Scaling Interactive Data Science Transparently with Modin

Devin Petersohn
Anthony D. Joseph, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-191
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-191.html

December 19, 2018

Acknowledgement

I would like to thank Professor Anthony Joseph for advising me during my candidacy. He provided support and advice above and beyond the requirements of an advisor. I would also like to thank Professor Ion Stoica for his support and for introducing me to this opportunity in the beginning.

I would also like to thank all of the contributors to Modin. I would like to especially thank the committers of Modin. These are exceptional undergraduate and recently graduated students, and their help has been instrumental in the success of this project.

Finally, and most of all, I would like to thank my wife, Madelyn. Her support through the most difficult times of this journey have been crucial to the success of both myself and this project. I am deeply indebted to her for this support.

Scaling Interactive Data Science Transparently with Modin

by

Devin Petersohn

A technical report submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Anthony D. Joseph, Chair
Professor Ion Stoica

Fall 2018

Scaling Interactive Data Science Transparently with Modin

Abstract

Scaling Interactive Data Science Transparently with Modin

by

Devin Petersohn

Master of Science in Computer Science

University of California, Berkeley

Professor Anthony D. Joseph, Chair

The combined growth of data science and big datasets has increased the performance requirements for running data analysis experiments and workflows. However, popular data science toolkits such as Pandas have not adapted to the technical demands of modern multicore, parallel hardware. As such, data scientists aiming to work with large quantities of data find themselves either suffering from libraries that under-utilize modern hardware or being forced to use big data processing tools that do not adapt well to the interactive nature of exploratory data analyses.

In this report we present the foundations of Modin, a library for large scale data analysis. Modin emphasizes performant, parallel execution on big datasets previously deemed unwieldy for existing popular toolkits, all while importantly maintaining an interface and set of semantics similar to existing interactive data science tools. The experiments presented in this report demonstrate promising results towards developing a new generation of performant data science tools built for parallel and distributed modern hardware.

Contents

List of Figures

Acknowledgments

I would like to thank Professor Anthony Joseph for advising me during my candidacy. He provided support and advice above and beyond the requirements of an advisor. I would also like to thank Professor Ion Stoica for his support and for introducing me to this opportunity in the beginning.

I would also like to thank all of the contributors to Modin, past and present. I would like to especially thank the committers of Modin. These are exceptional undergraduate and recently graduated students, and their help has been instrumental in the success of this project.

Finally, and most of all, I would like to thank my wife, Madelyn. Her support through the most difficult times of this journey have been crucial to the success of both myself and this project. I am deeply indebted to her for this support.

Chapter 1

Introduction

Data Science is one of the fastest growing cross-cutting fields with the advent of ubiquitous systems for computational statistics [6]. Tools such as the R programming language [14] and Pandas library [8] for Python have created accessible interfaces for manipulating data and performing exploratory data analysis (EDA) [16], the process of summarizing and understanding a dataset. These tools have enabled academic institutions and companies small and large to better extract insight from collected data. Further, the ability for such tools to run on commodity, highly available hardware, with open-source implementations and community support, has pushed even organizations without technical roots in mathematics or computer science to investigate data science solutions. Concurrently, the rate of data creation and collection has been accelerating over the past decade, and only seems to continue growing faster. This growth causes problems for traditional tools and systems, which are constrained to single-node, single-threaded implementations, despite having the number of cores on an individual machine constantly growing. This growth has spurred a set of responses in the distributed and parallel computation scope, with many computation frameworks [4, 3, 15, 10] embracing distributed computing across cluster systems for big data problems.

Most data scientists, who may have trained backgrounds in mathematics or statistics instead of computer science, are not trained systems engineers. As such, they may have the knowledge to manipulate or infer conclusions about data, but may be unfamiliar with best practices for engineering fast, scalable software to work with said data. Clearly it is unreasonable for a data scientist to work directly with low level libraries such as MPI, and even interfaces that computer scientists label as high level, such as MapReduce or Spark DataFrames may be out of reach for a typical data scientist. The average data scientist is familiar with abstractions such as Pandas, or R to work with their data. On the other hand, data scientists typically interact with their data in interactive environments, using ad-hoc methods. For example, in Python, a typical data science workflow may start by initially inspecting the collection or store of data by loading it into Pandas, performing a variety of cleaning methods based on their manual inspection of the data, and displaying summary statistics on the data, all in an interactive environment such as IPython [11] or Jupyter Notebook [7]. The actions they perform on their data are usually dependent on the results

they receive on previous views of said data.

This ad-hoc approach towards handling data is very disjoint from the view provided by many existing popular parallel computation libraries, such as Dask [13], Spark [3], and Tensorflow [1], which design computation around building and lazily executing entire dataflow graphs. The advantage to the aforementioned designs is the ability to perform query optimization on the entire query, and to better schedule individual portions of the query job, compared to eagerly evaluated frameworks. However, data scientists are accustomed to eager evaluation-based tools, which have more natural interoperability with interactive environments. As such, to better enable data science in the presence of big data and parallel computations, we propose Modin, an evolution of existing DataFrame libraries. We envision Modin as a successor to libraries such as Pandas, with the capability of providing scalable, performant tools on parallel systems while maintaining familiar semantics and an equivalent user-level interface. In this report, we will show that Modin can achieve up to 4x improvement on 4 cores.

Chapter 2

Motivation

2.1 Data Science Landscape

Today in Data Science, one of the most frustrating things that Data Scientists regularly encounter is the change in requirements and API when the data is large versus what they have been using. Figure 2.1 illustrates this disconnect. On the left, there are tools that are designed for single-node usage. These are the tools that are used to teach students how to do Data Science, and they are tools that every Data Scientist knows well. On the left, there is a sample of the tools that are used in "large scale" Data Science today. These tools solve their problems well, but they do not expose the same API as those tools on the left. Perhaps more importantly, these tools typically require some distributed computing knowledge, e.g. the number of partitions to choose for your data. Because Data Scientists are often domain experts in their field, they do not understand the complexities of shuffling and data partitioning.

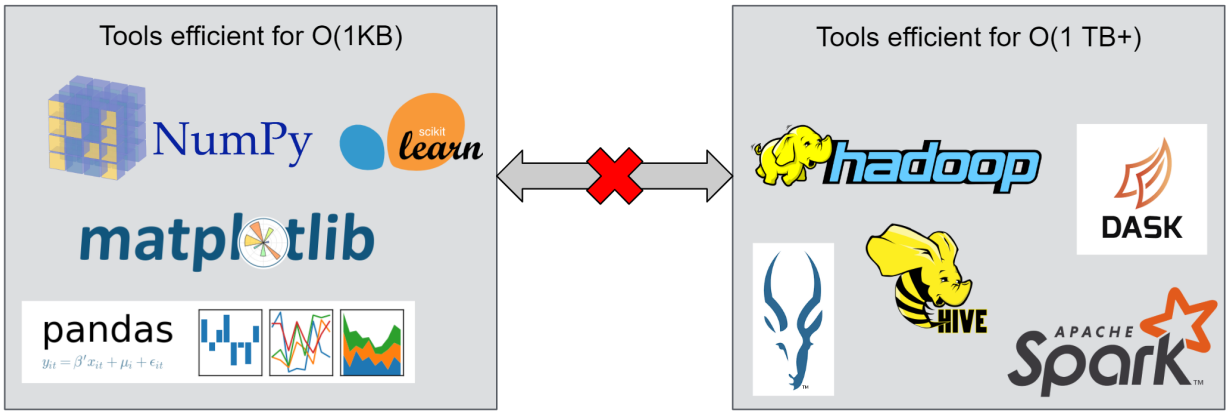


Figure 2.1: The Data Science Landscape at small and large scale

2.2 Problems tackled in Modin

The initial goal for Modin was to enable existing Python users performing exploratory data analysis using Pandas with the capability to scale their analyses from small datasets on the order of kilobytes, to massive datasets on the order of gigabytes. One key motivating factor behind the design was to ensure that users would not only be able quickly migrate to Modin, but also that any existing code, scripts, or notebooks would work unchanged regardless of the scale of data used. One could imagine this being very useful, for example, if a user would prefer to test hypotheses or workflows on small data for rapid iteration, and then use the same code for an entire massive dataset without modification.

To achieve that goal, we expose the same API as Pandas, which allows existing Pandas scripts or notebooks immediate interoperability with Modin through a single import change. Users who are already familiar with the Pandas interface will immediately know how to use Modin, as we engineered our API for concordance and feature parity with the existing Pandas API.

Apart from our user level API, to explain our engineering motivations in Modin, we first consider a typical EDA workflow to illustrate both our design focuses and design choices: Exploratory data analysis first begins with importing data, which involves reading data from an external source or store. Typically, these sources will be a text file, the most popular format being CSV or other related delimited formats, and other forms of tabular and columnar files. The user will then proceed to data wrangling, which typically involves examining the types of data imported, finding the amount of data, and printing out the first or last few rows of data, to provide a feel of the data. This step may also involve data cleaning or enacting other small mutations to the dataset in order to coerce the dataset into a usable form. After the data is successfully explored and cleaned, the user proceeds to exploratory data modeling, which involves operations performed across the dataset and deriving new datasets from the existing dataset to build insights from the original data. The user will then perform data visualization and communication, which involves generating charts and graphs, summarizing the insights gained from the experiment, or packaging the generated model into a usable form for later analysis. Lastly, the user may export results of the experiment, which usually involves writing derived or cleaned datasets or models to an external store. This process helps us organize our requirements for Modin into three categories.

Interactivity

The key realization for designing systems for EDA is that a typical EDA workflow is fundamentally an iterative process, as is the case with many data science workflows. In other words, the execution of the steps taken, or even the existence of steps themselves, can be changed on a whim based on the results from a previous step in the experiment. Likewise, the structure of said steps is fluid in practice; a data scientist may choose to explore different forms of manipulating or wrangling the dataset after receiving poor experimental results, or

may choose to import more data to augment the experiment. In this way, the process taken for data science is distinct enough from engineering that it could be classified more closely to traditional science fields, although engineering and data science are implied to be closely related.

It should be clear why data scientists favor tools that provide high degrees of interactivity, such as shells for Python or R or notebooks. Compare this paradigm to that presented by systems such as Dask, Tensorflow, or Spark, where actions are first declared to generate a dataflow graph, then executed only when the user explicitly initiates the graph computation. In the case of Dask or Tensorflow, explicit commands (df.compute in Dask, session.run in Tensorflow) are required to execute computations on the graph, while in Spark, computation only occurs when an action (a command generating output). These artifacts of lazy evaluation do not synchronize with interactive interfaces, since interactive users generally have the expectation cognitively of gradually constructing the end result solution. Likewise, debugging lazy evaluation systems is much more difficult than traditional eager systems, due to the inability to step through computation steps at a fine grain or the ability to inspect intermediary points of computation. Modin exposes a DataFrame API that acts exactly like DataFrame in eager evaluation interfaces such as R or Pandas. The user can execute individual operations on a DataFrame and use output functions to gradually verify the data within a DataFrame. Computation tasks within a DataFrame start immediately upon the function being called, instead of only when output is required. For an interactive user, who may be gradually composing their set of functions to perform on a DataFrame across multiple shell prompts or notebook blocks, this may lower the total computation time for the end result, as computation would be happening in the background for previous tasks while the user is still contemplating future functions to execute.

Concordance

The existing Pandas community is very large, for example there were 5 million questions asked by 1 million unique users on Stack Overflow in October 2017 [9], and it is cited as one of the largest growth factors for Python usage on Stack Overflow [17]. Of particular note is that a plurality of such additional users come from academia, who most likely use it for general purpose analysis as opposed to engineering, akin to the current role of R in academia. As such, we want to provide an interface that extends beyond familiar, making the Modin API as concordant with the existing Pandas API as possible. We strive to make it as simple as possible for a Pandas user to test out or convert their existing scripts and notebooks to Modin by only requiring a single import line change. This decreases barriers to entry for new Modin users, and also yields a roadmap to add support for other libraries that are frequently used with Pandas, such as scikit-learn or matplotlib, in the future.

Performance

With dataset sizes growing regularly into the tens of gigabytes, such as traffic data, genomic workloads, and weather data, even small-scale samples of working datasets may appear in the hundreds of megabytes. While popular libraries such as Pandas, which provide good interactivity, and work well on datasets in the tens of megabytes, their single-core implementations severely limit their performance when datasets grow into the multi-gigabyte range. By leveraging parallelism at the node and cluster level, we aim for Modin to enable exploratory analyses on large datasets with significant speedups in loading and processing time. Most importantly, we want to greatly increase performance while still also supporting the other points made above.

Chapter 3

Technical Foundation

3.1 User Interface

As previously described, the Modin interface emulates the existing Pandas interface. The user simply replaces their import statement, `import pandas as pd` with `import modin.pandas as pd`, and uses a new library with the same functional semantics as Pandas. Modin shadows existing Pandas class and top level functions, so under the hood `pandas.DataFrame` will be run using `modin.pandas.DataFrame`, `pandas.concat` will become `modin.pandas.concat`, and so on. Importantly, the user can be oblivious to this change: for an existing Pandas user with an existing script or notebook, replacing the import statement will yield the exact same functionality. For items that we feel are either inappropriate or inefficient to re-engineer for parallelism, such as Pandas data types and Pandas indexes, Modin transfers imports of such items, so `modin.pandas.Index` will simply be an alias for `pandas.Index`. This way, we cover all Pandas use cases a user might want to use, whether we decide to explicitly re-engineer it or not.

Noticeably absent from the framework is the API for specifying the number of cores in a system or cluster or data partitioning and distribution strategies. This is intentional, as Modin seeks to transparently distribute and compute user datasets in parallel. Users should not have to worry about tuning the library for their hardware, as Modin will perform optimization and parallel computation under the hood. Ideally, for the user, performance should be an afterthought and functionality should be their primary concern.

3.2 Execution Framework

On each parallel executor, Modin uses Pandas as its computation engine, specifically for each per-core DataFrame partition. Pandas is also used to ensure that all components, such as indexing or axis specification, remain fully concordant with themselves, without having to reimplement said functionality. Modin leverages Ray as the main underlying execution framework. It also preliminarily supports Dask.

Ray is a high-performance distributed execution framework targeted at large-scale machine learning and reinforcement learning applications. It achieves scalability and fault tolerance by abstracting the control state of the system in a global control store and keeping all other components stateless. Ray uses Apache Arrow [2], a modern columnar data serialization format and library specialized for in-memory data, and also uses Plasma In-Memory Object Store [12], a shared-memory distributed object store for use with Arrow. Ray's close integration with Apache Arrow allows for fast and efficient serialization of existing Pandas objects, greatly improving performance for Modin. Plasma Store's ability to support zero-copy data exchange across interprocess communication also helps improve performance for tasks on remote partitions. Rays usage of ObjectIDs to represent pieces of data in the object store allows Modin to maintain remote handles to data partitions and communicate said partitions across separate processes without explicitly transferring data over an RPC. Lastly, Ray has the ability to scale to hundreds or thousands of nodes in a single cluster, enabling it to operate on massive datasets. We currently are not focused on cluster performance, and do not investigate cluster performance for Modin in this report. Significantly, Pandas users have remained Pandas users specifically because they care mostly about single-node performance. Therefore, we instead are currently investigating optimizing single-machine, multi-core performance, an area which Modin still demonstrates large speedups.

Chapter 4

Technical Architecture

4.1 DataFrame Partitioning

The Modin DataFrame architecture follows in the footsteps of modern architectures for database and high performance matrix systems. We chose a partitioning schema that partitions along both columns and rows because it gives Modin flexibility and scalability in both the number of columns and the number of rows supported. Figure 4.1 illustrates this concept.

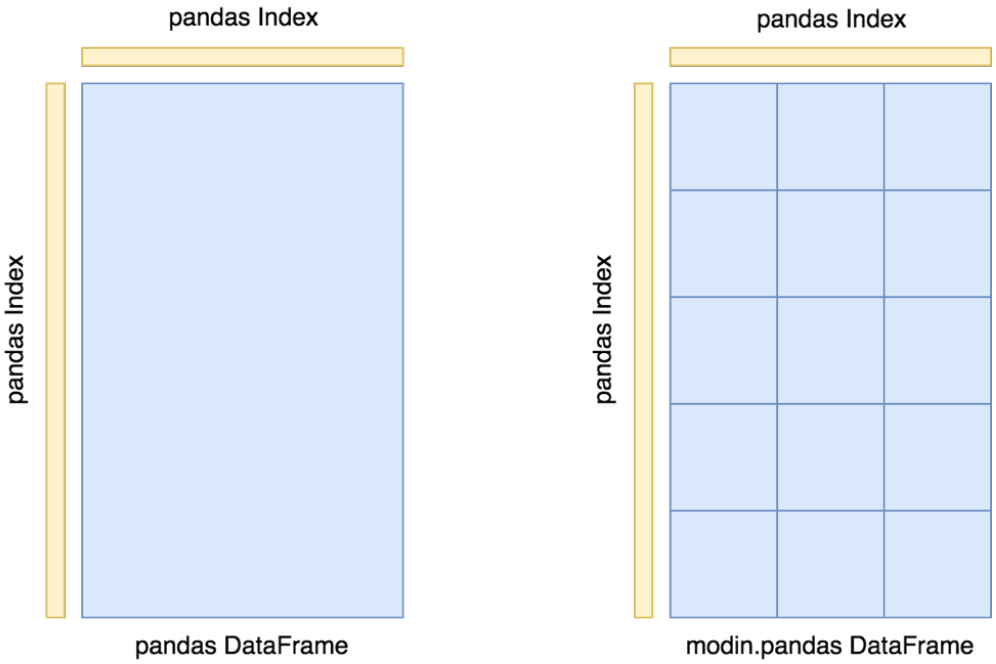


Figure 4.1: Comparison between pandas and a partitioned Modin DataFrame

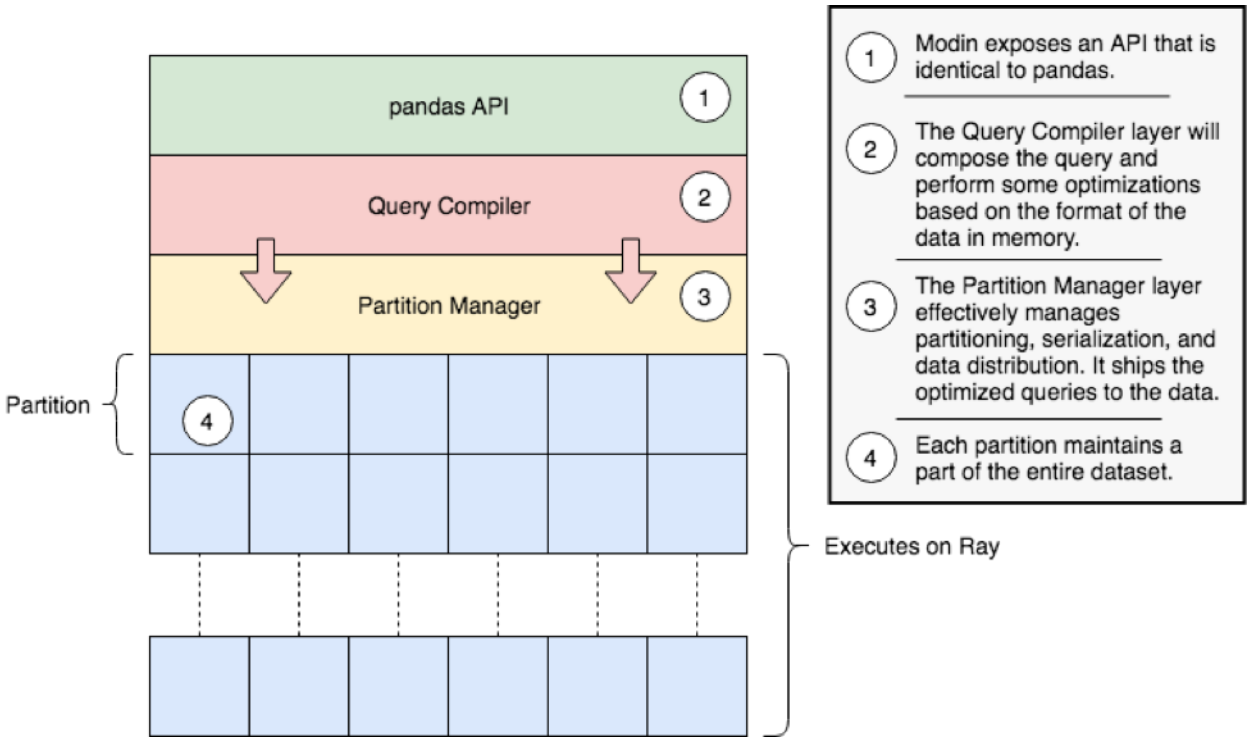


Figure 4.2: The system architecture of Modin

Indexing

We currently use the `pandas.Index` object for both indexing columns and rows. In the future, we will implement a distributed, pandas-compatible Index object in order remove this scaling limitation from the system. It does not start to become a problem until you are operating on more than 10s of billions of columns or rows, so most workloads will not be affected by this scalability limit. With the default index `pandas.RangeIndex` there is a fixed memory overhead (200 bytes) and there will be no scalability issues with the index.

4.2 Sytem Architecture

Figure 4.2 outlines the general architecture for the implementation of Modin.

Modin is logically separated into different layers that represent the hierarchy of a typical Database Management System. At the highest layer, we expose the pandas API. This is discussed in many other parts of the documentation, so we will not go into detail for it here. We will go through all of the other components in some detail below, starting with the next highest layer, the Query Compiler.

Query Compiler

The Query Compiler receives queries from the pandas API layer. The API layers responsibility is to ensure clean input to the Query Compiler. The Query Compiler must have knowledge of the in-memory format of the data (currently a pandas DataFrame) in order to efficiently compile the queries.

The Query Compiler is responsible for sending the compiled query to the Partition Management layer. In this design, the Query Compiler does not need to know what the execution framework is (Ray in this case), and gives the control of the partition layout to a lower layer.

At this layer, operations can be performed lazily. Currently, Modin executes most operations eagerly in an attempt to behave as pandas does. Some operations, e.g. `transpose` are expensive and create full copies of the data in-memory. In these cases, we keep some metadata about the operations and queue them up so they are somewhat lazy. In the future, we plan to add additional query planning and laziness to Modin to ensure that queries are performed efficiently.

Partition Manager

The Partition Manager is responsible for the data layout and shuffling, partitioning, and serializing the tasks that get sent to each partition.

The Partition Manager can change the size and shape of the partitions based on the type of operation. For example, certain operations are complex and require access to an entire column or row. The Partition Manager can convert the block partitions to row partitions or column partitions. This gives Modin the flexibility to perform operations that are difficult in row-only or column-only partitioning schemas.

Another important component of the Partition Manager is the serialization and shipment of compiled queries to the Partitions. It maintains metadata for the length and width of each partition, so when operations only need to operate on or extract a subset of the data, it can ship those queries directly to the correct partition. This is particularly important for some operations in pandas which can accept different arguments and operations for different columns, e.g. `fillna` with a dictionary.

Partition

Partitions are responsible for managing a subset of the DataFrame. As mentioned above, the DataFrame is partitioned both row and column-wise. This gives Modin scalability in both directions and flexibility in data layout. There are a number of optimizations in Modin that are implemented in the partitions. Partitions are specific to the execution framework and in-memory format of the data. This allows Modin to exploit potential optimizations across both of these. These optimizations are explained further below.

4.3 Pandas on Ray

Pandas on Ray is the component of Modin that runs on the Ray execution Framework. Currently, the in-memory format for Pandas on Ray is a pandas DataFrame on each partition. There are a number of Ray-specific optimizations we perform, which are explained below. Currently, Ray is the only execution framework fully supported on Modin. Dask is preliminiarily supported. There are additional optimizations we can do on the pandas in-memory format. Those are also explained below.

Ray-specific optimizations

Serialization of tasks and parameters

The optimization that improves the performance the most is the pre-serialization of the tasks and parameters. This is primarily applicable to map operations. We have designed the system such that there is a single remote function that accepts a serialized function as a parameter and applies it to a partition. The operation will be serialized separately for each partition if we do not call `ray.put` on it first. The `BaseBlockPartitions` abstract class exposes a unified way to preprocess functions. The primary purpose of the preprocess abstraction is to allow for optimizations such as this.

Memory Management

The second optimization we perform is related to how Ray and Arrow handle memory. Historically, pandas has used a significant amount of memory, and tends to create copies even for some simple computations. The plasma store in Arrow is immutable, which can cause problems for certain workloads, as objects that are no longer in scope for the Python application can be kept around and consume memory in Arrow. To resolve this issue, we free memory once the reference count for that memory goes to zero. This component is still experimental, but we plan to keep iterating on it to make Modin as memory efficient as possible.

Pandas-specific optimizations

Indexing

Internally, since each partition contains a pandas DataFrame, the indexing information for both rows and columns would be duplicated for every partition. Because we use block partitions layout, it would be replicated as many times as there were blocks. To avoid this issue, we use a `pandas.RangeIndex` internally, which has a fixed memory cost.

This optimization is also used to determine which columns or rows were dropped during a `dropna` or other similar operation. We use the `pandas.RangeIndex` internal to the partitions to communicate the missing values back to the external `Index`.

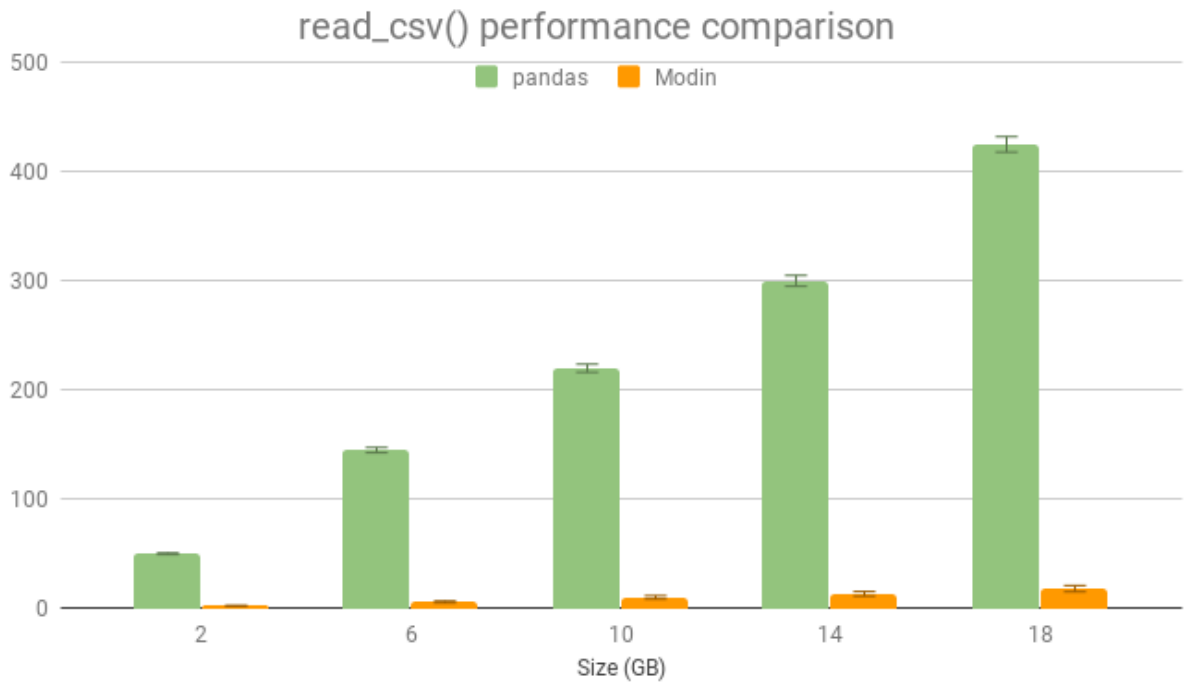


Figure 4.3: Performance of read_csv on 144 cores versus pandas. 5 trials were run for each size. This plot shows that read_csv is bottlenecked by the CPU.

4.4 Data Ingest

Data ingestion into Modin is done similarly to Pandas, and can be performed by turning existing Python objects into DataFrames, or importing from external files. Coercing existing Python objects, such as lists, dictionaries, and NumPy arrays currently uses the Pandas DataFrame constructor to turn said object into a Pandas DataFrame, which we refer to as a row partition. Each row partition is then further partitioned column-wise into block partitions. For file import operations, each file is first logically partitioned based on the size of the file in bytes, and Pandas import functions are used to turn those logical partitions into block partitions remotely. For example, for `modin.read_csv`, the file is initially marked for partitioning by line number and separate remote tasks perform `pandas.read_csv` on those logical partitions. This generates a set of disjoint row partitions for the DataFrame, and the row partitions are further converted into block partitions. This allows Modin to ingest data from an external source in parallel, performing reads much faster. Figure 4.3 illustrates an extreme case of the improvement in performance on a large server.

4.5 Data Manipulation

Applymap Functions

Applymap functions in Pandas are actions on a DataFrame that execute a function or lambda on each element of the DataFrame distinctly, returning a DataFrame of the same dimensions with modified elements. Functions such as applymap, isna, or scalar arithmetic operations on a DataFrame fall into this category. Due to the axis-invariant nature of the operation, operations are broadcast to each of the individual block partitions, returning another grid of block partitions that we package into a new Modin DataFrame. Notably, the DataFrame dimensions do not change, nor do the indexes, so the dimensionality of the block partition grid, as well as the index metadata objects, are preserved in the newly created DataFrame, saving some computation and communication time.

Axis-Variant Functions

Axis-variant functions in Pandas are actions that operate on entire rows or columns of a DataFrame, as a series, and return a result either as a series or single item. Functions such as sum, mean, and describe fall into this category. To make the execution of such functions simple, intuitive, and robust, the block partitions are first combined into entire rows or columns, dependent on the axis of operation. The action is then instead invoked as a remote task onto each of the row or column partitions. This simplifies the emulation of those functions on partitions while maintaining the parallel execution across distinct partitions. For functions that reduce a series to a single scalar value, the output is returned as a concatenation of the resulting series of values from each row/column partition. Otherwise, for functions that return a complete partition, a new DataFrame is built from the concatenation of the partitions.

Mutation Functions

Mutation functions in Pandas are actions that change values within a DataFrame at a certain location within the indexes. Functions such as insert, remove, and the delitem override fall into this category. Execution of these functions is generally tricky, since mutations require an inplace modification of a subset of block partitions, and due to the immutable nature of Ray's object store, these partitions must be replaced by completely new partitions.

Initially, the index metadata objects are consulted to find the coordinates for modification in both axes, if necessary. The requisite blocks are also combined into rows of column partitions, if the action involves a series. Using the coordinates, a function is then dispatched to the correct partitions to insert, remove, or modify the values specified, generating new partitions, and new partition ObjectIDs. Those new ObjectIDs are used to replace the existing stale partitions in the DataFrame.

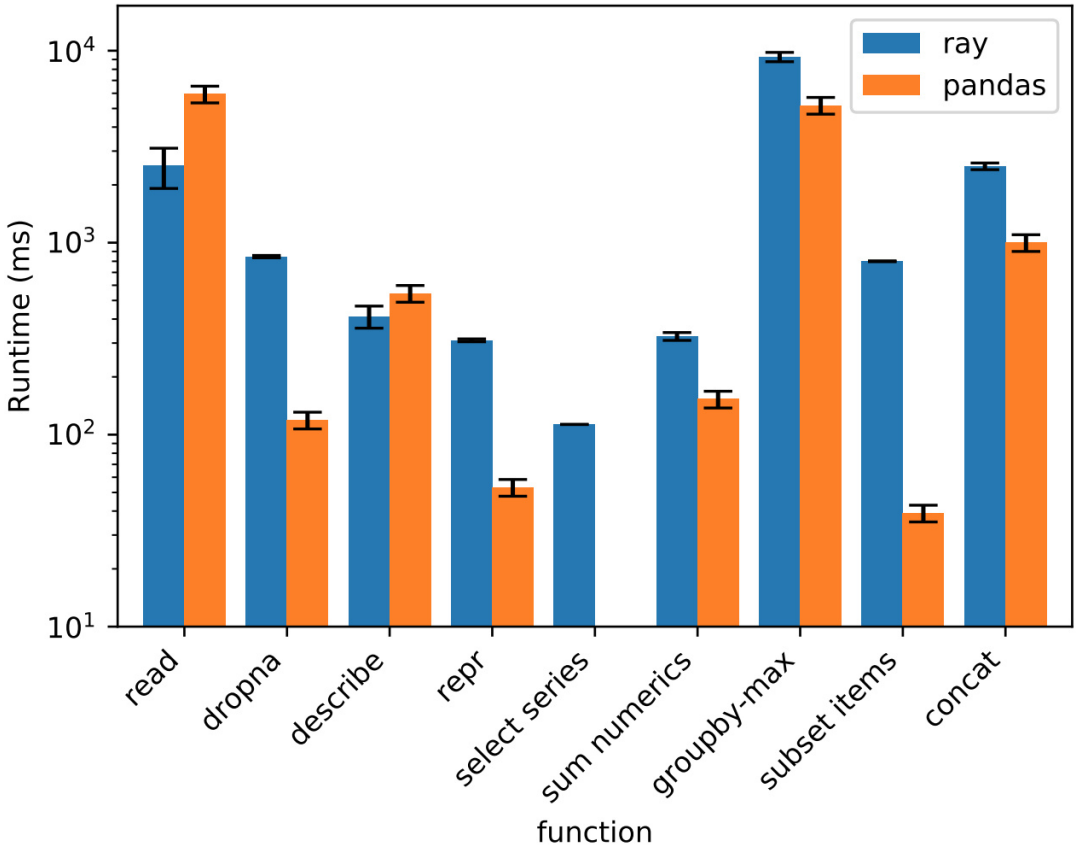


Figure 4.4: Common workflow function comparison. All operations were test 5 times each on a 250MB dataset. Note that for "select series", the Pandas operation consistently ran in under 200 μs.

Grouping Functions

Grouping functions in Pandas are the groupby function and all associated actions performed on grouped data. Since the DataFrame is logically partitioned across both axes, and the partitions currently do not follow any logical partitions with respect to the axis indexes or the underlying data, grouping can only be done by grouping across entire column partitions. Thus, the block partitions are first combined into column partitions and the specified grouping data is broadcast to each column partition. Since the grouping data is grouped in a deterministic fashion, performing individual remote groupings on each column partition should yield partition groups that can be reliably combined as well. All grouping operations are wrapped in a local driver object to manage any functions performed on the partition groupings, emulating the Pandas DataFrameGroupBy object.

Join Functions

Join functions in Pandas encompass all actions that require some sort of index coercion between two differently indexed DataFrames. This includes join, concat, and merge, but also covers rudimentary functionality such as add and other inter-DataFrame math operations. Since these functions implicitly perform an outer join on both input DataFrames passed in, they share much of the same logic with the concatenation and joining functions. The join is initially performed on the indexes for the two DataFrames, and then both DataFrames are reindexed into the new joined indexes. This approach is used to ensure that the two DataFrames are put into a co-partitioned state, such that the labels across their rows match. Depending on the function being called, the resulting DataFrames are then either concatenated or combined elementwise.

Export Functions

Export functions in Pandas are actions that take data stored in a DataFrame and export them in some usable representation, whether that is to the user console, to a different Python object, or serialized to disk. Currently, while some functions, such as the repr override, have subtle optimizations to reduce the amount of data covered, most export functions are handled through Pandas. For example, `to_csv` performs a `to_csv` on each row partition in serial order to a single file. There is room to optimize these methods in the future, as this was not our primary focus on the initial release of Modin

Chapter 5

API Compatability and Completeness

5.1 Study on Pandas usage

It is easy in a young project to start by implementing the components of the API that offer easy performance wins or are low effort. In order to avoid bias in the order of implementation and maximize coverage of typical use cases, we decided to take a data-driven approach to API completion. We went to Kaggle and downloaded the 1800 most upvoted Python notebooks and scripts and scraped them for the Pandas DataFrame methods they invoke.

We opted to implement the 90th percentile of operations first, such that the most popular methods would be completed. Currently, Modin supports about 180 of the over 280 Pandas DataFrame methods. The list of currently supported APIs, details, and limitations of these implementations can be found in the documentation. We felt it was important that the implementation of the project be community-driven, rather than driven by our own biases.

Figure 5.1 shows the most-used methods in pandas based on our study.

5.2 API Completeness

Currently, we support 71% of the pandas API. The exact methods we have implemented are listed in the documentation. We have taken a community-driven approach to implementing new methods. Based on the study above, Modin currently supports 93% of the pandas API, and we are actively expanding the API.

5.3 Defaulting to Pandas

The remaining unimplemented methods default to pandas. This allows users to continue using Modin even though their workloads contain functions not yet implemented in Modin. Figure 5.2 illustrates how the transformation is done during an operation that defaults to pandas.

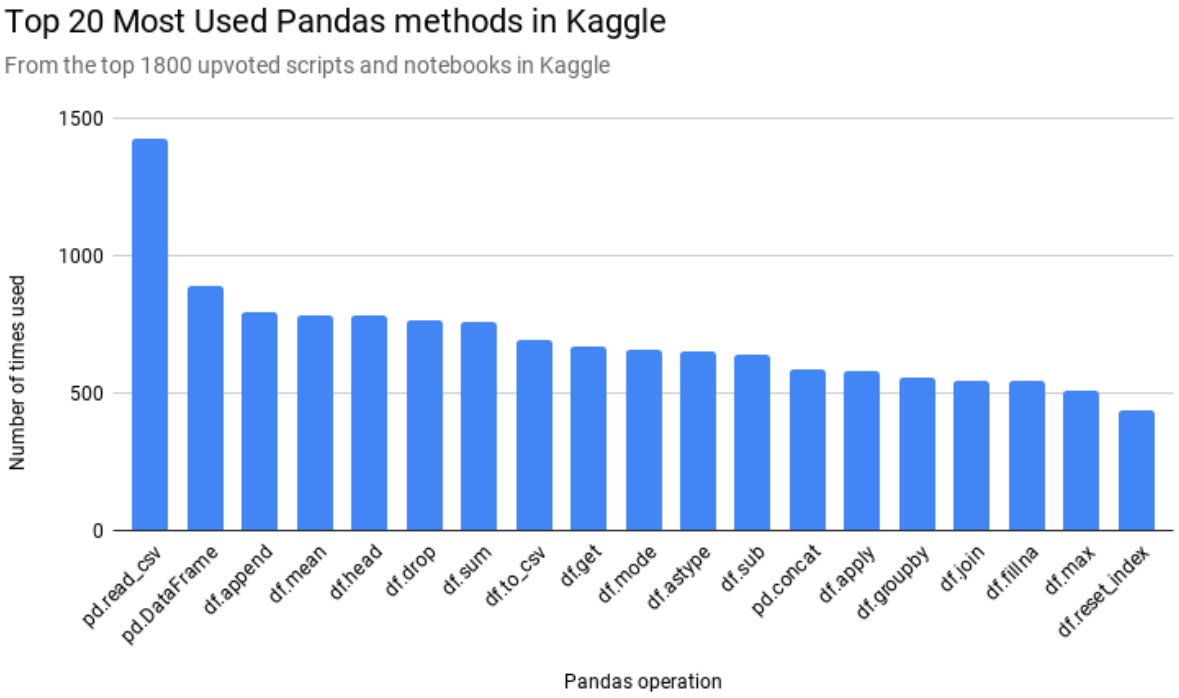
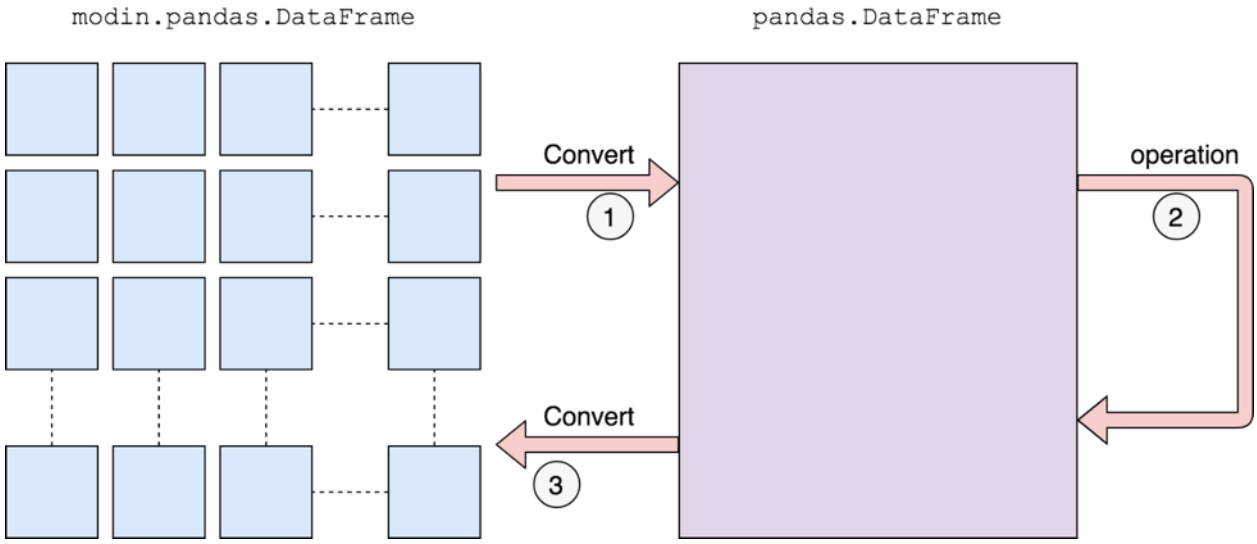


Figure 5.1: The 20 Most used pandas methods in Kaggle

Figure 5.2: A data flow diagram during an operation that defaults to pandas



We first convert to a pandas DataFrame, then perform the operation. There is a performance penalty for going from a partitioned Modin DataFrame to pandas because of the communication cost and single-threaded nature of pandas. Once the pandas operation has completed, we convert the DataFrame back into a partitioned Modin DataFrame. This way, operations performed after something defaults to pandas will be optimized with Modin.

5.4 Using Modin

As we previously mentioned, Modin is an early stage DataFrame library that wraps pandas and transparently distributes the data and computation, accelerating pandas workflows with one line of code change. The user does not need to know how many cores their system has, nor do they need to specify how to distribute the data. In fact, users can continue using their previous pandas notebooks while experiencing a considerable speedup from Modin, even on a single machine. Only a modification of the import statement is needed, as we demonstrate below due to the multicore nature of modern machines.

Using Modin on a single node

Once you import the library, you will see something similar to the following output:

```
>>> import modin.pandas as pd

Waiting for redis server at 127.0.0.1:14618 to respond...
Waiting for redis server at 127.0.0.1:31410 to respond...
Starting local scheduler with the following resources: {'CPU': 4, 'GPU': 0}.
```

Listing 5.1: Modin Startup

After executing `import modin.pandas as pd`, the user is ready to begin running their pandas pipeline as they were before. Modin automatically starts up a parallel environment for the user.

Exceeding memory (Out of core pandas)

When working with very large files or exceeding available memory, the user may change the primary location of the DataFrame. To exceed memory, the user can use your disk as backup for the memory. This Modin API is experimental.

Instead of limiting the size of DataFrames to the amount of available memory, Modin can back the memory with disk:

```
import ray
num_bytes = 2**40 # Make sure you have disk space to do this!
ray.init(plasma_directory="/tmp", object_store_memory=num_bytes)
import modin.pandas as pd
```

Listing 5.2: Out of Core Modin

Setting `plasma_directory="/tmp"` uses the disk for storing the DataFrame and setting `object_store_memory` sets the maximum size of the plasma store. Under the hood, Modin is using the operating system to page the necessary DataFrame blocks in and out of memory. This will impact performance for smaller DataFrames, but it will enable use of larger than memory DataFrames on a laptop or desktop.

Reducing or limiting the resources Modin can use

By default, Modin will use all of the resources available on the machine. It is possible, however, to limit the amount of resources Modin uses to free resources for another task or user. Here is how to limit the number of CPUs Modin uses:

```
1 import ray
2 ray.init(num_cpus=4)
3 import modin.pandas as pd
```

Listing 5.3: Limiting Resources for Modin

Specifying `num_cpus` limits the number of processors that Modin uses. It is possible to specify more processors than you have available on the machine, however this generally will not improve the performance (and might end up hurting the performance of the system).

Chapter 6

Future Work

6.1 API

Pandas

In the future, we aim to have a complete coverage of the entire Pandas API. This is a lofty goal, particularly because of the breadth of algorithms and use-cases that pandas currently covers.

SQL

As we previously mentioned, Modin's architecture is specifically set up to be able to support multiple APIs. Currently, the only way to perform SQL queries on a DataFrame object is to copy the data into a new structure or save it as a file. With Modin, we aim to expose an API that is familiar to users and allow them to operate on it in the way that is most comfortable to them. We have implemented a simple way to interact with a DataFrame using SQL and expose an example to show how it would work. The example is posted below.

```
1  import modin.sql as sql
2  conn = sql.connect("db_name")
3  c = conn.cursor()
4  c.execute("CREATE TABLE example (col1, col2, column 3, col4)")
5  c.execute("INSERT INTO example VALUES ('1', 2.0, 'A String of information', True)")
6
7    col1  col2                 column 3  col4
8  0    1   2.0  A String of information  True
9
10 c.execute("INSERT INTO example VALUES ('6', 17.0, 'A String of new information', False)")
11
12   col1  col2    column 3                      col4
13 0    1   2.0    A String of information       True
14 1    6  17.0    A String of new information   False
```

Listing 6.1: SQL API example for Modin

6.2 Additional Runtimes for Modin

Compute Engines

Modin has been designed to run on multiple backends. Currently, Ray is the primary runtime for Modin, and we have preliminary support for Dask. In the future, we envision a world where Modin is the front end for all backends that support some sort of computation. After we have strong stability in the Ray backend, we will likely move to Spark, which has a large number of Data Scientist users. The overwhelming majority of users we have talked with about Spark typically complain about its complex API. They all agree that being able to use the pandas API on Spark would make them much more productive, but there are systems-level challenges that make this difficult today.

SQL Systems

In addition to compute engine-style runtimes, we would also like to support database runtimes. The primary goal here is to be able to ingest data from its source, wherever it is, and perform manipulations at interactive latencies. The first step is to create an Intermediate Representation (IR) that can communicate the user's intentions to these systems such that it is interpreted and the correct output is delivered back.

6.3 Query Planning

Another important aspect to allow Modin to run at-scale is to perform some query optimizations. In order to do this effectively, we need to have an efficient way of operating on data in-memory. This section explains at a high level what our plans for the future are with respect to query planning.

Query Rewriting

Query rewriting is a common way in SQL systems to gain performance. The general idea is to take a query that someone has written and rearrange the operations to make the overall performance of the query perform more quickly. In pandas, there is a challenge to doing this, particularly because there are often multiple ways and orders to perform a given operation, and each of these has its own performance profile. We plan to solve this problem with a neural net that maps queries to each other so that our query solver can pick the most efficient path for query completion. As we add SQL systems to the backend, these will likely not benefit from query planning primarily because they typically include their own planners.

Query Optimization in Interactive Environments

Another very interesting avenue we would like to investigate is how to optimize queries in an interactive environment. Most of the time, users expect some computation to begin at the point that they submit the query. In a notebook or interpreter, this typically means that the main thread will be blocked until the computation is complete. Most systems solve the query planning problem by making the system lazy, meaning that nothing executes until the user explicitly tells it to. This puts extra burden on the end user that we do not want, and potentially wastes significant waiting time. We would like to explore a query planner that incrementally plans the query as new operations come in, and that can eagerly execute queries that manipulate data on a smaller scale[5]. We believe that the planner should not interfere with interactivity, and expect that it would improve the efficiency of the Data Scientist.

Chapter 7

Conclusion

In this report, we introduced Modin, a DataFrame library that can scale pandas workflows and requires changing only one line of code. Modin is a DataFrame library that wraps Pandas and transparently distributes the data and computation. The user does not need to know how many cores their system or cluster has, nor do they need to specify how to distribute the data. In fact, users can continue using their previous Pandas notebooks while experiencing a considerable speedup from Modin, even on a single machine. As we demonstrated in this report, Modin can show speed improvements of up to 4x on a 4-core laptop compared to Pandas.

Modin is targeted towards pandas users who are looking to improve performance and see faster runtimes without having to switch to another API. We are aggressively working to achieve functional parity with Pandas full API, and have so far implemented about 71% of the API. The code for Modin can be found on the Modin GitHub page: github.com/modin-project/modin.

Bibliography

[1] Martín Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[2] *Apache Arrow*. URL: `https://arrow.apache.org/`.

[3] Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.

[4] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[5] Joseph M. Hellerstein et al. "Adaptive query processing: Technology in evolution". In: *IEEE Data Eng. Bull.* 23.2 (2000), pp. 7–18.

[6] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*. Vol. 1. Microsoft research Redmond, WA, 2009.

[7] Thomas Kluyver et al. "Jupyter Notebooks-a publishing format for reproducible computational workflows." In: *ELPUB*. 2016, pp. 87–90.

[8] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[9] *Meet the Man behind the Most Important Tool in Data Science*. URL: `qz.com/1126615/the-story-of-the-most-important-tool-in-datascience/`.

[10] Christopher Olston et al. "Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1099–1110.

[11] Fernando Pérez and Brian E Granger. "IPython: a system for interactive scientific computing". In: *Computing in Science & Engineering* 9.3 (2007).

[12] *Plasma In-Memory Object Store*. URL: `https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store/`.

[13] Matthew Rocklin. "Dask: Parallel computation with blocked algorithms and task scheduling". In: *Proceedings of the 14th Python in Science Conference*. 130-136. Citeseer. 2015.

[14] R Core Team et al. "R: A language and environment for statistical computing". In: (2013).

[15] Ashish Thusoo et al. "Hive-a petabyte scale data warehouse using hadoop". In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE. 2010, pp. 996–1005.

[16] John W Tukey. *Exploratory data analysis*. Vol. 2. Reading, Mass., 1977.

[17] *Why is Python growing so quickly?* URL: `https://stackoverflow.blog/2017/09/14/python-growing-quickly/`.