# Transparent Dataframe Partitioning

William Ma
University of California, Berkeley
Berkeley, California
williamma@berkeley.edu

Vincent Truong
University of California, Berkeley
Berkeley, California
vincentt@berkeley.edu

## ABSTRACT

With the rise of big data and the death of Moore's law, the tools of the data analyst have been forced to innovate from single-threaded to multi-threaded, multi-core, and, even, multi-node. Although these tools can help the data analyst analyze an ever-increasing amount of data, the data analyst is forced to learn how the data interacts with the system and hardware to adjust the, similarly, ever-increasing number of parameters to improve performance. Thus, the data analysts are forced to spend time learning the latest and "greatest" systems rather than exploring the intricacies of their data sets. To remedy this problem, we introduce a system that helps the data analyst by removing one of the numerous parameters from the system by automatically choosing the ideal partitioning size for the data analyst. By only adjusting this one parameter, we can easily get an improvement of 50%.

**PVLDB Reference Format:**
William Ma and Vincent Truong. Transparent Dataframe Partitioning. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

## 1 INTRODUCTION

Although databases are the de jure standard when it comes to all things data, dataframes have recently become the de facto standard when it comes to exploratory data analysis. This change is due to the fact that dataframes are, simply, the more adequate tool for exploratory data analysis. Dataframes do not require neither a deep understanding of the underlying data or the execution systems interaction with the hardware, which often requires a database administrator to tune for optimal performance.

Dataframes were born out of a need by statisticians to have all the variables in one place back in the 70s [4]. Their popularity grew as more people recognized the convenience and contributed to the two major open source versions of dataframes, R dataframes [13] and Pandas [8], written for R and python, respectively. Since these two versions of the dataframe were developed in the late 90s and early 2000s, they were single-threaded and all contained in main memory. Instead of performance, the focus of the development of these systems were in the functionality, as users would become contributors by adding a feature that they wanted. This design made it such that dataframe users did not have to understand either the data or the system.

By the turn of the decade, however, the zeitgeist changed and an emphasis on big data began. The introduction of big data meant that exploratory data analysis would no longer fit on user's laptops memory and, if they did, it was no longer reasonable to have the dataframe systems be singled threaded. Solving this need, Spark [17], with Spark SQL [3], and Dask [5], with Dask dataframes, were introduced to parallelize dataframe workflows. Since they built on a data-parallel model, they had to partition the data. Turning to traditional relational database systems, they partitioned the data using a row-wise approach. The partitions were either grouped by the values of a column or arbitrarily divided amongst some predefined number of partitions. Although these systems still made it possible for users to not understand the data or the system, these design decisions simultaneously limited the functionality of dataframes and left a lot of performance on the table.

Specifically, the current state-of-the-art does not provide a satisfying solution to parallelizing dataframe operations. They require the user to have a deep understanding of the underlying data, in order to choose the correct column to partition by, and a deep understanding of how the system interacts with the hardware, in order to choose the correct number of partitions to have. Furthermore, their partitioning scheme inherently limits the functionality of dataframes (e.g., transpose is expensive to the point of being impossible under row- or column-wise partitioning).

Rather than building on top of a system that has limited functionality, we work within Modin [12]. Due to the underlying design decisions that are baked into Modin, we are able to have the full functionality of traditional dataframe systems, while simultaneously enjoying the benefits of the parallelism that was introduced by the current state-of-the-art dataframe systems.


Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

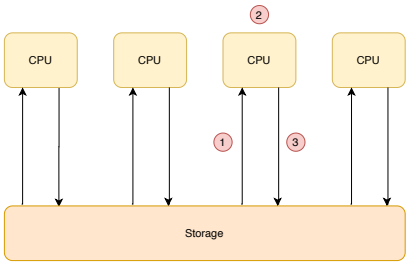


**Figure 1: Modin execution model**

However, our paper shows that it is not all sunshine and rainbows when using Modin. We show our performance analysis of various functions within Modin to reveal the bottlenecks when using Modin. Working with these bottlenecks, we demonstrate that there is still performance left on the table with regards to partitioning. Thus, we built a system that transparently determines the ideal partitioning scheme for each data set and operations to be run. Due to the preliminary nature of our work, we are able to achieve the ideal partitioning scheme over 85% of the time and achieve a 50% improvement over regular Modin. However, we also lay plans for improving our system to easily cover the entire dataframe functionality and work on any hardware configuration.

The rest of the paper is organized by first describing the performance analysis in Section 2. We describe the automatic partitioning system in section 3 and discuss the results of the analysis in section 4. We wrap up the paper with a discussion of the future work that we plan to do in section 6 and related work in section 5.

## 2 MODIN PERFORMANCE ANALYSIS

In this section, we give a brief introduction to the Modin architecture, with an emphasis on the aspects that affects the user experience and then introduce the performance analysis of each of these facets of Modin. We will show that a majority of the latency that the user experiences is not Modin itself but the execution and storage engines that Modin depends on.

### 2.1 Modin Overivew

At a high level, Modin is essentially a dataframe specific task manager with interchangeable storage and execution engines. The execution engine runs the functions and the storage engine ensures the data is distributed amongst the different cores/nodes. In order to parallelize execution, Modin uses a block partitioning scheme, which allows Modin to maintain the full functionality of dataframes. To demonstrate how Modin returns the result of a dataframe API call, we walk through the execution path of a typical dataframe function.

When functions called on a dataframe come into Modin, they are first rewritten using a set of operators, which is described in detail in the documentation [1]. In this paper, we focus only on `map`, or applying a function to each block partition; `map_axis`, or applying a function to the block partitions of a column merged together; `reduce`, or applying a function to each block partition that reduces the dimension of the block; or `map_reduce`, or applying a map to each operation and reducing the result [1].

These operators can be broken up into two different groups of operators, the block-oriented operators and the axis-oriented operators. The block-oriented operators, `map` and the `map` portion of `map_reduce`, have a degree of parallelism that is limited by the number of blocks there are because these operators only work with functions that can run on each block independently. On the other hand, the axis-oriented operators, `map_axis`, `reduce`, and the `reduce` portion of `map_reduce`, have a degree of parallelism that is limited by the number of columns or rows of blocks because these operators work with functions that require knowledge of the full column or row.

By rewriting into this limited set of operators, it reduces the amount of code needed to add in new execution and storage engines. Currently, the only execution engine are Ray [10] and Dask. At the time of the performance analysis, it was not possible to separate the storage and execution engines, but it has since been made possible due to the hard work of Devin Petersohn. In this paper, we use the Ray execution engine and Plasma [11], Ray's storage engine.

Each of these operators makes the necessary calls to the execution and storage engines to ensure that all the data is set up correctly. Although this is trivial for operators that can be distributed to each block, operators that can only be distributed to each column or row, such as `map_axis`, this requires all the blocks of a particular column or row to be co-located. For all operators, the data is then deserialized from the storage engine and made ready to run on the execution engine.

The execution engine then runs the function that the operator was given on the data.

After the operator has finished executing, the data is then serialized and placed back into the storage engine. Thus, there are three phases, as shown in figure 1, of operator within Modin: (1) read, (2) execute, and (3) write.

Since chaining multiple operators, as we will see, is simply repeating these three steps, we limit our performance analysis, and subsequent automatic partitioning, to only one function at a time.

[1] `map_reduce` is implemented as a `map` and then a `reduce` operation chained together

With this understanding of Modin's underlying architecture, we are able to do our performance analysis of Modin to find the bottlenecks and model these three steps to determine the ideal partitioning.

## 2.2 Modin Performance

To obtain data for our performance analysis of Modin, we ran functions that did nothing except for sleep with a variety of dataframe sizes from 256 MB to 4 GB at a variety of partition sizes that would use the operators `map`, `map_axis`, `reduce`, and `map_reduce`. Since Modin uses block partitioning, we ensured that all the block partitions were the same size for simplicity. To get a sense of the execution times, we ran the functions in table 1 with the same dataframe and partition sizes from the storage engine profiling full of randomly generated floating point values ranging from −100 to 100.

We did all of this profiling on AWS EC2 m5.xlarge instances (4 cores and 16 GB RAM), to simulate a laptop. Due to the preliminary nature of our work, we were unable to profile a larger variety of instances types, operators, functions, or execution engines. However, we believe that these results can easily be generalized.

From figure 2 [2], we can see that for both the storage and execution engine, the times appear to follow a linear "upper-bound estimate" and linear "lower-bound estimate" very closely with very few values falling far from these "estimates." We use this observation to build our partition models.

It is also noteworthy that the difference between falling on the lower-bound estimate and upper bound estimate can lead to a difference of nearly 100× for read times of only 200 MB of data. This will require a deeper understanding of the underlying Plasma storage engine used in Ray.

Additionally, since the storage and execution engines are chosen by the user, we are unable to simply change the execution or storage engine to improve performance. Thus, we have to work with the choices of the user. Thus, the main way to both reduce the variability and improve the performance in Modin, regardless of the storage or execution engine chosen by the user, is to control the partitioning.

To end the analysis on a more unsurprising note, we do notice, as expected, that the bottleneck of either the storage engine or the execution engine is dependent on the function being executed. For instance, we can see that there is a 25× difference between the runtime for `mean` and `nunique`, even though they both only use one `reduce` operator. This significant difference causes the bottleneck to be, on average, in the execution engine for `nunqiue`, but the bottleneck for `mean` is always in the storage engine.

[2]The data shown in these plots have already accounted for the overhead caused by combining all the blocks of a column and any difference in output sizes.

Although not shown, the rewrite from the dataframe API to the dataframe operators is trivial at these data sizes. Similarly, the execution data for other functions are not shown for the sake of space, but they follow similar patterns as `mean` and `nunique`.

Thus, this performance analysis into Modin reveals two of significant insights. Firstly, both the storage and execution engines that we tested had a large gap between the upper-bound and lower-bound times with very little falling in between the two estimates. Additionally, the difference between the upper and lower bound would grow linearly as the partition sizes increased. Using these two insights, we describe how we built our model to estimate the storage and execution engine times for each function given a partition size next.

# 3 TRANSPARENT DATAFRAME PARTITIONING

Using the insights from the performance analysis, we develop a model that will be able to account for the relative performance between different partitioning sizes. To develop this model, we first cleaned the data that we collected for the performance analysis, which we describe in section 3.1. Then, we choose and train models to represent the data in section 3.2 and show how we are able to deploy our models efficiently within Modin in section 3.3.

## 3.1 Clean Data

From section 2, we observed that the performance of both the storage and execution engines tended to be split between some upper- or lower-bound estimate. Although there was some data that was between these bounds, the majority of the data followed these upper- and lower-bound estimates relatively closely. Thus, we focused our efforts on using clustering algorithms to separate these two clusters of data.

To our knowledge, all of the clustering algorithms available would only group points together based on some relative distances between the points. In our data, however, the distance between the two groups of data was not fixed, but grew with respect to the partition size. To overcome this challenge, instead grouped the points based on the time to size ratio of these operations. In other words, we calculated the ratio between the execution or I/O time and the size of the data being executed on, read in, or written out. In essence, we were normalizing our data so that we would be able to have a fixed distance between the two clusters of data.

Having done this, we use DBSCAN [6], which is a density-based data clustering algorithm. DBSCAN works by having the user provide the hyperparameters that determine the maximum distance between two points to be considered within the same group and the minimum number of points

| Function | Operator(s) | Description |
|---|---|---|
| `round` | `map` | Rounds each value to the nearest integer. |
| `isna` | `map` | Replace each value with `True` if the value is `None`, else `False`. |
| `transpose` | `map` | Take the transpose of the dataframe. |
| `sort_values` | `map_axis` | Sort dataframe based on a given column. |
| `cumsum` | `map_axis` | Return the cumulative sum of all the values of the dataframe along either the rows or columns. |
| `nunique` | `reduce` | Returns the number of unique values in the rows or columns of a dataframe. |
| `mean` | `reduce` | Returns the mean of the values in the rows or columns of a dataframe. |
| `count` | `map_reduce` | Returns the number of non-none values of all the columns or rows of a dataframe. |
| `sum` | `map_reduce` | Returns the sum of the values along the columns or rows fo a dattaframe. |

**Table 1: Set of functions that we focused on for this work.**

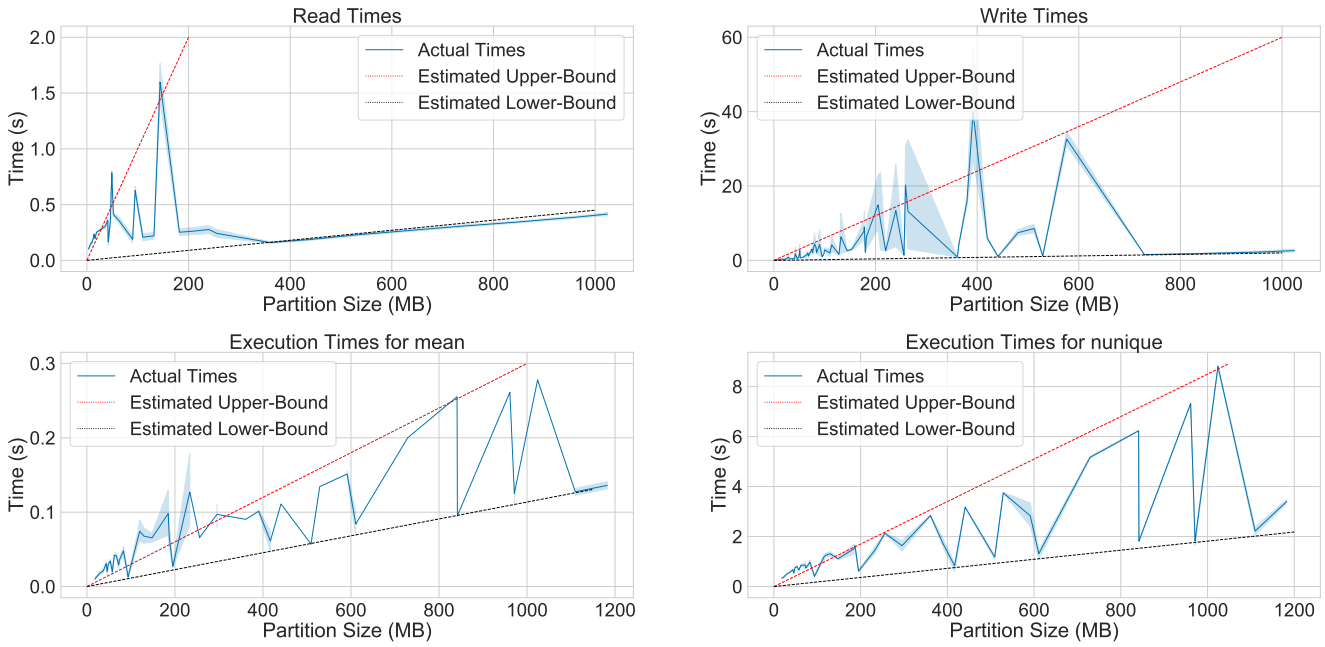


**Figure 2: Modin I/O times from the storage engine and execution times for `mean` and `nunique`**

within a group to be considered a real group, rather than an outlier. We show an example of this in figure 3. Although for this work we manually tuned the hyperparameters for each dataset, this process of hyperparameter tuning can be easily automated. Furthermore, though there is a lot of different models that need to be tuned, e.g., one for every function in every execution engine and the reads and writes for each storage engine, these models are fast, so we only need to store the hyperparameters, and the models only need to be tuned once during development. Most importantly, DBSCAN was chosen out of other data clustering algorithms, such as K-means clustering and Gaussian Mixture EM clustering, DBSCAN is robust to outliers. Given the nature of our profiling on AWS EC2 with its multitenancy and the eccentric behavior of the execution and storage engines that can occur, we are able to separate the outliers from the actual data for improved modeling.

With DBSCAN, however, we are unable to control the number of groups that get formed when clustering. Thus, we are not guaranteed to get two clear cut groups with some outliers. To remedy this, we always ensure that our hyperparameters return at least two groups of data and draw the line between the data that follows the lower-bound estimate and that which follows the upper-bound estimate by using the midpoint between the two adjacent groups that are the furthest apart. Using this midpoint, we are able to split the data into the ones that follow the lower-bound estimate and that which follows the upper-bound estimate.

Rather than directly using DBSCAN to detect outliers, we currently only separate outliers that are too big by only removing the datapoints that are 10 times larger than the

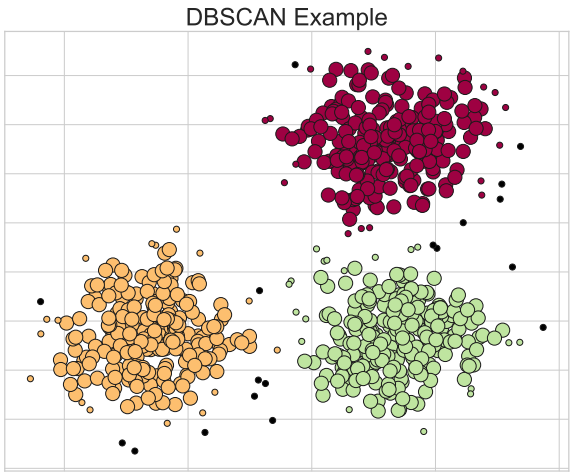


**Figure 3: Demo of how DBSCAN works**

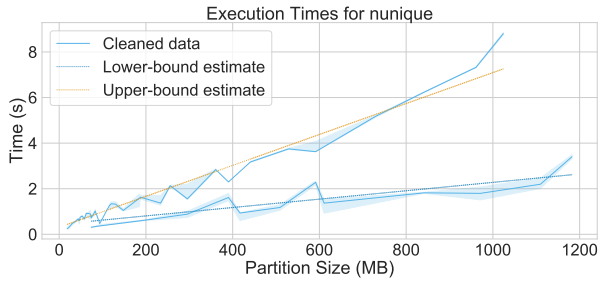


**Figure 4: Sample of our model estimating `nunique` run time given partition size**

value with which we use to separate the data into two groups. We do not consider outliers that are significantly smaller than the other estimates because there is a physical limit on how fast anything can run. However, there is no physical limit on how slow anything can execute.

Having separated the data into two groups and filtered out the outliers, we un-normalize the data back to time and partition size. It is with this process of cleaning the data that we can use simple and understandable models to estimate the run time of each phase given the partition size for every function, which we discuss next.

## 3.2 Modeling

Since the resulting separated data is roughly linear, as shown in figure 4, we use a linear regression with l2 regularization in order to estimate the run times of the execution engine for every function and for the I/O time for the storage engine.

To combine our models, we assume that the performance of the three phases of execution of each operator is independent. Although it could be argued that if the read from storage was slow, then either or both the write and execution should also be slow. However, we do not encounter this as there is a weak correlation ($< 0.4$) between the three phases of execution. Additionally, we assume that the execution and I/O times are independent across the multiple cores and repeat executions over the course of executing the operator. Similarly we find that there is a weak correlation ($< 0.4$) of the times even within the same execution of an operator.

Given the validity of these assumptions, we can simply add together these models to get an estimate on the overall runtime of an operator given a single model, using the following equation,

$$\left\lceil \frac{n_p}{n_c} \right\rceil \left( m_{read}(s_p) + m_{exe}(s_p) + m_{write}(s_p) \right) \tag{1}$$

where $n_p$ is the number of partitions, $n_c$ is the number of cpus, $s_p$ is the size of the partition, $m_{read}$ is the read model, $m_{exe}$ is execution model, and $m_{write}$ is the write model. Additionally, since these models are all linear models with respect to the partition size, the above equation easily simplifies to only two terms with simple algebra.

We combine the upper- and lower-bound estimates by calculating the expected value of the two estimates based on the number of data points that were in each group. In doing so, we are calculating the expected run time, which would lead to poor estimates of the actual run time of the entire function. However, in doing so, we believe that we are able to get more accurate relative run times. This line of work of how to best combine the lower- and upper-bound estimates will require more work to discover the best way to combine them. We discuss more about this in section 6.

Thus, the partitioning model that we have developed is both simple and quick to train because it builds on traditional and simple machine learning models. In doing so, we made it efficient to, if necessary, choose the partitioning on-the-fly. We discuss more about how we decide the partitioning using these models next.

## 3.3 Choosing a Partitioning

Currently, we use the naive approach of trying all the possible block partition sizes independently. This is done by putting every possible partitioning size into equation 1.

However, it should be trivial to frame this as an optimization problem over equation 1 rather than a black-box dynamic programming problem. With this, we do not see any major problems with extending this model to include multiple functions and more complex operators, as we will discuss in section 6.

# 4 PARTITIONING PERFORMANCE

Due to the preliminary nature of our work, we tested dataframe sizes of 256 MB, 1 GB, and 4 GB with 3, 4, and 5 different partition sizes, respectively, with the same setup as in the

performance analysis from section 2. For each of the partitioning sizes, we profiled the total run time for each function independently. With this data, calculating the ideal partitioning was simply calculating the minimum run time of the various partitioning sizes. The Modin baseline, which was to split the dataframe into the square of the number of cpus, was always included as one of the partitioning sizes.

To calculate the performance of our model, we put each of the partitioning sizes, dataframe sizes, and functions into our model and estimated the run time. Then, we chose the minimum of the estimate as the partitioning that our model chose. We look at the relative estimated times rather than the absolute times because our model estimates the expected run time, which will lay between the upper- and lower-bound estimates. For our current use-case, we also do not need the actual estimated time since we only need to find the most optimal partitioning.

For the rest of this section, we discuss how our model performs against the ideal case in section 4.1 and against the default Modin case in section 4.2.

## 4.1 Performance Against Ideal

Figure 5 shows that, for all functions except for `transpose`, the model picks a partitioning that is within 5 seconds of the ideal partitioning. However, once we take the 5 seconds into perspective for each function, by normalizing it against the ideal partition run time, we see that performance can vary anywhere from ideal all the way to 250% worse than ideal. Although this performance with respect to ideal is very poor, we can take solace in the fact that most of the functions that operate on an entire column or row, such as `nunique`, `mean`, and `cumsum`, perform within nearly all within 50% of the ideal run time. This dichotomy of performance between functions can be attributed to the large difference between the lower- and upper-bound estimates in the read times of the storage engine.

However, from figure 2, we can see that the read times [3] have a very large variability when <200MB. We believe that, given the time, increased profiling of the underlying storage engine can help us better determine the actual variability of the read times for different blocks. Additionally, with further profiling of the estimates and ideal partitioning, we can determine better partitioning sizes to getting closer to the ideal partitioning.

## 4.2 Performance Against Modin

From figure 6, we continue to see the dichotomy between the block-oriented and axis-oriented operators. For the axis-oriented operators, we see improvements of over 25 seconds. Although this improvement is good, when we normalize this by the original baseline time, we see that this is an improvement of 50%. Due to the preliminary nature of this work, we believe that this is a good starting off point for further work, which we will discuss in section 6.

Since the block-oriented operators do not see much improvement from using this model, we can simply have those functions continue to use the baseline partitioning. For those functions, the baseline partitioning already is nearly at the ideal partitioning, from figure 7. These functions, rather than those that require the full column or row, are more likely to be at the ideal partitioning because these functions perform at their optimal at larger partitions. Larger partitions are preferred for block-oriented operators due to the fact that there is less data to be read in and written out relative to the axis-oriented operators.

Although the model attempts to model this by simulating faster run times for larger partition sizes, the model in this paper does not give large enough partitions. We attribute this to, again, the high variability in the smaller partitions, which lead to a larger slope in the linear regression for the upper-bound read times in the read times, shown in figure 2. However, since the profiling was done on an instances with 4 cpus, there was only 16 partitions for every dataframe. Thus, this would lead to larger partition sizes. While the model would choose models that would have 64 or more partitions, which correlated to 4 or 16 MB partitions.

[3] The read times shown are only for block-oriented operators. We do not show the axis-oriented operators for space.

# 5 RELATED WORK

**Intra-operator planning** Other work [9] has looked at the problem of intra-operator planning. They approached the problem as a partitioned parallelism problem, similar to our work, and took the approach of determining the optimal amount of work for each processor. However, their worked looked at the problem from a more theoretical point of view and framed the problem as a flow of data from different operators and calculated the optimal partitioning to make the most of the flow.

**Inter-operator planning** There has also been work [7, 14, 15] into inter-operator query processing. [14] analyzed the performance of parallel scans and joins and determined, similar to us, that the degrees of parallelism and processor allocation, is heavily linked to the number of cpus and the amount of memory. Since we focused on dataframes, we did not have any issue with memory, as the entire dataset is always in memory. [7] approached the problem by separating the independent portions of the entire query and framed the scheduling problem as a multi-dimensional binning problem. Finally, [15] looks at the problem of inter-operator planning, through the lens of view maintenance, in a streaming setting.

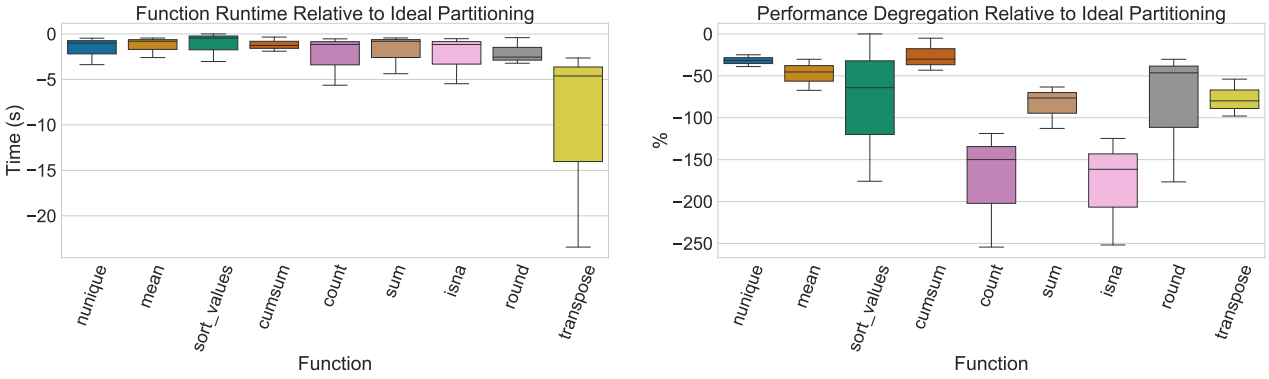


**Figure 5: Performance of our model against ideal partitioning**

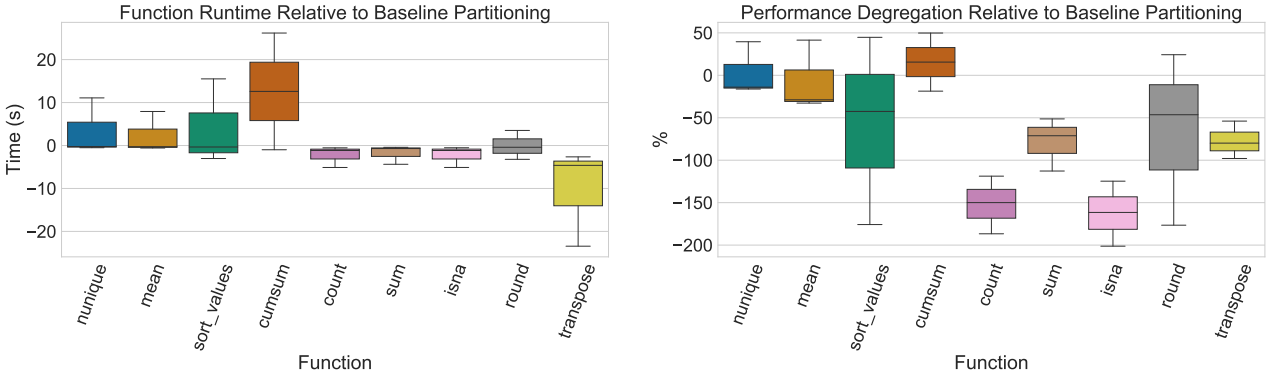


**Figure 6: Performance of our model against default Modin partitioning**

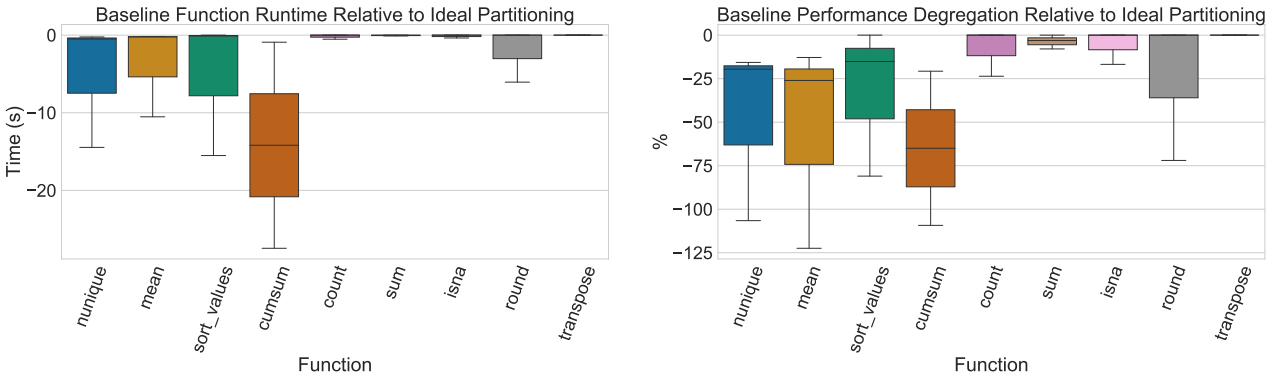


**Figure 7: Performance of our model and default Modin partitioning against ideal partitioning**

Their work differs from ours due to them trying to optimize the latency in a streaming setting.

**Query planning for the cloud** There is also a lot of work [2, 16] that looks at query planning for the cloud. These two works, where [2] is an iteration on [16], focused instead on trying to find the ideal number of instances and types of instances to launch for each batch job. Thus, rather than coming at the problem with a fixed number of nodes and a dynamic partitioning, they approached the problem with a fixed partitioning and a dynamic number of nodes.

## 6 FUTURE WORK

**More complex operations** Due to the preliminary nature of this work, we only look at simple operators. However, it will be interesting to explore how to expand this work to include operators such as `group-by` and `filters`. We do not expect much difficulty in expanding this to operators that have predictable sizes since these models should extend without issue.

On the other hand, we expect there to be some work in including operators that do not have have predictable sizes, such as `join`. One approach for this type of issue would be to include a selectivity estimator. Since this selectivity estimator would be run on-the-fly, we would need this estimator to be lightweight to be able to be shipped with the library and fast enough to justify running the estimator for operations that may only take seconds to run.

With these additional operators, it helps us expand this model to be able to capture the full dataframe functionality, which includes non-relational operations such as `pivot`.

**Multiple operations** Though it should be trivial to extend this model to multiple operations, since multiple operations essentially chain more operators together. We have shown that this model can work in settings with multiple operators with the inclusion of `map_reduce`. However, the accuracy of the model will inevitably deteriorate as the number of operators increase, we plan to explore how to capture the variability of the storage and execution engines more accurately.

For this current work, we did attempt to capture the variability as a shifted exponential distribution, Gamma distribution, and log Gamma distribution. However, these distributions, which are known for their long- and heavy-tails, fail to capture the variability introduced by the storage and execution engine that we used.

**Automation** We do plan to include this work to ship with Modin. However, due to the open-source nature of Modin, we will have to do a non-trivial amount of engineering work and improve the documentation to ensure that we do not require an undue burden of requiring contributors to understand the inner workings of our model, or even understand systems or machine learning in order to help improve the coverage of dataframe functionality or adding new storage or execution engines to Modin.

However, we do not believe that this will be too difficult as the actual model is relatively simple to automate as there are open-source implementations of hyperparameter search algorithms.

**Hardware independence** Although it is obvious that the performance of different partition sizes is dependent on the underlying hardware, it is not clear if the relative performance of partition sizes will be significantly impacted by different hardware configurations. This will require profiling on different hardware configurations to determine if the relative performance of different partition sizes will be significantly impacted to justify introducing additional complexity to capture the difference.

If we did have to capture this difference, we could ship Modin with a model that can warm-start the model. Thus, as the user uses Modin, the accuracy of the model will increase. However, this introduces issues with the trade-off between how much data to store with each operation execution and the accuracy of the model.

## 7 CONCLUSION

In this paper, we have shown an in-depth performance analysis of Modin. This performance analysis has revealed that the underlying storage and execution engines have a large variability in their performance. Additionally, the performance tends to follow either an upper- or lower-bound estimate. Using this, we are able to build a model that can predict the relative performance between different partition sizes. However, the performance of this model is limited by a lack of profiling data to help make sense of the large perceived variability in the storage and execution engines. Even with this limitation, we are able to achieve a 50% improvement over baseline Modin.

## REFERENCES


[1] 2020. Modin. https://modin.readthedocs.io/en/latest/

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[3] Michael Armbrust, Ali Ghodsi, Matei Zaharia, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, and Michael J. Franklin. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*. ACM Press, Melbourne, Victoria, Australia, 1383–1394. https://doi.org/10.1145/2723372.2742797

[4] J. Chambers, T. Hastie, and D. Pregibon. 1990. Statistical Models in S. In *Compstat*, Konstantin Momirović and Vesna Mildner (Eds.). Physica-Verlag HD, Heidelberg, 317–321.

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

[6] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. AAAI Press, 226–231.

[7] Minos N. Garofalakis and Yannis E. Ioannidis. 1996. Multi-dimensional resource scheduling for parallel queries. *ACM SIGMOD Record* 25, 2 (June 1996), 365–376. https://doi.org/10.1145/235968.233352

[8] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56 – 61. https://doi.org/10.25080/Majora-92bf1922-00a

[9] Manish Mehta and David J DeWitt. 1995. Managing intra-operator parallelism in parallel database systems. In *VLDB*, Vol. 95. 382–394.

[10] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *CoRR* abs/1712.05889 (2017). http://arxiv.org/abs/1712.05889 _eprint: 1712.05889.

[11] Philipp Mortiz and Robert Nishihara. [n.d.]. Plasma In-Memory Object Store. https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store/

[12] Devin Petersohn, William Ma, Doris Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. *Towards Scalable Dataframe Systems*. _eprint: 2001.00888.

[13] R Core Team. 2017. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/

[14] Erhard Rahm. 1996. Dynamic load balancing in parallel database systems. In *European Conference on Parallel Processing*. Springer, 37–52.

[15] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2019. Intermittent query processing. *Proceedings of the VLDB Endowment* 12, 11 (July 2019), 1427–1441. https://doi.org/10.14778/3342263.3342278

[16] Shivaram Venkataraman, Zongheng Yang, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[17] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets