begins to recognize its important role in data exploration and analysis. Some of these issues are brought on by the increasingly complex API and ever-growing data sizes. Pandas itself has best practices for performance optimization embedded within its user guide [4]. However, enabling these optimizations often requires a change to the user's behavior.

Many open source systems attempt to provide improved performance or scalability for dataframes. Often, this means only supporting dataframe functionalities that are simple to parallelize (e.g., Dask [3]), or supporting only those operations which can be represented in SQL (e.g. Koalas [5] and SparkSQL [2]). Our project, Modin [6], is the only open source system with an architecture that can support all dataframe operators.

In the research community, there are multiple notable papers that have tackled dataframe optimization through vastly different approaches. Sinthong et al. propose AFrame, a dataframe system implemented on top of AsterixDB by translating dataframe APIs into SQL++ queries that are supported by AsterixDB [15]. Another work by Yan et al. aims to accelerate EDA with dataframes by "auto-suggesting" data exploration operations [17]. Their approach has achieved considerable success in predicting the operations that were actually carried out by users given an observed sequence of operations. More recently, Hagedorn et. al. designed a system for translating pandas operations to SQL and executing on existing RDBMSs [9]. In a similar vein, Jindal et. al. built a system called Magpie for determining the optimal RDBMS to execute a given query [11]. Finally, Sioulas et. al. describe techniques for combining the techniques from recommendation systems to speculatively execute dataframe queries [16].

Our proposed approach draws upon a number of well established techniques from the systems, PL, and DB communities. Specifically, determining and manipulating the DAG of operators blends control flow and data flow analysis techniques from the PL community [8]. The optimization of dataframe operators draws inspiration from battle-tested database approaches such as predicate pushdown, operator reordering, multi-query optimization, and materialized views [10], as well as compiler optimizations such as program slicing and common subexpression elimination. Furthermore, we borrow from the systems literature on task scheduling to take enable asynchronous execution of dataframe operators during think time.

8 Conclusion & Future Work

We proposed opportunistic evaluation, a framework for accelerating interactions with dataframes. Interactive latency is critical for iterative, human-in-the-loop dataframe workloads for supporting data validation, both for ML and for EDA. Opportunistic evaluation significantly reduces interactive latency by 1) prioritizing computation directly relevant to the interactions and 2) leveraging think time for asynchronous background computation for non-critical operators that might be relevant to future interactions. We have shown, through empirical analysis, that current user behavior presents ample opportunities for optimization, and the solutions we propose effectively harness such opportunities.

While opportunistic evaluation addresses data validation prior to model training, data validation challenges are present in other parts of the end-to-end ML workflow. For example, after a trained model has been deployed, it is crucial to monitor and validate online data against the training data in order to detect data drift, both in terms of distribution shift and schema changes. A common practice to address data drift is to retrain the model on newly observed data, thus introducing data drift into the data pre-processing stage of the end-to-end ML workflow. Being able to adapt the data validation steps in a continuous deployment setting to unexpected data changes is an open challenge.

References

[1] Google Colab. colab.research.google.com.

[2] PySpark 2.4.4 Documentation: pyspark.sql module. http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions.

[3] Dask Documentation. https://docs.dask.org/en/latest/, 2020.

[4] Enhancing performance, Pandas Documentation. https://pandas.pydata.org/pandas-docs/stable/user_guide/enhancingperf.html, 2020.

[5] Koalas: pandas api on apache spark. https://koalas.readthedocs.io/en/latest/, 2020.

[6] Modin Documentation. https://modin.readthedocs.io/en/latest/, 2020.

[7] Pandas API reference. https://pandas.pydata.org/pandas-docs/stable/reference/index.html, 2020.

[8] K. Cooper and L. Torczon. *Engineering a compiler*. Elsevier, 2011.

[9] S. Hagedorn, S. Kläbe, and K.-U. Sattler. Putting pandas in a box. *The Conference on Innovative Data Systems Research (CIDR)*.

[10] J. M. Hellerstein, M. Stonebraker, J. Hamilton, et al. Architecture of a database system. *Foundations and Trends® in Databases*, 1(2):141–259, 2007.

[11] A. Jindal, K. V. Emani, M. Daum, O. Poppe, B. Haynes, A. Pavlenko, A. Gupta, K. Ramachandra, C. Curino, A. Mueller, et al. Magpie: Python at speed and scale using cloud backends. *The Conference on Innovative Data Systems Research (CIDR)*.

[12] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[13] S. Macke, H. Gong, D. J.-L. Lee, A. Head, D. Xin, and A. Parameswaran. Fine-grained lineage for safer notebook interactions. *Proceedings of the VLDB Endowment*, 2021.

[14] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran. Towards scalable dataframe systems. *Proceedings of the VLDB Endowment*, 13(11).

[15] P. Sinthong and M. J. Carey. Aframe: Extending dataframes for large-scale modern data analysis. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 359–371. IEEE, 2019.

[16] P. Sioulas, V. Sanca, I. Mytilinis, and A. Ailamaki. Accelerating complex analytics using speculation. *The Conference on Innovative Data Systems Research (CIDR)*.

[17] C. Yan and Y. He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1539–1554, 2020.