Modin

Release 0.30.0+5.g43eff5b.dirty

Modin contributors

May 17, 2024

CONTENTS

Note:

Estimated Reading Time: 10 minutes

You can follow along this tutorial in a Jupyter notebook here.

CHAPTER

ONE

INSTALLATION

Note:

Estimated Reading Time: 15 minutes

If you already installed Modin on your machine, you can skip this section.

There are several ways to install Modin. Most users will want to install with pip or using conda tool, but some users may want to build from the main branch on the GitHub repo. The main branch has the most recent patches, but may be less stable than a release installed from pip or conda.

1.1 Installing with pip

1.1.1 Stable version

Modin can be installed with pip on Linux, Windows and MacOS. To install the most recent stable release run the following:

```
pip install -U modin # -U for upgrade in case you have an older version
```

Modin can be used with *Ray*, *Dask*, *Unidist* engines. If you don't have Ray, Dask or Unidist installed, you will need to install Modin with one of the targets:

```
pip install "modin[ray]" # Install Modin dependencies and Ray to run on Ray
pip install "modin[dask]" # Install Modin dependencies and Dask to run on Dask
pip install "modin[mpi]" # Install Modin dependencies and MPI to run on MPI through␣
↪unidist
pip install "modin[all]" # Install Ray and Dask
```

To get Modin on MPI through unidist (as of unidist 0.5.0) fully working it is required to have a working MPI implementation installed beforehand. Otherwise, installation of modin[mpi] may fail. Refer to Installing with pip section of the unidist documentation for more details about installation.

Note: Since Modin 0.30.0 we use a reduced set of Ray dependencies: ray instead of ray[default]. This means that the dashboard and cluster launcher are no longer installed by default. If you need those, consider installing ray[default] along with modin[ray].

Modin will automatically detect which engine you have installed and use that for scheduling computation!

1.1.2 Release candidates

Before most major releases, we will upload a release candidate to test and check if there are any problems. If you would like to install a pre-release of Modin, run the following:

```
pip install --pre modin
```

These pre-releases are uploaded for dependencies and users to test their existing code to ensure that it still works. If you find something wrong, please raise an issue or email the bug reporter: bug_reports@modin.org.

1.1.3 Installing specific dependency sets

Modin has a number of specific dependency sets for running Modin on different execution engines and storage formats or for different functionalities of Modin. Here is a list of dependency sets for Modin:

```
pip install "modin[ray]" # If you want to use the Ray execution engine
```

```
pip install "modin[dask]" # If you want to use the Dask execution engine
```

```
pip install "modin[mpi]" # If you want to use MPI through unidist execution engine
```

1.1.4 Consortium Standard-compatible implementation based on Modin

```
pip install "modin[consortium-standard]"
```

1.1.5 Installing on Google Colab

Modin can be used with Google Colab via the `pip` command, by running the following code in a new cell:

```
!pip install "modin[all]"
```

Since Colab preloads several of Modin's dependencies by default, we need to restart the Colab environment once Modin is installed by either clicking on the `"RESTART RUNTIME"` button in the installation output or by run the following code:

```
# Post-install automatically kill and restart Colab environment
import os
os.kill(os.getpid(), 9)
```

Once you have restarted the Colab environment, you can use Modin in Colab in subsequent sessions.

Note that on the free version of Colab, there is a limit on the compute resource. To leverage the full power of Modin, you may have to upgrade to Colab Pro to get access to more compute resources.

1.2 Installing with conda

1.2.1 Using conda-forge channel

Modin releases can be installed using `conda` from conda-forge channel. Starting from 0.10.1 it is possible to install modin with chosen engine(s) alongside. Current options are:

Package name in conda-forge	Engine(s)	Supported OSs
modin	Dask	Linux, Windows, MacOS
modin-dask	Dask	Linux, Windows, MacOS
modin-ray	Ray	Linux, Windows
modin-mpi	MPI through unidist	Linux, Windows, MacOS
modin-all	Dask, Ray, Unidist	Linux

Note: Since Modin 0.30.0 we use a reduced set of Ray dependencies: `ray-core` instead of `ray-default`. This means that the dashboard and cluster launcher are no longer installed by default. If you need those, consider installing `ray-default` along with `modin-ray`.

For installing Dask, Ray and MPI through unidist engines into conda environment following command should be used:

```
conda install -c conda-forge modin-ray modin-dask modin-mpi
```

All set of engines could be available in conda environment by specifying:

```
conda install -c conda-forge modin-all
```

or explicitly:

```
conda install -c conda-forge modin-ray modin-dask modin-mpi
```

Refer to Installing with conda section of the unidist documentation for more details on how to install a specific MPI implementation to run on.

`conda` may be slow installing `modin-all` or combitations of execution engines so we currently recommend using libmamba solver for the installation process. To do this install it in a base environment:

```
conda install -n base conda-libmamba-solver
```

Then it can be used during installation either like

```
conda install -c conda-forge modin-ray modin- --experimental-solver=libmamba
```

or starting from conda 22.11 and libmamba solver 22.12 versions

```
conda install -c conda-forge modin-ray --solver=libmamba
```

1.3 Installing from the GitHub main branch

If you'd like to try Modin using the most recent updates from the main branch, you can also use `pip`.

```
pip install "modin[all] @ git+https://github.com/modin-project/modin"
```

This will install directly from the repo without you having to manually clone it! Please be aware that these changes have not made it into a release and may not be completely stable.

If you would like to install Modin with a specific engine, you can use `modin[ray]` or `modin[dask]` or `modin[mpi]` instead of `modin[all]` in the command above.

1.4 Windows

All Modin engines are available both on Windows and Linux as mentioned above. Default engine on Windows is *Ray*. It is also possible to use Windows Subsystem For Linux (WSL), but this is generally not recommended due to the limitations and poor performance of Ray on WSL, a roughly 2-3x worse than native Windows.

1.5 Building Modin from Source

If you're planning on *contributing* to Modin, you will need to ensure that you are building Modin from the local repository that you are working off of. Occasionally, there are issues in overlapping Modin installs from pypi and from source. To avoid these issues, we recommend uninstalling Modin before you install from source:

```
pip uninstall modin
```

To build from source, you first must clone the repo. We recommend forking the repository first through the GitHub interface, then cloning as follows:

```
git clone https://github.com/<your-github-username>/modin.git
```

Once cloned, `cd` into the `modin` directory and use `pip` to install:

```
cd modin
pip install -e .
pip install -e ".[all]"  # will install dependencies for all engines
```

CHAPTER

TWO

USING MODIN

In this section, we show how Modin can be used to accelerate your pandas workflows on a single machine up to multiple machines in a cluster setting.

2.1 Using Modin Locally

Note:

Estimated Reading Time: 5 minutes

You can follow along this tutorial in the Jupyter notebook.

In our quickstart example, we have already seen how you can achieve considerable speedup from Modin, even on a single machine. Users do 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 existing pandas code** while experiencing a considerable speedup from Modin, even on a single machine.

To use Modin on a single machine, only a modification of the import statement is needed. Once you've changed your import statement, you're ready to use Modin just like you would pandas, since the API is identical to pandas.

```
# import pandas as pd
import modin.pandas as pd
```

That's it. You're ready to use Modin on your previous pandas workflows!

2.1.1 Advanced: Configuring the resources Modin uses

Modin automatically check the number of CPUs available on your machine and sets the number of partitions to be equal to the number of CPUs. You can verify this by running the following code:

```
import modin
print(modin.config.NPartitions.get()) #prints 16 on a laptop with 16 physical cores
```

Modin fully utilizes the resources on your machine. To read more about how this works, see *Why Modin?* page for more details.

Since Modin will use all of the resources available on your machine by default, at times, it is possible that you may like to limit the amount of resources Modin uses to free resources for another task or user. Here is how you would limit the number of CPUs Modin used in your bash environment variables:

```
export MODIN_CPUS=4
```

You can also specify this in your python script with `os.environ`:

```
import os
os.environ["MODIN_CPUS"] = "4"
import modin.pandas as pd
```

If you're using a specific engine and want more control over the environment Modin uses, you can start Ray or Dask in your environment and Modin will connect to it.

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

Specifying `num_cpus` limits the number of processors that Modin uses. You may also specify more processors than you have available on your machine; however this will not improve the performance (and might end up hurting the performance of the system).

Note: Make sure to update the `MODIN_CPUS` configuration and initialize your preferred engine before you start working with the first operation using Modin! Otherwise, Modin will opt for the default setting.

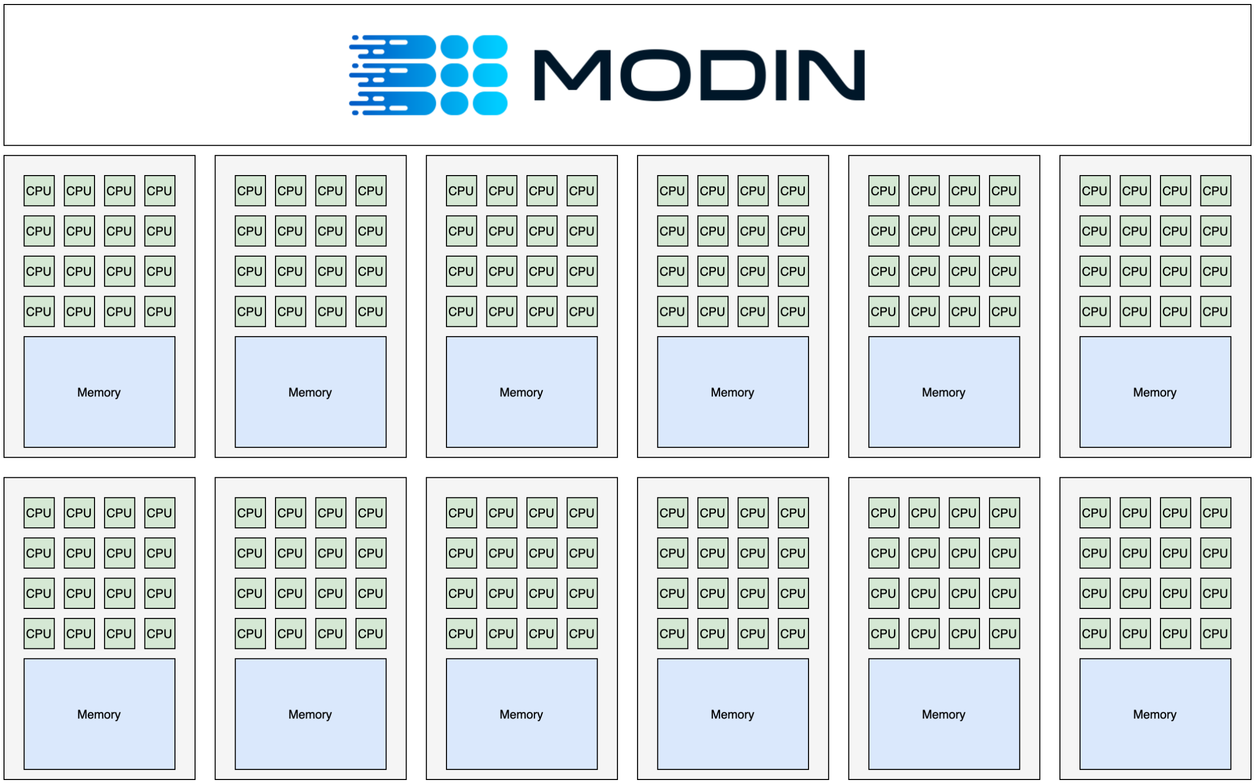
2.2 Using Modin in a Cluster

Note:

Estimated Reading Time: 15 minutes

Often in practice we have a need to exceed the capabilities of a single machine. Modin works and performs well in both local mode and in a cluster environment. The key advantage of Modin is that your python code does not change between local development and cluster execution. Users are not required to think about how many workers exist or how to distribute and partition their data; Modin handles all of this seamlessly and transparently.

Note: It is possible to use a Jupyter notebook, but you will have to deploy a Jupyter server on the remote cluster head node and connect to it.



2.2.1 Extra requirements for AWS authentication

First of all, install the necessary dependencies in your environment:

```
pip install boto3
```

The next step is to setup your AWS credentials. One can set `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` and `AWS_SESSION_TOKEN` (Optional) (refer to AWS CLI environment variables to get more insight on this) or just run the following command:

```
aws configure
```

2.2.2 Starting and connecting to the cluster

This example starts 1 head node (m5.24xlarge) and 5 worker nodes (m5.24xlarge), 576 total CPUs. You can check the Amazon EC2 pricing page.

It is possble to manually create AWS EC2 instances and configure them or just use the Ray CLI to create and initialize a Ray cluster on AWS using Modin's Ray cluster setup config, which we are going to utilize in this example. Refer to Ray's autoscaler options page on how to modify the file.

More details on how to launch a Ray cluster can be found on Ray's cluster docs.

To start up the Ray cluster, run the following command in your terminal:

```
ray up modin-cluster.yaml
```

Once the head node has completed initialization, you can optionally connect to it by running the following command.

```
ray attach modin-cluster.yaml
```

To exit the ssh session and return back into your local shell session, type:

```
exit
```

2.2.3 Executing in a cluster environment

Note: Be careful when using the Ray client to connect to a remote cluster. We don't recommend this connection mode, beacuse it may not work. Known bugs: - https://github.com/ray-project/ray/issues/38713, - https://github.com/modin-project/modin/issues/6641.

Modin lets you instantly speed up your workflows with a large data by scaling pandas on a cluster. In this tutorial, we will use a 12.5 GB `big_yellow.csv` file that was created by concatenating a 200MB NYC Taxi dataset file 64 times. Preparing this file was provided as part of our Modin's Ray cluster setup config.

If you want to use the other dataset, you should provide it to each of the cluster nodes with the same path. We recomnend doing this by customizing the `setup_commands` section of the Modin's Ray cluster setup config.

To run any script in a remote cluster, you need to submit it to the Ray. In this way, the script file is sent to the the remote cluster head node and executed there.

In this tutorial, we provide the exercise_5.py script, which reads the data from the CSV file and executes such pandas operations as count, groupby and map. As the result, you will see the size of the file being read and the execution time of the entire script.

You can submit this script to the existing remote cluster by running the following command.

```
ray submit modin-cluster.yaml exercise_5.py
```

To download or upload files to the cluster head node, use `ray rsync_down` or `ray rsync_up`. It may help if you want to use some other Python modules that should be available to execute your own script or download a result file after executing the script.

```
# download a file from the cluster to the local machine:
ray rsync_down modin-cluster.yaml '/path/on/cluster' '/local/path'
# upload a file from the local machine to the cluster:
ray rsync_up modin-cluster.yaml '/local/path' '/path/on/cluster'
```

2.2.4 Shutting down the cluster

Now that we have finished the computation, we need to shut down the cluster with *ray down* command.

```
ray down modin-cluster.yaml
```

CHAPTER

THREE

WHY MODIN?

In this section, we explain the design and motivation behind Modin and why you should use Modin to scale up your pandas workflows. We first describe the architectural differences between pandas and Modin. Then we describe how Modin can also help resolve out-of-memory issues common to pandas. Finally, we look at the key differences between Modin and other distributed dataframe libraries.

3.1 How does Modin differ from pandas?

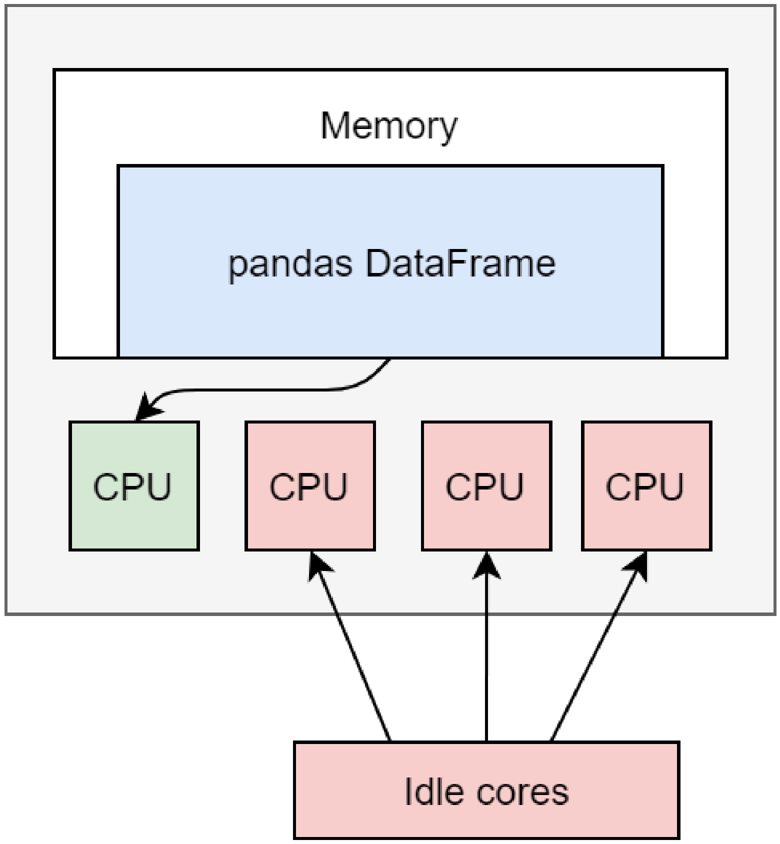
Note:

Estimated Reading Time: 10 minutes

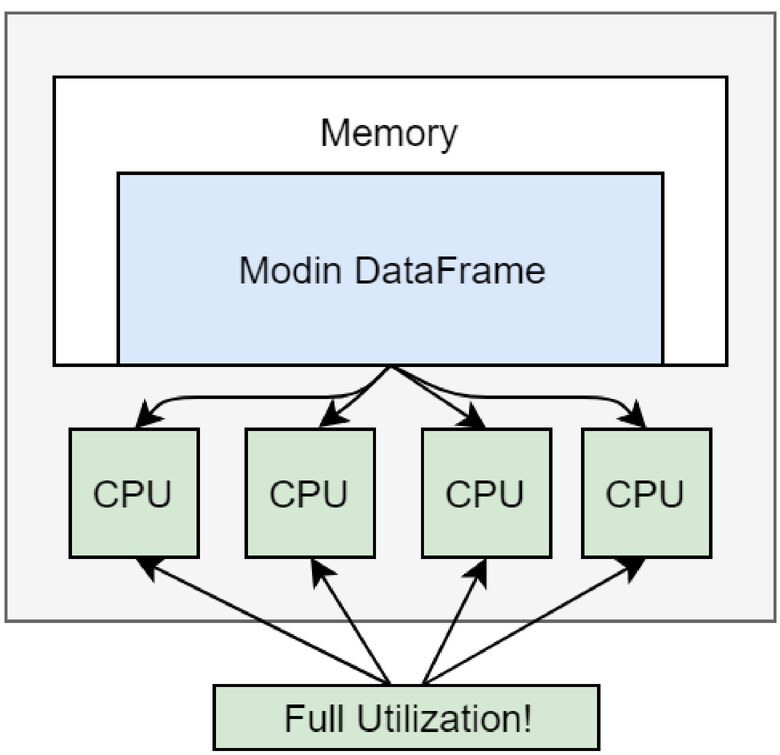
In the earlier tutorials, we have seen how Modin can be used to speed up pandas workflows. Here, we discuss at a high level how Modin works, in particular, how Modin's dataframe implementation differs from pandas.

3.1.1 Scalablity of implementation

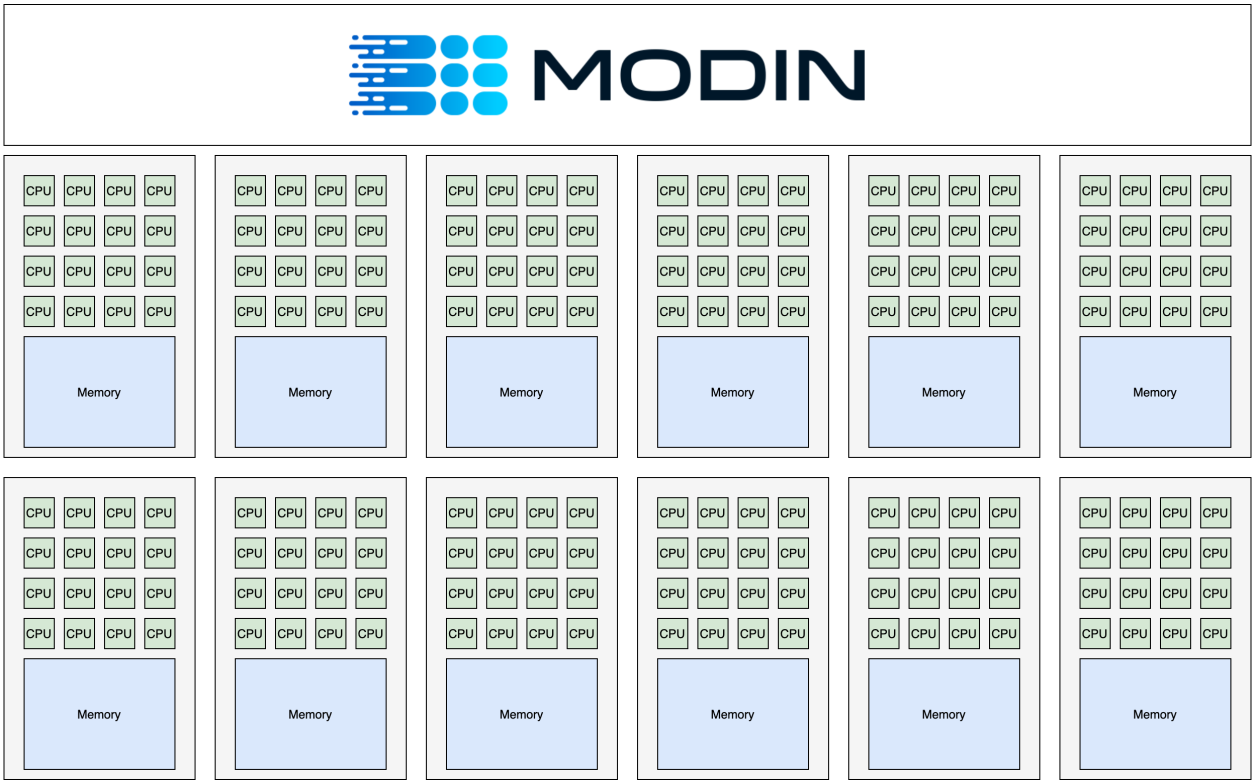
Modin exposes the pandas API through `modin.pandas`, but it does not inherit the same pitfalls and design decisions that make it difficult to scale. The pandas implementation is inherently single-threaded. This means that only one of your CPU cores can be utilized at any given time. In a laptop, it would look something like this with pandas:



However, Modin's implementation enables you to use all of the cores on your machine, or all of the cores in an entire cluster. On a laptop, it will look something like this:



The additional utilization leads to improved performance, however if you want to scale to an entire cluster, Modin suddenly looks something like this:



Modin is able to efficiently make use of all of the hardware available to it!

3.1.2 Memory usage and immutability

The pandas API contains many cases of "inplace" updates, which are known to be controversial. This is due in part to the way pandas manages memory: the user may think they are saving memory, but pandas is usually copying the data whether an operation was inplace or not.

Modin allows for inplace semantics, but the underlying data structures within Modin's implementation are immutable, unlike pandas. This immutability gives Modin the ability to internally chain operators and better manage memory layouts, because they will not be changed. This leads to improvements over pandas in memory usage in many common cases, due to the ability to share common memory blocks among all dataframes.

Modin provides the inplace semantics by having a mutable pointer to the immutable internal Modin dataframe. This pointer can change, but the underlying data cannot, so when an inplace update is triggered, Modin will treat it as if it were not inplace and just update the pointer to the resulting Modin dataframe.

3.1.3 API vs implementation

It is well known that the pandas API contains many duplicate ways of performing the same operation. Modin instead enforces that any one behavior have one and only one implementation internally. This guarantee enables Modin to focus on and optimize a smaller code footprint while still guaranteeing that it covers the entire pandas API. Modin has an internal algebra, which is roughly 15 operators, narrowed down from the original >200 that exist in pandas. The algebra is grounded in both practical and theoretical work. Learn more in our VLDB 2020 paper. More information about this algebra can be found in the *architecture* documentation.

3.2 Out-of-memory data with Modin

Note:

Estimated Reading Time: 10 minutes

When using pandas, you might run into a memory error if you are working with large datasets that cannot fit in memory or perform certain memory-intensive operations (e.g., joins).

Modin solves this problem by spilling over to disk, in other words, it uses your disk as an overflow for memory so that you can work with datasets that are too large to fit in memory. By default, Modin leverages out-of-core methods to handle datasets that don't fit in memory for both Ray and Dask engines.

Note: Object spilling is disabled in a multi-node Ray cluster by default. To enable object spilling use Ray instruction.

3.2.1 Motivating Example: Memory error with pandas

pandas makes use of in-memory data structures to store and operate on data, which means that if you have a dataset that is too large to fit in memory, it will cause an error on pandas. As an example, let's creates a 80GB DataFrame by appending together 40 different 2GB DataFrames.

```
import pandas
import numpy as np
df = pandas.concat([pandas.DataFrame(np.random.randint(0, 100, size=(2**20, 2**8))) for _
↪ in range(40)]) # Memory Error!
```

When we run this on a laptop with 32GB of RAM, pandas will run out of memory and throw an error (e.g., `MemoryError`, `Killed: 9`).

The pandas documentation has a great section on recommendations for scaling your analysis to these larger datasets. However, this generally involves loading in less data or rewriting your pandas code to process the data in smaller chunks.

3.2.2 Operating on out-of-memory data with Modin

In order to work with data that exceeds memory constraints, you can use Modin to handle these large datasets.

```
import modin.pandas as pd
import numpy as np
df = pd.concat([pd.DataFrame(np.random.randint(0, 100, size=(2**20, 2**8))) for _ in␣
↪range(40)]) # 40x2GB frames -- Working!
df.info()
```

Not only does Modin let you work with datasets that are too large to fit in memory, we can perform various operations on them without worrying about memory constraints.

3.2.3 Advanced: Configuring out-of-core settings

By default, out-of-core functionality is enabled by the compute engine selected. To disable it, start your preferred compute engine with the appropriate arguments. For example:

```
import modin.pandas as pd
import ray

ray.init(_plasma_directory="/tmp")  # setting to disable out of core in Ray
df = pd.read_csv("some.csv")
```

If you are using Dask, you have to modify local configuration files. Visit the Dask documentation on object spilling for more details.

3.3 Modin vs. Dask DataFrame vs. Koalas

Libraries such as Dask DataFrame (DaskDF for short) and Koalas aim to support the pandas API on top of distributed computing frameworks, Dask and Spark respectively. Instead, Modin aims to preserve the pandas API and behavior as is, while abstracting away the details of the distributed computing framework underneath. Thus, the aims of these libraries are fundamentally different.

Specifically, Modin enables pandas-like

- row and column-parallel operations, unlike DaskDF and Koalas that only support row-parallel operations
- indexing & ordering semantics, unlike DaskDF and Koalas that deviate from these semantics
- eager execution, unlike DaskDF and Koalas that provide lazy execution

As a result, Modin's coverage is more than 90% of the pandas API, while DaskDF and Koalas' coverage is about 55%.

For more technical details please see our VLDB 2022 research paper, referenced here.

3.3.1 Brief Overview of DaskDF and Koalas

Dask's DataFrame (DaskDF) is effectively a meta-DataFrame, partitioning and scheduling many smaller `pandas.DataFrame` objects. Users construct a task graph of dataframe computation step by step and then trigger computation using the `compute` function.

Spark's Koalas provides the pandas API on Spark, leveraging the preexisting Spark SQL optimizer to execute select pandas commands. Like DaskDF, Koalas also employs lazy computation, only triggering computation when the user requests to see the results.

3.3.2 Partitioning and Parallelization

Modin, DaskDF, Koalas are all examples of parallel dataframe systems. Parallelism is achieved by partitioning a large dataframe into smaller ones that can be operated on in parallel. As a result, the partitioning scheme chosen by the system dictates the pandas functions that can or can not be supported.

DaskDF and Koalas only support row-oriented partitioning and parallelism. This approach is analogous to relational databases. The dataframe is conceptually broken down into horizontal partitions along rows, where each partition is independently processed if possible. When DaskDF or Koalas are required to perform column-parallel operations that to be done on columns independently (e.g., dropping columns with null values via `dropna` on the column `axis`), they either perform very poorly with no parallelism or do not support that operation.

Modin supports both row, column, and cell-oriented partitioning and parallelism. That is, the dataframe can be conceptually broken down as groups of rows, groups of columns, or both groups of rows and groups of columns (effectively a block or sub-matrix). Modin will transparently reshape the partitioning as necessary for the corresponding operation, based on whether the operation is row-parallel, column-parallel, or cell-parallel (independently applied to each unit cell). This allows Modin to support more of the pandas API and do so efficiently. Due to the finer-grained control over the partitioning, Modin can support a number of operations that are very challenging to parallelize in row-oriented systems (e.g., `transpose`, `median`, `quantile`). This flexibility in partitioning also gives Modin tremendous power to implement efficient straggler mitigation and improve utilization over the entire cluster.

3.3.3 API Coverage

One of the key benefits of pandas is its versatility, due to the wide array of operations, with more than 600+ API operations for data cleaning, feature engineering, data transformation, data summarization, data exploration, and machine learning. However, it is not trivial to develop scalable implementations of each of these operations in a dataframe system. **DaskDF and Koalas only implements about 55% of the pandas API**; they do not implement certain APIs that would deviate from the row-wise partitioning approach, or would be inefficient with the row-wise parallelization. For example, Dask does not implement `iloc`, `MultiIndex`, `apply(axis=0)`, `quantile` (only approximate quantile is available), `median`, and more. Given DaskDF's row-oriented architecture, `iloc`, for example, can technically be implemented, but it would be inefficient, and column-wise operations such as `apply(axis=0)` would be impossible to implement. Similarly, Koalas does not implement `apply(axis=0)` (it only applies the function per row partition, giving a different result), `quantile`, `median` (only approximate quantile/median is available), `MultiIndex`, `combine`, `compare` and more.

Modin supports all of the above pandas API functions, as well as others, with more than 90% coverage of the pandas API. Modin additionally acts as a drop-in replacement for pandas, such that even if the API is not yet supported, it still works by falling back to running vanilla pandas. One of the key features of being a drop-in replacement is that not only will it work for existing code, if a user wishes to go back to running pandas directly, they are not locked in to using Modin and can switch between Modin and pandas at no cost. In other words, scripts and notebooks written in Modin can be converted to and from pandas as the user desires by simply replacing the import statement.

3.3.4 Execution Semantics

DaskDF and Koalas make use of lazy evaluation, which means that the computation is delayed until users explicitly evaluate the results. This mode of evaluation places a lot of optimization responsibility on the user, forcing them to think about when it would be useful to inspect the intermediate results or delay doing so. Specifically, DaskDF's API differs from pandas in that it requires users to explicitly call `.compute()` to materialize the result of the computation. Often if that computation corresponds to a long chain of operators, this call can take a very long time to execute. Overall, the need to explicitly trigger computation makes the API less convenient to work with, but gives DaskDF and Koalas the opportunity to perform holistic optimizations over the entire dataflow graph. However, to the best of our knowledge, neither DaskDF nor Koalas actually leverage holistic optimizations.

Modin employs eager evaluation, like pandas. Eager evaluation is the default mode of operation for data scientists when working with pandas in an interactive environment, such as Jupyter Notebooks. Modin reproduces this familiar behavior by performing all computations eagerly as soon as it is issued, so that users can inspect intermediate results and quickly see the results of their computations without having to wait or explicitly trigger computation. This is especially useful during interactive data analysis, where users often iterate on their dataframe workflows or build up their dataframe queries in an incremental fashion. We also have developed techniques for opportunistic evaluation that bridges the gap between lazy and eager evaluation that will be incorporated in Modin in the future.

3.3.5 Ordering Semantics

By default, pandas preserves the order of the dataframe, so that users can expect a consistent, ordered view as they are operating on their dataframe.

Both DaskDF and Koalas make no guarantees about the order of rows in the DataFrame. This is because DaskDF sorts the `index` for optimization purposes to speed up computations that involve the row index; and as a result, it does not support user-specified order. Likewise, Koalas does not support ordering by default because it will lead to a performance overhead when operating on distributed datasets.

DaskDF additionally does not support multi-indexing or sorting. DaskDF sorts the data based on a single set of row labels for fast row lookups, and builds an indexing structure based on these row labels. Data is both logically and physically stored in the same order. As a result, DaskDF does not support a *sort* function.

Modin reproduces the intuitive behavior in pandas where the order of the DataFrame is preserved, and supports multi-indexing. Enforcing ordering on a parallel dataframe system like Modin requires non-trivial effort that involves decoupling of the logical and physical representation of the data, enabling the order to be lazily kept up-to-date, but eagerly computed based on user needs (See Section 4.2 in our recent paper). Modin abstracts away the physical representation of the data and provides an ordered view that is consistent with user's expectations.

3.3.6 Compatibility with Computational Frameworks

DaskDF and Koalas are meant to be run on Dask and Spark respectively. They are highly tuned to the corresponding frameworks, and cannot be ported to other computational frameworks.

Modin's highly modular design is architected to run on a variety of systems, and support a variety of APIs. The goal for the extensible design is that users can take the same notebook or script and seamlessly move between different clusters and environments, with Modin being able to support the pandas API on your preexisting infrastructure. Currently, Modin support running on Dask's compute engine in addition to Ray. The modular design makes it easier for developers to different execution engines or compile to different memory formats. Modin can run on a Dask cluster in the same way that DaskDF can, but they differ in the ways described above. In addition, Modin is continually expanding to support popular data processing APIs (SQL in addition to pandas, among other DSLs for data processing) while leveraging the same underlying execution framework. Modin's flexible architecture also means that as the pandas API continues to evolve, Modin can quickly move towards supporting new versions of the pandas API.

3.3.7 Performance Comparison

On operations supported by all systems, Modin provides substantial speedups. Thanks to its optimized design, Modin is able to take advantage of multiple cores relative to both Koalas and DaskDF to efficiently execute pandas operations. It is notable that Koalas is often slower than pandas, due to the overhead of Spark.

Modin provides substantial speedups even on operators not supported by other systems. Thanks to its flexible partitioning schemes that enable it to support the vast majority of pandas operations — be it row, column, or cell-oriented - Modin provides benefits on operations such as `join`, `median`, and `infer_types`. While Koalas performs `join` slower than Pandas, Dask failed to support `join` on more than 20M rows, likely due poor support for shuffles. Details of the benchmark and additional join experiments can be found in our paper.

Modin is built on many years of research and development at UC Berkeley. For more information on how this works underneath the hoods, check out our publications in this space:

- Flexible Rule-Based Decomposition and Metadata Independence in Modin (VLDB 2021)
- Enhancing the Interactivity of Dataframe Queries by Leveraging Think Time (IEEE Data Eng 2021)
- Dataframe Systems: Theory, Architecture, and Implementation (PhD Dissertation 2021)
- Scaling Data Science does not mean Scaling Machines (CIDR 2021)
- Towards Scalable Dataframe Systems (VLDB 2020)

CHAPTER

FOUR

EXAMPLES AND RESOURCES

Here you can find additional resources to learn about Modin. To learn more about advanced usage for Modin, please refer to *Usage Guide* section..

4.1 Usage Examples

The following notebooks demonstrate how Modin can be used for scalable data science:

- Quickstart Guide to Modin [Source]
- Using Modin with the NYC Taxi Dataset [Source]
- Modin for Machine Learning with scikit-learn [Source]

4.2 Tutorials

The following tutorials cover the basic usage of Modin. Here is a one hour video tutorial that walks through these basic exercises.

- Exercise 1: Introduction to Modin [Source PandasOnRay, Source PandasOnDask]
- Exercise 2: Speed Improvements with Modin [Source PandasOnRay, Source PandasOnDask]
- Exercise 3: Defaulting to pandas with Modin [Source PandasOnRay, Source PandasOnDask]

The following tutorials covers more advanced features in Modin:

- Exercise 4: Experimental Features in Modin (Spreadsheet, Progress Bar) [Source PandasOnRay, Source PandasOnDask]
- Exercise 5: Setting up Modin in a Cluster Environment [Source PandasOnRay]
- Exercise 6: Running Modin in a Cluster Environment [Source PandasOnRay]

How to get required dependencies for the tutorial notebooks and to run them please refer to the respective README.md file.

4.3 Talks & Podcasts

- Scaling Interactive Data Science with Modin and Ray (20 minute, Ray Summit 2021)
- Unleash The Power Of Dataframes At Any Scale With Modin (40 minute, Python Podcast 2021)
- [Russian] Distributed Data Processing and XGBoost Training and Prediction with Modin (30 minute, PyCon Russia 2021)
- [Russian] Efficient Data Science with Modin (30 minute, ISP RAS Open 2021)
- Modin: Scaling the Capabilities of the Data Scientist, not the Machine (1 hour, RISE Camp 2020)
- Modin: Pandas Scalability with Devin Petersohn (1 hour, Software Engineering Daily Podcast 2020)
- Introduction to the DataFrame and Modin (20 minute, RISECamp 2019)
- Scaling Interactive Pandas Workflows with Modin (40 minute, PyData NYC 2018)

4.4 Community contributions

Here are some blogposts and articles about Modin:

- Anaconda Blog: Scale your pandas workflow with Modin by Vasilij Litvinov
- The Modin view of Scaling Pandas by Devin Petersohn
- Data Science at Scale with Modin by Areg Melik-Adamyan
- Speed up Pandas using Modin by Eric D. Brown, D.Sc.
- Explore Python Libraries: Make Your DataFrames Parallel With Modin by Zachary Bennett
- Get faster pandas with Modin, even on your laptops by Parul Pandey
- How to speedup pandas by changing one line of code by Shrivarsheni
- How To Accelerate Pandas With Just One Line Of Code by Analytics India
- An Easy Introduction to Modin: A Step-by-Step Guide to Accelerating Pandas by Intel

Here are some articles contributed by the international community:

- [Chinese] Modin pandas by Python Chinese Community
- [German] Was ist Modin? by Dipl.-Ing. (FH) Stefan Luber
- [Russian] Pandas modin by
- [Korean] modin pandas by

If you would like your articles to be featured here, please submit a pull request to let us know!

CHAPTER

FIVE

FREQUENTLY ASKED QUESTIONS (FAQS)

Below, you will find answers to the most commonly asked questions about Modin. If you still cannot find the answer you are looking for, please post your question on the #support channel on our Slack community or open a Github issue.

5.1 FAQs: Why choose Modin?

5.1.1 What's wrong with pandas and why should I use Modin?

While pandas works extremely well on small datasets, as soon as you start working with medium to large datasets that are more than a few GBs, pandas can become painfully slow or run out of memory. This is because pandas is single-threaded. In other words, you can only process your data with one core at a time. This approach does not scale to larger data sets and adding more hardware does not lead to more performance gain.

The `DataFrame` is a highly scalable, parallel DataFrame. Modin transparently distributes the data and computation so that you can continue using the same pandas API while being able to work with more data faster. Modin lets you use all the CPU cores on your machine, and because it is lightweight, it often has less memory overhead than pandas. See :doc:` Why Modin? </getting_started/why_modin/pandas>` page to learn more about how Modin is different from pandas.

5.1.2 Why not just improve pandas?

pandas is a massive community and well established codebase. Many of the issues we have identified and resolved with pandas are fundamental to its current implementation. While we would be happy to donate parts of Modin that make sense in pandas, many of these components would require significant (or total) redesign of the pandas architecture. Modin's architecture goes beyond pandas, which is why the pandas API is just a thin layer at the user level. To learn more about Modin's architecture, see the *architecture* documentation.

5.1.3 How much faster can I go with Modin compared to pandas?

Modin is designed to scale with the amount of hardware available. Even in a traditionally serial task like `read_csv`, we see large gains by efficiently distributing the work across your entire machine. Because it is so light-weight, Modin provides speed-ups of up to 4x on a laptop with 4 physical cores. This speedup scales efficiently to larger machines with more cores. We have several published papers that include performance results and comparisons against pandas.

5.1.4 How much more data would I be able to process with Modin?

Often data scientists have to use different tools for operating on datasets of different sizes. This is not only because processing large dataframes is slow, but also pandas does not support working with dataframes that don't fit into the available memory. As a result, pandas workflows that work well for prototyping on a few MBs of data do not scale to tens or hundreds of GBs (depending on the size of your machine). Modin supports operating on data that does not fit in memory, so that you can comfortably work with hundreds of GBs without worrying about substantial slowdown or memory errors. For more information, see *out-of-memory support* for Modin.

5.1.5 How does Modin compare to Dask DataFrame and Koalas?

TLDR: Modin has better coverage of the pandas API, has a flexible backend, better ordering semantics, and supports both row and column-parallel operations. Check out *Modin vs Dask vs Koalas* page detailing the differences!

5.1.6 How does Modin work under the hood?

Modin is logically separated into different layers that represent the hierarchy of a typical Database Management System. User queries which perform data transformation, data ingress or data egress pass through the Modin Query Compiler which translates queries from the top-level pandas API Layer that users interact with to the Modin Core Dataframe layer. The Modin Core DataFrame is our efficient DataFrame implementation that utilizes a partitioning schema which allows for distributing tasks and queries. From here, the Modin DataFrame works with engines like Ray, Dask or Unidist to execute computation, and then return the results to the user.

For more details, take a look at our system *architecture*.

5.2 FAQs: How to use Modin?

5.2.1 If I'm only using my laptop, can I still get the benefits of Modin?

Absolutely! Unlike other parallel DataFrame systems, Modin is an extremely light-weight, robust DataFrame. Because it is so light-weight, Modin provides speed-ups of up to 4x on a laptop with 4 physical cores and allows you to work on data that doesn't fit in your laptop's RAM.

5.2.2 How do I use Jupyter or Colab notebooks with Modin?

You can take a look at this Google Colab installation guide and this notebook tutorial. Once Modin is installed, simply replace your pandas import with Modin import:

```python
# import pandas as pd
import modin.pandas as pd
```

5.2.3 Which execution engine (Ray, Dask or Unidist) should I use for Modin?

Modin lets you effortlessly speed up your pandas workflows with either Ray's, Dask's or Unidist's execution engine. You don't need to know anything about either engine in order to use it with Modin. If you only have one engine installed, Modin will automatically detect which engine you have installed and use that for scheduling computation. If you don't have a preference, we recommend starting with Modin's default Ray engine. If you want to use a specific compute engine, you can set the environment variable MODIN_ENGINE and Modin will do computation with that engine:

```
pip install "modin[ray]" # Install Modin dependencies and Ray to run on Ray
export MODIN_ENGINE=ray  # Modin will use Ray

pip install "modin[dask]" # Install Modin dependencies and Dask to run on Dask
export MODIN_ENGINE=dask  # Modin will use Dask

pip install "modin[mpi]" # Install Modin dependencies and MPI to run on MPI through␣
↪unidist.
export MODIN_ENGINE=unidist  # Modin will use Unidist
export UNIDIST_BACKEND=mpi   # Unidist will use MPI backend.
```

This can also be done with:

```
import modin.config as modin_cfg
import unidist.config as unidist_cfg

modin_cfg.Engine.put("ray")  # Modin will use Ray
modin_cfg.Engine.put("dask")  # Modin will use Dask

modin_cfg.Engine.put('unidist') # Modin will use Unidist
unidist_cfg.Backend.put('mpi') # Unidist will use MPI backend
```

We plan to support more execution engines in future. If you have a specific request, please post on the #feature-requests channel on our Slack community.

5.2.4 How do I connect Modin to a database via *read_sql*?

To read from a SQL database, you have two options:

1) Pass a connection string, e.g. `postgresql://reader:NWDMCE5xdipIjRrp@hh-pgsql-public.ebi.ac.uk:5432/pfmegrnargs`
2) Pass an open database connection, e.g. for psycopg2, `psycopg2.connect("dbname=pfmegrnargs user=reader password=NWDMCE5xdipIjRrp host=hh-pgsql-public.ebi.ac.uk")`

The first option works with both Modin and pandas. If you try the second option in Modin, Modin will default to pandas because open database connections cannot be pickled. Pickling is required to send connection details to remote workers. To handle the unique requirements of distributed database access, Modin has a distributed database connection called ModinDatabaseConnection:

```
import modin.pandas as pd
from modin.db_conn import ModinDatabaseConnection
con = ModinDatabaseConnection(
    'psycopg2',
    host='hh-pgsql-public.ebi.ac.uk',
    dbname='pfmegrnargs',
```

(continues on next page)

(continued from previous page)

```
    user='reader',
    password='NWDMCE5xdipIjRrp')
df = pd.read_sql("SELECT * FROM rnc_database",
        con,
        index_col=None,
        coerce_float=True,
        params=None,
        parse_dates=None,
        chunksize=None)
```

The `ModinDatabaseConnection` will save any arguments you supply it and forward them to the workers to make their own connections.

5.2.5 How can I contribute to Modin?

Modin is currently under active development. Requests and contributions are welcome!

If you are interested in contributing please check out the *Contributing Guide* and then refer to the *Development Documentation*, where you can find system architecture, internal implementation details, and other useful information. Also check out the Github to view open issues and make contributions.

CHAPTER

SIX

TROUBLESHOOTING

We hope your experience with Modin is bug-free, but there are some quirks about Modin that may require troubleshooting. If you are still having issues, please post on the #support channel on our Slack community or open a Github issue.

6.1 Frequently encountered issues

This is a list of the most frequently encountered issues when using Modin. Some of these are working as intended, while others are known bugs that are being actively worked on.

6.1.1 Warning during execution: `defaulting to pandas`

Please note, that while Modin covers a large portion of the pandas API, not all functionality is implemented. For methods that are not yet implemented, such as `asfreq`, you may see the following:

```
UserWarning: `DataFrame.asfreq` defaulting to pandas implementation.
```

To understand which functions will lead to this warning, we have compiled a list of *currently supported methods*. When you see this warning, Modin defaults to pandas by converting the Modin dataframe to pandas to perform the operation. Once the operation is complete in pandas, it is converted back to a Modin dataframe. These operations will have a high overhead due to the communication involved and will take longer than pandas. When this is happening, a warning will be given to the user to inform them that this operation will take longer than usual. You can learn more about this *here*.

If you would like to request a particular method be implemented, feel free to open an issue. Before you open an issue please make sure that someone else has not already requested that functionality.

6.1.2 Hanging on `import modin.pandas as pd`

This can happen when Ray fails to start. It will keep retrying, but often it is faster to just restart the notebook or interpreter. Generally, this should not happen. Most commonly this is encountered when starting multiple notebooks or interpreters in quick succession.

Solution

Restart your interpreter or notebook kernel.

Avoiding this Error

Avoid starting many Modin notebooks or interpreters in quick succession. Wait 2-3 seconds before starting the next one.

6.1.3 Importing heterogeneous data using read_csv

Since Modin's `read_csv` imports data in parallel, it is possible for data across partitions to be heterogeneously typed (this can happen when columns contain heterogeneous data, i.e. values in the same column are of different types). An example of how this is handled is shown below.

```
import os
import pandas
import modin.pandas as pd
from modin.config import NPartitions

NPartitions.put(2)

test_filename = "test.csv"
# data with heterogeneous values in the first column
data = """one,2
3,4
5,6
7,8
9.0,10
"""
kwargs = {
    # names of the columns to set, if `names` parameter is set,
    # header inffering from the first data row/rows will be disabled
    "names": ["col1", "col2"],

    # explicit setting of data type of column/columns with heterogeneous
    # data will force partitions to read data with correct dtype
    # "dtype": {"col1": str},
}


try :
    with open(test_filename, "w") as f:
        f.write(data)

    pandas_df = pandas.read_csv(test_filename, **kwargs)
    pd_df = pd.read_csv(test_filename, **kwargs)

    print(pandas_df)
    print(pd_df)
finally:
    os.remove(test_filename)

Output:

pandas_df:
  col1  col2
0  one     2
1    3     4
2    5     6
3    7     8
4  9.0    10
```

(continues on next page)

(continued from previous page)

```
pd_df:
  col1  col2
0  one     2
1    3     4
2    5     6
3  7.0     8
4  9.0    10
```

In this case, `col1` of the *DataFrame* read by pandas contains only `str` data because the first value ("one") is inferred to have type `str`, which forces pandas to handle the rest of the values in the column as strings. The first Modin partition (the first three rows) handles the data as pandas does, but the second partition (the last two rows) reads the data as floats. This is because the second column contains an int and a float, and thus the column type is inferred to be float. As a result, *7* is interpreted as *7.0*, which differs from the pandas output.

The above example demonstrates heterogenous data import with str, int, and float types, but heterogeneous data consisting of other data/parameter combinations can also result in data type mismatches with pandas.

Solution

When heterogeneous data is detected, a warning will be raised. Currently, these discrepancies aren't properly handled by Modin, so to avoid this issue, you need to set the `dtype` parameter of `read_csv` manually to force the correct data type coercion during data import. Note that to avoid excessive performance degradation, the `dtype` value should only be set for columns that may contain heterogenous data. as possible (specify `dtype` parameter only for columns with heterogeneous data).

Specifying the `dtype` parameter will work well in most cases. If the file contains a column that should be interpreted as the index (the `index_col` parameter is specified) there may still be type discrepancies in the index, since the `dtype` parameter is only responsible for data fields. If in the above example, `kwargs` was set like so:

```
kwargs = {
    "names": ["col1", "col2"],
    "dtype": {"col1": str},
    "index_col": "col1",
}
```

The resulting Modin DataFrame will contain incorrect values - just as if `dtype` had not been specified:

```
col1
one      2
3        4
5        6
7.0      8
9.0     10
```

One workaround is to import the data without setting the `index_col` parameter, and then set the index column using the `DataFrame.set_index` function as shown in the example below:

```
pd_df = pd.read_csv(filename, dtype=data_dtype, index_col=None)
pd_df = pd_df.set_index(index_col_name)
pd_df.index.name = None
```

6.1.4 Using Modin with python multiprocessing

We strongly recommend against using a distributed execution engine (e.g. Ray or Dask) in conjunction with Python multiprocessing because that can lead to undefined behavior. One such example is shown below:

```
import modin.pandas as pd

# Ray engine is used by default
df = pandas.DataFrame([1, 2, 3])

def f(arg):
  return df + arg

if __name__ == '__main__':
  from multiprocessing import Pool

  with Pool(5) as p:
      print(p.map(f, [1]))
```

Although this example may work on your machine, we do not recommend it, because the Python multiprocessing library will duplicate Ray clusters, causing both excessive resource usage and conflict over the available resources.

6.1.5 Poor performance of the first operation with Modin on Ray engine

There might be cases when the first operation with Modin on Ray engine is much slower than the subsequent calls of the operation. That happens because Ray workers may not be fully set up yet to perform computation after initialization of the engine with `ray.init(runtime_env={'env_vars': {'__MODIN_AUTOIMPORT_PANDAS__': '1'}})`, which is the default behavior of Modin on Ray engine if Ray has not been initialised yet. Modin intentionaly initializes Ray this way to import `pandas` in workers once Python interpreter is started in them so that to avoid a race condition in Ray between the import thread and the thread executing the code.

```
import time
import pandas
import numpy as np
import ray
import modin.pandas as pd
import modin.config as cfg

# Look at the Ray documentation with respect to the Ray configuration suited to you most.
ray.init(runtime_env={'env_vars': {'__MODIN_AUTOIMPORT_PANDAS__': '1'}})

pandas_df = pandas.DataFrame(
  np.random.randint(0, 100, size=(1000000, 13))
)
pandas_df.to_csv("foo.csv", index=False)

def read_csv_with_pandas():
  start_time = time.time()
  pandas_df = pandas.read_csv("foo.csv", index_col=0)
  end_time = time.time()
  pandas_duration = end_time - start_time
  print("Time to read_csv with pandas: {} seconds".format(round(pandas_duration, 3)))
  return pandas_df
```

(continues on next page)

(continued from previous page)

```
def read_csv_with_modin():
  start_time = time.time()
  modin_df = pd.read_csv("foo.csv", index_col=0)
  end_time = time.time()
  modin_duration = end_time - start_time
  print("Time to read_csv with Modin: {} seconds".format(round(modin_duration, 3)))
  return modin_df

for i in range(5):
  pandas_df = read_csv_with_pandas()
  modin_df = read_csv_with_modin()

Time to read_csv with pandas: 0.708 seconds
Time to read_csv with Modin: 4.132 seconds
Time to read_csv with pandas: 0.735 seconds
Time to read_csv with Modin: 0.37 seconds
Time to read_csv with pandas: 0.646 seconds
Time to read_csv with Modin: 0.377 seconds
Time to read_csv with pandas: 0.673 seconds
Time to read_csv with Modin: 0.371 seconds
Time to read_csv with pandas: 0.672 seconds
Time to read_csv with Modin: 0.379 seconds
```

Solution

So far there is no a solution to fix or work around the problem rather than not to pass a non-empty runtime_env to `ray.init()`. However, this may lead to other problem regarding a race condition in Ray between the import thread and the thread executing the code. So for now we just highlight the problem in hope of a future fix in Ray itself.

Also, it is worth noting that every distributed engine by its nature has a little overhead for the first operation being called, which may be important for microbenchmarks. What you likely want to do is warm up worker processes either by excluding the time of the first iteration from your measurements or execute a simple function in workers to fully set up them.

6.2 Common errors

6.2.1 Error when using Dask engine: `RuntimeError: if __name__ == '__main__':`

The following *script.py* uses Modin with Dask as an execution engine and produces errors:

```
# script.py
import modin.pandas as pd
import modin.config as cfg

cfg.Engine.put("dask")

df = pd.DataFrame([0,1,2,3])
print(df)
```

A part of the produced errors by the script above would be the following:

```
File "/path/python3.9/multiprocessing/spawn.py", line 134, in _check_not_importing_main
  raise RuntimeError('''
  RuntimeError:
      An attempt has been made to start a new process before the
      current process has finished its bootstrapping phase.

      This probably means that you are not using fork to start your
      child processes and you have forgotten to use the proper idiom
      in the main module:

          if __name__ == '__main__':
              freeze_support()
              ...

      The "freeze_support()" line can be omitted if the program
      is not going to be frozen to produce an executable.
```

This happens because Dask Client uses fork to start processes.

Solution

To avoid the problem the Dask Client creation code needs to be moved into the `__main__` scope of the module.

The corrected *script.py* would look like:

```python
# script.py
import modin.pandas as pd
import modin.config as cfg

cfg.Engine.put("dask")

if __name__ == "__main__":
  df = pd.DataFrame([0, 1, 2, 3]) # Dask Client creation is hidden in the first call of␣
↪Modin functionality.
  print(df)
```

or

```python
# script.py
from distributed import Client
import modin.pandas as pd
import modin.config as cfg

cfg.Engine.put("dask")

if __name__ == "__main__":
  # Explicit Dask Client creation.
  # Look at the Dask Distributed documentation with respect to the Client configuration␣
↪suited to you most.
  client = Client()
  df = pd.DataFrame([0, 1, 2, 3])
  print(df)
```

6.2.2 Spurious error "cannot import partially initialised pandas module" on custom Ray cluster

If you're using some pre-configured Ray cluster to run Modin, it's possible you would be seeing spurious errors like

```
ray.exceptions.RaySystemError: System error: partially initialized module 'pandas' has␣
↪no attribute 'core' (most likely due to a circular import)
traceback: Traceback (most recent call last):
  File "/usr/share/miniconda/envs/modin/lib/python3.8/site-packages/ray/serialization.py
↪", line 340, in deserialize_objects
    obj = self._deserialize_object(data, metadata, object_ref)
  File "/usr/share/miniconda/envs/modin/lib/python3.8/site-packages/ray/serialization.py
↪", line 237, in _deserialize_object
    return self._deserialize_msgpack_data(data, metadata_fields)
  File "/usr/share/miniconda/envs/modin/lib/python3.8/site-packages/ray/serialization.py
↪", line 192, in _deserialize_msgpack_data
    python_objects = self._deserialize_pickle5_data(pickle5_data)
  File "/usr/share/miniconda/envs/modin/lib/python3.8/site-packages/ray/serialization.py
↪", line 180, in _deserialize_pickle5_data
    obj = pickle.loads(in_band, buffers=buffers)
  File "/usr/share/miniconda/envs/modin/lib/python3.8/site-packages/pandas/__init__.py",␣
↪line 135, in <module>
    from pandas import api, arrays, errors, io, plotting, testing, tseries
  File "/usr/share/miniconda/envs/modin/lib/python3.8/site-packages/pandas/testing.py",␣
↪line 6, in <module>
    from pandas._testing import (
  File "/usr/share/miniconda/envs/modin/lib/python3.8/site-packages/pandas/_testing/__
↪init__.py", line 979, in <module>
    cython_table = pd.core.common._cython_table.items()
AttributeError: partially initialized module 'pandas' has no attribute 'core' (most␣
↪likely due to a circular import)
```

Solution

Modin contains a workaround that should automatically do `import pandas` upon worker process starts.

It is triggered by the presence of non-empty `__MODIN_AUTOIMPORT_PANDAS__` environment variable which Modin sets up automatically on the Ray clusters it spawns, but it might be missing on pre-configured clusters.

So if you're seeing the issue like shown above, please make sure you set this environment variable on all worker nodes of your cluster before actually spawning the workers.

CHAPTER

SEVEN

QUICK START GUIDE

To install the most recent stable release for Modin run the following:

```
pip install "modin[all]"
```

For further instructions on how to install Modin with conda or for specific platforms or engines, see our detailed installation guide.

Modin acts as a drop-in replacement for pandas so you simply have to replace the import of pandas with the import of Modin as follows to speed up your pandas workflows:

```
# import pandas as pd
import modin.pandas as pd
```

CHAPTER

EIGHT

EXAMPLE: INSTANT SCALABILITY WITH NO EXTRA EFFORT

When working on large datasets, pandas becomes painfully slow or *runs out of memory*. Modin automatically scales up your pandas workflows by parallelizing the dataframe operations, so that you can more effectively leverage the compute resources available.

For the purpose of demonstration, we will load in modin as `pd` and pandas as `pandas`.

```
import modin.pandas as pd
import pandas

#############################################
### For the purpose of timing comparisons ###
#############################################
import time
import ray
# Look at the Ray documentation with respect to the Ray configuration suited to you most.
ray.init()
#############################################
```

In this toy example, we look at the NYC taxi dataset, which is around 200MB in size. You can download this dataset to run the example locally.

```
# This may take a few minutes to download
import urllib.request
dataset_url = "https://modin-datasets.intel.com/testing/yellow_tripdata_2015-01.csv"
urllib.request.urlretrieve(dataset_url, "taxi.csv")
```

8.1 Faster Data Loading with `read_csv`

```
start = time.time()

pandas_df = pandas.read_csv(dataset_url, parse_dates=["tpep_pickup_datetime", "tpep_
↪dropoff_datetime"], quoting=3)

end = time.time()
pandas_duration = end - start
print("Time to read with pandas: {} seconds".format(round(pandas_duration, 3)))
```

By running the same command `read_csv` with Modin, we generally get around 4X speedup for loading in the data in parallel.

```
start = time.time()

modin_df = pd.read_csv(dataset_url, parse_dates=["tpep_pickup_datetime", "tpep_dropoff_
→datetime"], quoting=3)

end = time.time()
modin_duration = end - start
print("Time to read with Modin: {} seconds".format(round(modin_duration, 3)))

print("Modin is {}x faster than pandas at `read_csv`!".format(round(pandas_duration /␣
→modin_duration, 2)))
```

8.2 Faster concat across multiple dataframes

Our previous read_csv example operated on a relatively small dataframe. In the following example, we duplicate the same taxi dataset 100 times and then concatenate them together, resulting in a dataset around 19GB in size.

```
start = time.time()

big_pandas_df = pandas.concat([pandas_df for _ in range(25)])

end = time.time()
pandas_duration = end - start
print("Time to concat with pandas: {} seconds".format(round(pandas_duration, 3)))
```

```
start = time.time()

big_modin_df = pd.concat([modin_df for _ in range(25)])

end = time.time()
modin_duration = end - start
print("Time to concat with Modin: {} seconds".format(round(modin_duration, 3)))

print("Modin is {}x faster than pandas at `concat`!".format(round(pandas_duration /␣
→modin_duration, 2)))
```

Modin speeds up the concat operation by more than 60X, taking less than a second to create the large dataframe, while pandas took close to a minute.

8.3 Faster apply over a single column

The performance benefits of Modin become apparent when we operate on large gigabyte-scale datasets. Let's say we want to round up values across a single column via the apply operation.

```
start = time.time()
rounded_trip_distance_pandas = big_pandas_df["trip_distance"].apply(round)

end = time.time()
```

(continues on next page)

(continued from previous page)

```
pandas_duration = end - start
print("Time to apply with pandas: {} seconds".format(round(pandas_duration, 3)))
```

```
start = time.time()

rounded_trip_distance_modin = big_modin_df["trip_distance"].apply(round)

end = time.time()
modin_duration = end - start
print("Time to apply with Modin: {} seconds".format(round(modin_duration, 3)))

print("Modin is {}x faster than pandas at `apply` on one column!".format(round(pandas_
↪duration / modin_duration, 2)))
```

Modin is more than 30X faster at applying a single column of data, operating on 130+ million rows in a second.

In short, Modin provides orders of magnitude speed up over pandas for a variety of operations out of the box.

CHAPTER

NINE

SUMMARY

Hopefully, this tutorial demonstrated how Modin delivers significant speedup on pandas operations without the need for any extra effort. Throughout example, we moved from working with 100MBs of data to 20GBs of data all without having to change anything or manually optimize our code to achieve the level of scalable performance that Modin provides.

Note that in this quickstart example, we've only shown `read_csv`, `concat`, `apply`, but these are not the only pandas operations that Modin optimizes for. In fact, Modin covers more than 90% of the pandas API, yielding considerable speedups for many common operations.

CHAPTER

TEN

USAGE GUIDE

This guide describes both basic and advanced Modin usage, including usage examples, details regarding Modin configuration settings, as well as tips and tricks on how you can further optimize the performance of your workload with Modin.

10.1 Modin Configuration Settings

To adjust Modin's default behavior, you can set the value of Modin configs by setting an environment variable or by using the `modin.config` API. To list all available configs in Modin, please run `python -m modin.config` to print all Modin configs with descriptions.

10.1.1 Public API

Potentially, the source of configs can be any, but for now only environment variables are implemented. Any environment variable originate from *EnvironmentVariable*, which contains most of the config API implementation.

class modin.config.envvars.**EnvironmentVariable**

Base class for environment variables-based configuration.

classmethod get() → Any

Get config value.

Returns

Decoded and verified config value.

Return type

Any

classmethod get_help() → str

Generate user-presentable help for the config.

Return type

str

classmethod get_value_source() → ValueSource

Get value source of the config.

Return type

ValueSource

classmethod once(*onvalue: Any*, *callback: Callable*) → None

Execute *callback* if config value matches *onvalue* value.

Otherwise accumulate callbacks associated with the given *onvalue* in the *_once* container.

Parameters

- **onvalue** (*Any*) – Config value to set.
- **callback** (*callable*) – Callable that should be executed if config value matches *onvalue*.

classmethod put(*value: Any*) → None

Set config value.

Parameters

value (*Any*) – Config value to set.

classmethod subscribe(*callback: Callable*) → None

Add *callback* to the *_subs* list and then execute it.

Parameters

callback (*callable*) – Callable to execute.

10.1.2 Modin Configs List

Config Name	Env. Variable Name	Default Value	Description	Options
AsvDataSizeConfig	MODIN_ASV_DATASIZE_CONFIG		Allows to override default size of data (shapes).	
AsvImplementation	MODIN_ASV_USE_IMPL	modin	Allows to select a library that we will use for testing performance.	('modin', 'pandas')

continues on next page

Table 1 – continued from previous page

Config Name	Env. Variable Name	Default Value	Description	Options
AsyncReadMode	MODIN_ASYNC_READ_MODE	False	It does not wait for the end of reading information from the source. It basically means, that the reading function only launches tasks for the dataframe to be read/created, but not ensures that the construction is finalized by the time the reading function returns a dataframe. This option was brought to improve performance of reading/construction of Modin DataFrames, however it may also: 1. Increase the peak memory consumption. Since the garbage collection of the temporary objects created during the reading is now also lazy and will only be performed when the reading/construction is actually finished. 2. Can break situations when the source is manually deleted after the reading function returns a result, for example, when reading inside of a context-block that deletes the file on `__exit__()`.	
BenchmarkMode	MODIN_BENCHMARK_MODE	False	Whether or not to perform computations synchronously.	

continues on next page

Table 1 – continued from previous page

Config Name	Env. Variable Name	Default Value	Description	Options
CIAWSAccessKeyID	AWS_ACCESS_KEY_ID	foobar_key	Set to AWS_ACCESS_KEY_ID when running mock S3 tests for Modin in GitHub CI.	
CIAWSSecretAccessKey	AWS_SECRET_ACCESS_KEY	foobar_secret	Set to AWS_SECRET_ACCESS_KEY when running mock S3 tests for Modin in GitHub CI.	
CpuCount	MODIN_CPUS	multiprocessing.cpu_count()	How many CPU cores to use during initialization of the Modin engine.	
DaskThreadsPerWorker	MODIN_DASK_THREADS_PER_WORKER		Number of threads per Dask worker.	
DocModule	MODIN_DOC_MODULE	pandas	The module to use that will be used for docstrings. The value set here must be a valid, importable module. It should have a *DataFrame*, *Series*, and/or several APIs directly (e.g. *read_csv*).	
Engine	MODIN_ENGINE	Ray	Distribution engine to run queries by.	('Ray', 'Dask', 'Python', 'Unidist')
ExperimentalGroupbyImpl	MODIN_EXPERIMENTAL_GROUPBY	False	Set to true to use Modin's range-partitioning group by implementation. This parameter is deprecated. Use `RangePartitioning` instead.	
ExperimentalNumPyAPI	MODIN_EXPERIMENTAL_NUMPY_API	False	Set to true to use Modin's implementation of NumPy API. This parameter is deprecated. Use `ModinNumpy` instead.	
GithubCI	MODIN_GITHUB_CI	False	Set to true when running Modin in GitHub CI.	

continues on next page

Table 1 – continued from previous page

Config Name	Env. Variable Name	Default Value	Description	Options
GpuCount	MODIN_GPUS		How may GPU devices to utilize across the whole distribution.	
IsDebug	MODIN_DEBUG		Force Modin engine to be "Python" unless specified by $MODIN_ENGINE.	
IsExperimental	MODIN_EXPERIMENTAL		Whether to Turn on experimental features.	
IsRayCluster	MODIN_RAY_CLUSTER		Whether Modin is running on pre-initialized Ray cluster.	
LazyExecution	MODIN_LAZY_EXECUTION	Auto	Lazy execution mode. **Supported values:** *Auto* - the execution mode is chosen by the engine for each operation (default value). *On* - the lazy execution is performed wherever it's possible. *Off* - the lazy execution is disabled.	('Auto', 'On', 'Off')
LogFileSize	MODIN_LOG_FILE_SIZE	10	Max size of logs (in MBs) to store per Modin job.	
LogMemoryInterval	MODIN_LOG_MEMORY_INTERVAL	5	Interval (in seconds) to profile memory utilization for logging.	
LogMode	MODIN_LOG_MODE	disable	Set `LogMode` value if users want to opt-in.	('enable', 'disable')

continues on next page

Table 1 – continued from previous page

Config Name	Env. Variable Name	Default Value	Description	Options
Memory	MODIN_MEMORY		How much memory (in bytes) give to an execution engine. Notes: • In Ray case: the amount of memory to start the Plasma object store with. • In Dask case: the amount of memory that is given to each worker depending on CPUs used.	
MinPartitionSize	MODIN_MIN_PARTITION_SIZE	32	Minimum number of rows/columns in a single pandas partition split. Once a partition for a pandas dataframe has more than this many elements, Modin adds another partition.	
ModinNumpy	MODIN_NUMPY	False	Set to true to use Modin's implementation of NumPy API.	
NPartitions	MODIN_NPARTITIONS	equals to MODIN_CPUS env	How many partitions to use for a Modin DataFrame (along each axis).	
PersistentPickle	MODIN_PERSISTENT_PICKLE	False	Whether serialization should be persistent.	
ProgressBar	MODIN_PROGRESS_BAR	False	Whether or not to show the progress bar.	

continues on next page

Table 1 – continued from previous page

Config Name	Env. Variable Name	Default Value	Description	Options
RangePartitioning	MODIN_RANGE_PARTITIONING	False	Set to true to use Modin's range-partitioning implementation where possible. Please refer to documentation for cases where enabling this options would be beneficial: https://modin.readthedocs.io/en/stable/flow/modin/experimental/range_partitioning_groupby.html	
RangePartitioningGroupby	MODIN_RANGE_PARTITIONING_GROUPBY	False	Set to true to use Modin's range-partitioning group by implementation. This parameter is deprecated. Use `RangePartitioning` instead.	
RayInitCustomResources	MODIN_RAY_INIT_CUSTOM_RESOURCES		Ray node's custom resources to initialize with. Visit Ray documentation for more details: https://docs.ray.io/en/latest/ray-core/scheduling/resources.html#custom-resources Notes: Relying on Modin to initialize Ray, you should set this config for the proper initialization with custom resources.	
RayRedisAddress	MODIN_REDIS_ADDRESS		Redis address to connect to when running in Ray cluster.	
RayRedisPassword	MODIN_REDIS_PASSWORD	random string	What password to use for connecting to Redis.	

continues on next page

Table 1 – continued from previous page

Config Name	Env. Variable Name	Default Value	Description	Options
RayTaskCustomResources	MODIN_RAY_TASK_CUSTOM_RESOURCES		Ray node's custom resources to request them in tasks or actors. Visit Ray documentation for more details: https://docs.ray.io/en/latest/ray-core/scheduling/resources.html#custom-resources Notes: You can use this config to limit the parallelism for the entire workflow by setting the config at the very beginning. >>> import modin.config as cfg >>> cfg.RayTaskCustomResources.put({"special_hardware": 0.001}) This way each single remote task or actor will require 0.001 of "special_hardware" to run. You can also use this config to limit the parallelism for a certain operation by setting the config with context. >>> with context(RayTaskCustomResources={"special_hardware": 0.001}): … df.<op> This way each single remote task or actor will require 0.001 of "special_hardware" to run within the context only.	
ReadSqlEngine	MODIN_READ_SQL_ENGINE	Pandas	Engine to run *read_sql*.	('Pandas', 'Connectorx')
StorageFormat	MODIN_STORAGE_FORMAT	Pandas	Engine to run on a single node of distribution.	('Pandas', 'Cudf')

continues on next page

Table 1 – continued from previous page

Config Name	Env. Variable Name	Default Value	Description	Options
TestDatasetSize	MODIN_TEST_DATASET_SIZE		Dataset size for running some tests.	('Small', 'Normal', 'Big')
TestReadFromPostgres	MODIN_TEST_READ_FROM_POSTGRES	False	Set to true to test reading from Postgres.	
TestReadFromSqlServer	MODIN_TEST_READ_FROM_SQL_SERVER	False	Set to true to test reading from SQL server.	
TrackFileLeaks	MODIN_TEST_TRACK_FILE_LEAKS	True	Whether to track for open file handles leakage during testing.	

10.1.3 Usage Guide

See example of interaction with Modin configs below, as it can be seen config value can be set either by setting the environment variable or by using config API.

```python
import os

# Setting `MODIN_ENGINE` environment variable.
# Also can be set outside the script.
os.environ["MODIN_ENGINE"] = "Dask"

import modin.config
import modin.pandas as pd

# Checking initially set `Engine` config,
# which corresponds to `MODIN_ENGINE` environment
# variable
print(modin.config.Engine.get()) # prints 'Dask'

# Checking default value of `NPartitions`
print(modin.config.NPartitions.get()) # prints '8'

# Changing value of `NPartitions`
modin.config.NPartitions.put(16)
print(modin.config.NPartitions.get()) # prints '16'
```

One can also use config variables with a context manager in order to use some config only for a certain part of the code:

```python
import modin.config as cfg

# Default value for this config is 'False'
print(cfg.RangePartitioning.get()) # False

# Set the config to 'True' inside of the context-manager
with cfg.context(RangePartitioning=True):
    print(cfg.RangePartitioning.get()) # True
```

(continues on next page)

(continued from previous page)

```
    df.merge(...) # will use range-partitioning impl

# Once the context is over, the config gets back to its previous value
print(cfg.RangePartitioning.get()) # False

# You can also set multiple config at once when you pass a dictionary to 'cfg.context'
print(cfg.AsyncReadMode.get()) # False

with cfg.context(RangePartitioning=True, AsyncReadMode=True):
    print(cfg.RangePartitioning.get()) # True
    print(cfg.AsyncReadMode.get()) # True
print(cfg.RangePartitioning.get()) # False
print(cfg.AsyncReadMode.get()) # False
```

10.2 Modin Usage Examples

This section shows Modin usage examples in different scenarios like Modin on a local/remote cluster, the use of Modin spreadsheet.

10.2.1 Tutorials

The following tutorials cover the basic usage of Modin. Here is a one hour video tutorial that walks through these basic exercises.

- Exercise 1: Introduction to Modin [Source PandasOnRay, Source PandasOnDask]
- Exercise 2: Speed Improvements with Modin [Source PandasOnRay, Source PandasOnDask]
- Exercise 3: Defaulting to pandas with Modin [Source PandasOnRay, Source PandasOnDask]

The following tutorials covers more advanced features in Modin:

- Exercise 4: Experimental Features in Modin (Spreadsheet, Progress Bar) [Source PandasOnRay, Source PandasOnDask]
- Exercise 5: Setting up Modin in a Cluster Environment [Source PandasOnRay]
- Exercise 6: Running Modin in a Cluster Environment [Source PandasOnRay]

How to get required dependencies for the tutorial notebooks and to run them please refer to the respective README.md file.

10.2.2 Data Science Benchmarks

- Using Modin with the NYC Taxi Dataset [Source]
- Using Modin with the Census Dataset (coming soon…)
- Using Modin with the Plasticc Dataset (coming soon…)

10.2.3 Modin Spreadsheets

- Using Modin along with the Spreadsheets API [Source]

10.2.4 Modin with scikit-learn

- Modin for Machine Learning with scikit-learn [Source]

10.3 Advanced Usage

10.3.1 Pandas partitioning API

This page contains a description of the API to extract partitions from and build Modin Dataframes.

unwrap_partitions

`modin.distributed.dataframe.pandas.`**`unwrap_partitions`**(*api_layer_object: Union[DataFrame, Series], axis: Optional[int] = None, get_ip: bool = False*) → list

Unwrap partitions of the `api_layer_object`.

Parameters

- **`api_layer_object`** (`DataFrame` *or* `Series`) – The API layer object.
- **`axis`** (*{None, 0, 1}, default: None*) – The axis to unwrap partitions for (0 - row partitions, 1 - column partitions). If `axis is None`, the partitions are unwrapped as they are currently stored.
- **`get_ip`** (*bool, default: False*) – Whether to get node ip address to each partition or not.

Returns

A list of Ray.ObjectRef/Dask.Future to partitions of the `api_layer_object` if Ray/Dask is used as an engine.

Return type

list

Notes

If `get_ip=True`, a list of tuples of Ray.ObjectRef/Dask.Future to node ip addresses and partitions of the `api_layer_object`, respectively, is returned if Ray/Dask is used as an engine (i.e. `[(Ray.ObjectRef/Dask.Future, Ray.ObjectRef/Dask.Future), ...]`).

from_partitions

`modin.distributed.dataframe.pandas.from_partitions(partitions: list, axis: Optional[int], index: Optional[Union[ExtensionArray, ndarray, Index, Series, SequenceNotStr, range]] = None, columns: Optional[Union[ExtensionArray, ndarray, Index, Series, SequenceNotStr, range]] = None, row_lengths: Optional[list] = None, column_widths: Optional[list] = None) → DataFrame`

Create DataFrame from remote partitions.

Parameters

- **partitions** (*list*) – A list of Ray.ObjectRef/Dask.Future to partitions depending on the engine used. Or a list of tuples of Ray.ObjectRef/Dask.Future to node ip addresses and partitions depending on the engine used (i.e. `[(Ray.ObjectRef/Dask.Future, Ray.ObjectRef/Dask.Future), ...]`).
- **axis** (*{None, 0 or 1}*) – The `axis` parameter is used to identify what are the partitions passed. You have to set:
  - `axis=0` if you want to create DataFrame from row partitions
  - `axis=1` if you want to create DataFrame from column partitions
  - `axis=None` if you want to create DataFrame from 2D list of partitions
- **index** (*sequence, optional*) – The index for the DataFrame. Is computed if not provided.
- **columns** (*sequence, optional*) – The columns for the DataFrame. Is computed if not provided.
- **row_lengths** (*list, optional*) – The length of each partition in the rows. The “height” of each of the block partitions. Is computed if not provided.
- **column_widths** (*list, optional*) – The width of each partition in the columns. The “width” of each of the block partitions. Is computed if not provided.

Returns

DataFrame instance created from remote partitions.

Return type

modin.pandas.DataFrame

Notes

Pass *index*, *columns*, *row_lengths* and *column_widths* to avoid triggering extra computations of the metadata when creating a DataFrame.

Example

```
import modin.pandas as pd
from modin.distributed.dataframe.pandas import unwrap_partitions, from_partitions
import numpy as np
data = np.random.randint(0, 100, size=(2 ** 10, 2 ** 8))
df = pd.DataFrame(data)
partitions = unwrap_partitions(df, axis=0, get_ip=True)
print(partitions)
new_df = from_partitions(partitions, axis=0)
print(new_df)
```

10.3.2 Modin Spreadsheets API

Getting started

Install Modin-spreadsheet using pip:

```
pip install "modin[spreadsheet]"
```

The following code snippet creates a spreadsheet using the FiveThirtyEight dataset on labor force information by college majors (licensed under CC BY 4.0):

```
import modin.pandas as pd
import modin.experimental.spreadsheet as mss
df = pd.read_csv('https://raw.githubusercontent.com/fivethirtyeight/data/master/college-
↪majors/all-ages.csv')
spreadsheet = mss.from_dataframe(df)
spreadsheet
```

Basic Manipulations through User Interface

The Spreadsheet API allows users to manipulate the DataFrame with simple graphical controls for sorting, filtering, and editing.

Here are the instructions for each operation:

- **Sort**: Click on the column header of the column to sort on.
- **Filter**: Click on the filter button on the column header and apply the desired filter to the column. The filter dropdown changes depending on the type of the column. Multiple filters are automatically combined.
- **Edit Cell**: Double click on a cell and enter the new value.
- **Add Rows**: Click on the "Add Row" button in the toolbar to duplicate the last row in the DataFrame. The duplicated values provide a convenient default and can be edited as necessary.
- **Remove Rows**: Select row(s) and click the "Remove Row" button. Select a single row by clicking on it. Multiple rows can be selected with Cmd+Click (Windows: Ctrl+Click) on the desired rows or with Shift+Click to specify a range of rows.

Some of these operations can also be done through the spreadsheet's programmatic interface. Sorts and filters can be reset using the toolbar buttons. Edits and adding/removing rows can only be undone manually.

```python
import modin.pandas as pd
import modin.spreadsheet as mss
df = pd.read_csv('https://raw.githubusercontent.com/fivethirtyeight/data/master/college-majors/all-ages.csv')
spreadsheet = mss.from_dataframe(df)
spreadsheet
```

Add Row | Remove Row | Clear History | Filter History | Reset Filters | Reset Sort

	Major_code	Major	Major_category	Total	Employed	Employed_fu
0	1100	GENERAL AGRICULT…	Agriculture & Natural …	128148	90245	74078
1	1101	AGRICULTURE PROD…	Agriculture & Natural …	95326	76865	64240
2	1102	AGRICULTURAL ECO…	Agriculture & Natural …	33955	26321	22810
3	1103	ANIMAL SCIENCES	Agriculture & Natural …	103549	81177	64937
4	1104	FOOD SCIENCE	Agriculture & Natural …	24280	17281	12722
5	1105	PLANT SCIENCE AND…	Agriculture & Natural …	79409	63043	51077
6	1106	SOIL SCIENCE	Agriculture & Natural …	6586	4926	4042
7	1199	MISCELLANEOUS AG…	Agriculture & Natural …	8549	6392	5074
8	1301	ENVIRONMENTAL SC…	Biology & Life Science	106106	87602	65238
9	1302	FORESTRY	Agriculture & Natural …	69447	48228	39613
10	1303	NATURAL RESOURC…	Agriculture & Natural …	83188	65937	50595
11	1401	ARCHITECTURE	Engineering	294692	216770	163020
12	1501	AREA ETHNIC AND C…	Humanities & Liberal …	103740	75798	50530
13	1901	COMMUNICATIONS	Communications & Jo…	987676	790696	595739
14	1902	JOURNALISM	Communications & Jo…	418104	314438	235407

```python
# ---- spreadsheet transformation history ----
unfiltered_df = df.copy()
```

Virtual Rendering

The spreadsheet will only render data based on the user's viewport. This allows for quick rendering even on very large DataFrames because only a handful of rows are loaded at any given time. As a result, scrolling and viewing your data is smooth and responsive.

Transformation History and Exporting Code

All operations on the spreadsheet are recorded and are easily exported as code for sharing or reproducibility. This history is automatically displayed in the history cell, which is generated below the spreadsheet whenever the spreadsheet widget is displayed. The history cell is displayed on default, but this can be turned off. Modin Spreadsheet API provides a few methods for interacting with the history:

- *SpreadsheetWidget.get_history()*: Retrieves the transformation history in the form of reproducible code.
- *SpreadsheetWidget.filter_relevant_history(persist=True)*: Returns the transformation history that contains only code relevant to the final state of the spreadsheet. The *persist* parameter determines whether the internal state and the displayed history is also filtered.
- *SpreadsheetWidget.reset_history()*: Clears the history of transformation.

Customizable Interface

The spreadsheet widget provides a number of options that allows the user to change the appearance and the interactivity of the spreadsheet. Options include:

- Row height/Column width
- Preventing edits, sorts, or filters on the whole spreadsheet or on a per-column basis
- Hiding the toolbar and history cell
- Float precision
- Highlighting of cells and rows
- Viewport size

Converting Spreadsheets To and From Dataframes

`modin.experimental.spreadsheet.general.`**`from_dataframe`**(*dataframe*, *show_toolbar=None*, *show_history=None*, *precision=None*, *grid_options=None*, *column_options=None*, *column_definitions=None*, *row_edit_callback=None*)

Renders a DataFrame or Series as an interactive spreadsheet, represented by an instance of the `SpreadsheetWidget` class. The `SpreadsheetWidget` instance is constructed using the options passed in to this function. The `dataframe` argument to this function is used as the `df` kwarg in call to the SpreadsheetWidget constructor, and the rest of the parameters are passed through as is.

If the `dataframe` argument is a Series, it will be converted to a DataFrame before being passed in to the SpreadsheetWidget constructor as the `df` kwarg.

Return type
SpreadsheetWidget

Parameters

- **dataframe** (`DataFrame`) – The DataFrame that will be displayed by this instance of SpreadsheetWidget.
- **grid_options** (*dict*) – Options to use when creating the SlickGrid control (i.e. the interactive grid). See the Notes section below for more information on the available options, as well as the default options that this widget uses.
- **precision** (*integer*) – The number of digits of precision to display for floating-point values. If unset, we use the value of *pandas.get_option('display.precision')*.
- **show_toolbar** (*bool*) – Whether to show a toolbar with options for adding/removing rows. Adding/removing rows is an experimental feature which only works with DataFrames that have an integer index.
- **show_history** (*bool*) – Whether to show the cell containing the spreadsheet transformation history.
- **column_options** (*dict*) – Column options that are to be applied to every column. See the Notes section below for more information on the available options, as well as the default options that this widget uses.

- **column_definitions** (*dict*) – Column options that are to be applied to individual columns. The keys of the dict should be the column names, and each value should be the column options for a particular column, represented as a dict. The available options for each column are the same options that are available to be set for all columns via the `column_options` parameter. See the Notes section below for more information on those options.
- **row_edit_callback** (*callable*) – A callable that is called to determine whether a particular row should be editable or not. Its signature should be `callable(row)`, where `row` is a dictionary which contains a particular row's values, keyed by column name. The callback should return True if the provided row should be editable, and False otherwise.

Notes

The following dictionary is used for `grid_options` if none are provided explicitly:

```
{
    # SlickGrid options
    'fullWidthRows': True,
    'syncColumnCellResize': True,
    'forceFitColumns': False,
    'defaultColumnWidth': 150,
    'rowHeight': 28,
    'enableColumnReorder': False,
    'enableTextSelectionOnCells': True,
    'editable': True,
    'autoEdit': False,
    'explicitInitialization': True,

    # Modin-spreadsheet options
    'maxVisibleRows': 15,
    'minVisibleRows': 8,
    'sortable': True,
    'filterable': True,
    'highlightSelectedCell': False,
    'highlightSelectedRow': True
}
```

The first group of options are SlickGrid "grid options" which are described in the SlickGrid documentation.

The second group of option are options that were added specifically for modin-spreadsheet and therefore are not documented in the SlickGrid documentation. The following bullet points describe these options.

- **maxVisibleRows** The maximum number of rows that modin-spreadsheet will show.
- **minVisibleRows** The minimum number of rows that modin-spreadsheet will show
- **sortable** Whether the modin-spreadsheet instance will allow the user to sort columns by clicking the column headers. When this is set to `False`, nothing will happen when users click the column headers.
- **filterable** Whether the modin-spreadsheet instance will allow the user to filter the grid. When this is set to `False` the filter icons won't be shown for any columns.
- **highlightSelectedCell** If you set this to True, the selected cell will be given a light blue border.
- **highlightSelectedRow** If you set this to False, the light blue background that's shown by default for selected rows will be hidden.

The following dictionary is used for `column_options` if none are provided explicitly:

```
{
    # SlickGrid column options
    'defaultSortAsc': True,
    'maxWidth': None,
    'minWidth': 30,
    'resizable': True,
    'sortable': True,
    'toolTip': "",
    'width': None

    # Modin-spreadsheet column options
    'editable': True,
}
```

The first group of options are SlickGrid "column options" which are described in the SlickGrid documentation.

The `editable` option was added specifically for modin-spreadsheet and therefore is not documented in the SlickGrid documentation. This option specifies whether a column should be editable or not.

See also:

`set_defaults`
Permanently set global defaults for the parameters of `show_grid`, with the exception of the `dataframe` and `column_definitions` parameters, since those depend on the particular set of data being shown by an instance, and therefore aren't parameters we would want to set for all SpreadsheetWidget instances.

`set_grid_option`
Permanently set global defaults for individual grid options. Does so by changing the defaults that the `show_grid` method uses for the `grid_options` parameter.

`SpreadsheetWidget`
The widget class that is instantiated and returned by this method.

`modin.experimental.spreadsheet.general.`**`to_dataframe`**(*spreadsheet*)

Get a copy of the DataFrame that reflects the current state of the `spreadsheet` SpreadsheetWidget instance UI. This includes any sorting or filtering changes, as well as edits that have been made by double clicking cells.

Return type
DataFrame

Parameters
`spreadsheet` (*`SpreadsheetWidget`*) – The SpreadsheetWidget instance that DataFrame that will be displayed by this instance of SpreadsheetWidget.

Further API Documentation

class `modin_spreadsheet.grid.`**`SpreadsheetWidget`**(***kwargs: Any*)

The widget class which is instantiated by the `show_grid` method. This class can be constructed directly but that's not recommended because then default options have to be specified explicitly (since default options are normally provided by the `show_grid` method).

The constructor for this class takes all the same parameters as `show_grid`, with one exception, which is that the required `data_frame` parameter is replaced by an optional keyword argument called `df`.

See also:

show_grid

The method that should be used to construct SpreadsheetWidget instances, because it provides reasonable defaults for all of the modin-spreadsheet options.

df

Get/set the DataFrame that's being displayed by the current instance. This DataFrame will NOT reflect any sorting/filtering/editing changes that are made via the UI. To get a copy of the DataFrame that does reflect sorting/filtering/editing changes, use the `get_changed_df()` method.

Type
DataFrame

grid_options

Get/set the grid options being used by the current instance.

Type
dict

precision

Get/set the precision options being used by the current instance.

Type
integer

show_toolbar

Get/set the show_toolbar option being used by the current instance.

Type
bool

show_history

Get/set the show_history option being used by the current instance.

Type
bool

column_options

Get/set the column options being used by the current instance.

Type
bool

column_definitions

Get/set the column definitions (column-specific options) being used by the current instance.

Type
bool

add_row(*row=None*)

Append a row at the end of the DataFrame. Values for the new row can be provided via the `row` argument, which is optional for DataFrames that have an integer index, and required otherwise. If the `row` argument is not provided, the last row will be duplicated and the index of the new row will be the index of the last row plus one.

Parameters
row (*list (default: None)*) – A list of 2-tuples of (column name, column value) that specifies the values for the new row.

See also:

`SpreadsheetWidget.remove_rows`
The method for removing a row (or rows).

change_grid_option(*option_name*, *option_value*)

Change a SlickGrid grid option without rebuilding the entire grid widget. Not all options are supported at this point so this method should be considered experimental.

Parameters

- **option_name** (*str*) – The name of the grid option to be changed.
- **option_value** (*str*) – The new value for the grid option.

change_selection(*rows=[]*)

Select a row (or rows) in the UI. The indices of the rows to select are provided via the optional `rows` argument.

Parameters

rows (*list (default: [])*) – A list of indices of the rows to select. For a multi-indexed DataFrame, each index in the list should be a tuple, with each value in each tuple corresponding to a level of the MultiIndex. The default value of `[]` results in the no rows being selected (i.e. it clears the selection).

edit_cell(*index*, *column*, *value*)

Edit a cell of the grid, given the index and column of the cell to edit, as well as the new value of the cell. Results in a `cell_edited` event being fired.

Parameters

- **index** (*object*) – The index of the row containing the cell that is to be edited.
- **column** (*str*) – The name of the column containing the cell that is to be edited.
- **value** (*object*) – The new value for the cell.

get_changed_df()

Get a copy of the DataFrame that was used to create the current instance of SpreadsheetWidget which reflects the current state of the UI. This includes any sorting or filtering changes, as well as edits that have been made by double clicking cells.

Return type

DataFrame

get_selected_df()

Get a DataFrame which reflects the current state of the UI and only includes the currently selected row(s). Internally it calls `get_changed_df()` and then filters down to the selected rows using `iloc`.

Return type

DataFrame

get_selected_rows()

Get the currently selected rows.

Return type

List of integers

off(*names*, *handler*)

Remove a modin-spreadsheet event handler that was registered with the current instance's `on` method.

Parameters

- **names** (*list, str, All (default: All)*) – The names of the events for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all events.
- **handler** (*callable*) – A callable that was previously registered with the current instance's on method.

See also:

`SpreadsheetWidget.on`
The method for hooking up instance-level handlers that this `off` method can remove.

`on`(*names*, *handler*)

Setup a handler to be called when a user interacts with the current instance.

Parameters

- **names** (*list, str, All*) – If names is All, the handler will apply to all events. If a list of str, handler will apply to all events named in the list. If a str, the handler will apply just the event with that name.
- **handler** (*callable*) – A callable that is called when the event occurs. Its signature should be `handler(event, spreadsheet_widget)`, where `event` is a dictionary and `spreadsheet_widget` is the SpreadsheetWidget instance that fired the event. The `event` dictionary at least holds a `name` key which specifies the name of the event that occurred.

Notes

Here's the list of events that you can listen to on SpreadsheetWidget instances via the `on` method:

```
[
    'cell_edited',
    'selection_changed',
    'viewport_changed',
    'row_added',
    'row_removed',
    'filter_dropdown_shown',
    'filter_changed',
    'sort_changed',
    'text_filter_viewport_changed',
    'json_updated'
]
```

The following bullet points describe the events listed above in more detail. Each event bullet point is followed by sub-bullets which describe the keys that will be included in the `event` dictionary for each event.

- **cell_edited** The user changed the value of a cell in the grid.
  - **index** The index of the row that contains the edited cell.
  - **column** The name of the column that contains the edited cell.
  - **old** The previous value of the cell.
  - **new** The new value of the cell.
- **filter_changed** The user changed the filter setting for a column.
  - **column** The name of the column for which the filter setting was changed.

- **filter_dropdown_shown** The user showed the filter control for a column by clicking the filter icon in the column's header.
  - **column** The name of the column for which the filter control was shown.
- **json_updated** A user action causes SpreadsheetWidget to send rows of data (in json format) down to the browser. This happens as a side effect of certain actions such as scrolling, sorting, and filtering.
  - **triggered_by** The name of the event that resulted in rows of data being sent down to the browser. Possible values are `change_viewport`, `change_filter`, `change_sort`, `add_row`, `remove_row`, and `edit_cell`.
  - **range** A tuple specifying the range of rows that have been sent down to the browser.
- **row_added** The user added a new row using the "Add Row" button in the grid toolbar.
  - **index** The index of the newly added row.
  - **source** The source of this event. Possible values are `api` (an api method call) and `gui` (the grid interface).
- **row_removed** The user added removed one or more rows using the "Remove Row" button in the grid toolbar.
  - **indices** The indices of the removed rows, specified as an array of integers.
  - **source** The source of this event. Possible values are `api` (an api method call) and `gui` (the grid interface).
- **selection_changed** The user changed which rows were highlighted in the grid.
  - **old** An array specifying the indices of the previously selected rows.
  - **new** The indices of the rows that are now selected, again specified as an array.
  - **source** The source of this event. Possible values are `api` (an api method call) and `gui` (the grid interface).
- **sort_changed** The user changed the sort setting for the grid.
  - **old** The previous sort setting for the grid, specified as a dict with the following keys:
    * **column** The name of the column that the grid was sorted by
    * **ascending** Boolean indicating ascending/descending order
  - **new** The new sort setting for the grid, specified as a dict with the following keys:
    * **column** The name of the column that the grid is currently sorted by
    * **ascending** Boolean indicating ascending/descending order
- **text_filter_viewport_changed** The user scrolled the new rows into view in the filter dropdown for a text field.
  - **column** The name of the column whose filter dropdown is visible
  - **old** A tuple specifying the previous range of visible rows in the filter dropdown.
  - **new** A tuple specifying the range of rows that are now visible in the filter dropdown.
- **viewport_changed** The user scrolled the new rows into view in the grid.
  - **old** A tuple specifying the previous range of visible rows.
  - **new** A tuple specifying the range of rows that are now visible.

The `event` dictionary for every type of event will contain a `name` key specifying the name of the event that occurred. That key is excluded from the lists of keys above to avoid redundacy.

See also:

`on`
Same as the instance-level `on` method except it listens for events on all instances rather than on an individual SpreadsheetWidget instance.

`SpreadsheetWidget.off`
Unhook a handler that was hooked up using the instance-level `on` method.

remove_row(*rows=None*)

Alias for `remove_rows`, which is provided for convenience because this was the previous name of that method.

remove_rows(*rows=None*)

Remove a row (or rows) from the DataFrame. The indices of the rows to remove can be provided via the optional `rows` argument. If the `rows` argument is not provided, the row (or rows) that are currently selected in the UI will be removed.

Parameters
rows (*list (default: None)*) – A list of indices of the rows to remove from the DataFrame. For a multi-indexed DataFrame, each index in the list should be a tuple, with each value in each tuple corresponding to a level of the MultiIndex.

See also:

`SpreadsheetWidget.add_row`
The method for adding a row.

`SpreadsheetWidget.remove_row`
Alias for this method.

toggle_editable()

Change whether the grid is editable or not, without rebuilding the entire grid widget.

10.3.3 Progress Bar

The progress bar allows users to see the estimated progress and completion time of each line they run, in environments such as a shell or Jupyter notebook.

Quickstart

The progress bar uses the *tqdm* library to visualize displays:

```
pip install tqdm
```

Import the progress bar into your notebook by running the following:

```
from modin.config import ProgressBar
ProgressBar.enable()
```

10.3.4 Distributed XGBoost on Modin

Modin provides an implementation of distributed XGBoost machine learning algorithm on Modin DataFrames. Please note that this feature is experimental and behavior or interfaces could be changed.

Install XGBoost on Modin

Modin comes with all the dependencies except `xgboost` package by default. Currently, distributed XGBoost on Modin is only supported on the Ray execution engine, therefore, see the *installation page* for more information on installing Modin with the Ray engine. To install `xgboost` package you can use `pip`:

```
pip install xgboost
```

XGBoost Train and Predict

Distributed XGBoost functionality is placed in `modin.experimental.xgboost` module. `modin.experimental.xgboost` provides a drop-in replacement API for `train` and `Booster.predict` xgboost functions.

Module holds public interfaces for Modin XGBoost.

`modin.experimental.xgboost.`**`train`**(*params: Dict*, *dtrain: DMatrix*, **args*, *evals=()*, *num_actors: Optional[int] = None*, *evals_result: Optional[Dict] = None*, ***kwargs*)

Run distributed training of XGBoost model.

During work it evenly distributes *dtrain* between workers according to IP addresses partitions (in case of not even distribution of *dtrain* over nodes, some partitions will be re-distributed between nodes), runs xgb.train on each worker for subset of *dtrain* and reduces training results of each worker using Rabit Context.

Parameters

- **params** (*dict*) – Booster params.
- **dtrain** (*modin.experimental.xgboost.DMatrix*) – Data to be trained against.
- ***args** (*iterable*) – Other parameters for *xgboost.train*.
- **evals** (*list of pairs (modin.experimental.xgboost.DMatrix, str), default: empty*) – List of validation sets for which metrics will evaluated during training. Validation metrics will help us track the performance of the model.
- **num_actors** (*int, optional*) – Number of actors for training. If unspecified, this value will be computed automatically.
- **evals_result** (*dict, optional*) – Dict to store evaluation results in.
- ****kwargs** (*dict*) – Other parameters are the same as *xgboost.train*.

Returns
A trained booster.

Return type
modin.experimental.xgboost.Booster

class `modin.experimental.xgboost.`**`Booster`**(*params=None*, *cache=()*, *model_file=None*)

A Modin Booster of XGBoost.

Booster is the model of XGBoost, that contains low level routines for training, prediction and evaluation.

Parameters

- **params** (*dict, optional*) – Parameters for boosters.
- **cache** (*list, default: empty*) – List of cache items.
- **model_file** (*string/os.PathLike/xgb.Booster/bytearray, optional*) – Path to the model file if it's string or PathLike or xgb.Booster.

predict(*data: DMatrix*, ***kwargs*)

Run distributed prediction with a trained booster.

During execution it runs `xgb.predict` on each worker for subset of *data* and creates Modin DataFrame with prediction results.

Parameters

- **data** (*modin.experimental.xgboost.DMatrix*) – Input data used for prediction.
- ****kwargs** (*dict*) – Other parameters are the same as for `xgboost.Booster.predict`.

Returns
Modin DataFrame with prediction results.

Return type
modin.pandas.DataFrame

ModinDMatrix

Data is passed to `modin.experimental.xgboost` functions via a Modin `DMatrix` object.

Module holds public interfaces for Modin XGBoost.

class modin.experimental.xgboost.**DMatrix**(*data*, *label=None*, *missing=None*, *silent=False*, *feature_names=None*, *feature_types=None*, *feature_weights=None*, *enable_categorical=None*)

DMatrix holds references to partitions of Modin DataFrame.

On init stage unwrapping partitions of Modin DataFrame is started.

Parameters

- **data** (*modin.pandas.DataFrame*) – Data source of DMatrix.
- **label** (*modin.pandas.DataFrame or modin.pandas.Series, optional*) – Labels used for training.
- **missing** (*float, optional*) – Value in the input data which needs to be present as a missing value. If `None`, defaults to `np.nan`.
- **silent** (*boolean, optional*) – Whether to print messages during construction or not.
- **feature_names** (*list, optional*) – Set names for features.
- **feature_types** (*list, optional*) – Set types for features.
- **feature_weights** (*array_like, optional*) – Set feature weights for column sampling.
- **enable_categorical** (*boolean, optional*) – Experimental support of specializing for categorical features.

Notes

Currently DMatrix doesn't support *weight*, *base_margin*, *nthread*, *group*, *qid*, *label_lower_bound*, *label_upper_bound* parameters.

property feature_names

Get column labels.

Return type
Column labels.

property feature_types

Get column types.

Return type
Column types.

get_dmatrix_params()

Get dict of DMatrix parameters excluding *self.data/self.label*.

Return type
dict

get_float_info(*name*)

Get float property from the DMatrix.

Parameters
name (*str*) – The field name of the information.

Return type
A NumPy array of float information of the data.

num_col()

Get number of columns.

Return type
int

num_row()

Get number of rows.

Return type
int

set_info(*, *label=None*, *feature_names=None*, *feature_types=None*, *feature_weights=None*) → None

Set meta info for DMatrix.

Parameters

- **label** (*modin.pandas.DataFrame or modin.pandas.Series, optional*) – Labels used for training.
- **feature_names** (*list, optional*) – Set names for features.
- **feature_types** (*list, optional*) – Set types for features.
- **feature_weights** (*array_like, optional*) – Set feature weights for column sampling.

Currently, the Modin `DMatrix` supports `modin.pandas.DataFrame` only as an input.

A Single Node / Cluster setup

The XGBoost part of Modin uses a Ray resources by similar way as all Modin functions.

To start the Ray runtime on a single node:

```
import ray
# Look at the Ray documentation with respect to the Ray configuration suited to you most.
ray.init()
```

If you already had the Ray cluster you can connect to it by next way:

```
import ray
ray.init(address='auto')
```

A detailed information about initializing the Ray runtime you can find in starting ray page.

Usage example

In example below we train XGBoost model using the Iris Dataset and get prediction on the same data. All processing will be in a *single node* mode.

```
from sklearn import datasets

import ray
# Look at the Ray documentation with respect to the Ray configuration suited to you most.
ray.init() # Start the Ray runtime for single-node

import modin.pandas as pd
import modin.experimental.xgboost as xgb

# Load iris dataset from sklearn
iris = datasets.load_iris()

# Create Modin DataFrames
X = pd.DataFrame(iris.data)
y = pd.DataFrame(iris.target)

# Create DMatrix
dtrain = xgb.DMatrix(X, y)
dtest = xgb.DMatrix(X, y)

# Set training parameters
xgb_params = {
    "eta": 0.3,
    "max_depth": 3,
    "objective": "multi:softprob",
    "num_class": 3,
    "eval_metric": "mlogloss",
}
steps = 20

# Create dict for evaluation results
evals_result = dict()
```

(continues on next page)

(continued from previous page)

```
# Run training
model = xgb.train(
    xgb_params,
    dtrain,
    steps,
    evals=[(dtrain, "train")],
    evals_result=evals_result
)

# Print evaluation results
print(f'Evals results:\n{evals_result}')

# Predict results
prediction = model.predict(dtest)

# Print prediction results
print(f'Prediction results:\n{prediction}')
```

10.3.5 Modin Logging

Modin logging offers users greater insight into their queries by logging internal Modin API calls, partition metadata, and profiling system memory. When Modin logging is enabled (default disabled), log files are written to a local `.modin` directory at the same directory level as the notebook/script used to run Modin.

The logs generated by Modin Logging will be written to a `.modin/logs/job_<uuid>` directory, uniquely named after the job uuid. The logs that contain the Modin API stack traces are named `trace.log`. The logs that contain the memory utilization metrics are named `memory.log`. By default, if any log file exceeds 10MB (configurable with `LogFileSize`), that file will be saved and a separate log file will be created. For instance, if users have 20MB worth of Modin API logs, they can expect to find `trace.log.1` and `trace.log.2` in the `.modin/logs/job_<uuid>` directory. After `10 * LogFileSize` MB or by default 100MB of logs, the logs will rollover and the original log files beginning with `trace.log.1` will be overwritten with the new log lines.

Developer Warning: In some cases, running services like JupyterLab in the `modin/modin` directory may result in circular dependency issues. This is due to a naming conflict between the `modin/logging` directory and the Python `logging` module, which may be used as a default in such environments. To resolve this, please run Jupyterlab or other similar services from directories other than `modin/modin`.

Usage examples

In the example below, we enable logging for internal Modin API calls, partition metadata and memory profiling. We can set the granularity (in seconds) at which the system memory utilization is logged using `LogMemoryInterval`. We can also set the maximum size of the logs (in MBs) using `LogFileSize`.

```
import modin.pandas as pd
from modin.config import LogMode, LogMemoryInterval, LogFileSize
LogMode.enable()
LogMemoryInterval.put(2) # Defaults to 5 seconds, new interval is 2 seconds
LogFileSize.put(5) # Defaults to 10 MB per log file, new size is 5 MB

# User code goes here
```

Disable Modin logging like so:

```
import modin.pandas as pd
from modin.config import LogMode
LogMode.disable()

# User code goes here
```

In Modin the lower-level functionality is logged in debug level, and higher level functionality in info level. By default when logging is enabled in Modin, both high level and low level functionality are logged. The below example script could be used to switch between logging all functions vs only logging higher level functions. Setting logger level to `logging.INFO` logs only higher level functions.

```
import modin.pandas as pd
from modin.logging.config import get_logger
from modin.config import LogMode
import logging
LogMode.enable()
logger = get_logger()
logger.setLevel(logging.INFO) # Replace with logger.setLevel(logging.DEBUG)  for lower␣
↪level logs
df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
df = pd.concat([df, df])
```

Debugging from user defined functions:

> **Warning:** When attempting to use Modin logging in user defined functions that execute in workers for logging lower-level operators as in example below, multiple log directories `.modin/logs/job_**` would be created for each worker executing the UDF.

```
import modin.pandas as pd

def udf(x):
    from modin.config import LogMode

    LogMode.enable()

    return x + 1

modin_df = pd.DataFrame([0, 1, 2, 3])
print(modin_df.map(udf))
```

So the **recommended** approach would be to use a different logger as in the below snipet to log from user defined functions that execute on workers. Below is an an example to log from UDF. For this the logger config has to be specified inside the UDF that would execute on a remote worker.

```
import logging
import modin.pandas as pd

def udf(x):
    logging.basicConfig(filename='modin_udf.log', level=logging.INFO)
    logging.info("This log message will be written to modin_udf.log ")
```

(continues on next page)

(continued from previous page)

```
    # User code goes here
    return x + 1

modin_df = pd.DataFrame([0, 1, 2, 3])
print(modin_df.map(udf))
```

10.3.6 Batch Pipline API Usage Guide

Modin provides an experimental batching feature that pipelines row-parallel queries. This feature is currently only supported for the PandasOnRay engine. Please note that this feature is experimental and behavior or interfaces could be changed.

Usage examples

In examples below we build and run some pipelines. It is important to note that the queries passed to the pipeline operate on Modin DataFrame partitions, which are backed by pandas. When using pandas- module level functions, please make sure to import and use pandas rather than modin.pandas.

Simple Batch Pipelining

This example walks through a simple batch pipeline in order to familiarize the user with the API.

```
from modin.experimental.batch import PandasQueryPipeline
import modin.pandas as pd
import numpy as np

df = pd.DataFrame(
    np.random.randint(0, 100, (100, 100)),
    columns=[f"col {i}" for i in range(1, 101)],
) # Build the dataframe we will pipeline.
pipeline = PandasQueryPipeline(df) # Build the pipeline.
pipeline.add_query(lambda df: df + 1, is_output=True) # Add the first query and specify␣
↪that
                                                      # it is an output query.
pipeline.add_query(
    lambda df: df.rename(columns={f"col {i}":f"col {i-1}" for i in range(1, 101)})
) # Add a second query.
pipeline.add_query(
    lambda df: df.drop(columns=['col 99']),
    is_output=True,
) # Add a third query and specify that it is an output query.
new_df = pd.DataFrame(
    np.ones((100, 100)),
    columns=[f"col {i}" for i in range(1, 101)],
) # Build a second dataframe that we will pipeline now instead.
pipeline.update_df(new_df) # Update the dataframe that we will pipeline to be `new_df`
                           # instead of `df`.
result_dfs = pipeline.compute_batch() # Begin batch processing.
```

(continues on next page)

(continued from previous page)

```
# Print pipeline results
print(f"Result of Query 1:\n{result_dfs[0]}")
print(f"Result of Query 2:\n{result_dfs[1]}")
# Output IDs can also be specified
pipeline = PandasQueryPipeline(df) # Build the pipeline.
pipeline.add_query(
    lambda df: df + 1,
    is_output=True,
    output_id=1,
) # Add the first query, specify that it is an output query, as well as specify an␣
↪output id.
pipeline.add_query(
    lambda df: df.rename(columns={f"col {i}":f"col {i-1}" for i in range(1, 101)})
) # Add a second query.
pipeline.add_query(
    lambda df: df.drop(columns=['col 99']),
    is_output=True,
    output_id=2,
) # Add a third query, specify that it is an output query, and specify an output_id.
result_dfs = pipeline.compute_batch() # Begin batch processing.

# Print pipeline results - should be a dictionary mapping Output IDs to resulting␣
↪dataframes:
print(f"Mapping of Output ID to dataframe:\n{result_dfs}")
# Print results
for query_id, res_df in result_dfs.items():
    print(f"Query {query_id} resulted in\n{res_df}")
```

Batch Pipelining with Postprocessing

A postprocessing function can also be provided when calling `pipeline.compute_batch`. The example below runs a similar pipeline as above, but the postprocessing function writes the output dfs to a parquet file.

```
from modin.experimental.batch import PandasQueryPipeline
import modin.pandas as pd
import numpy as np
import os
import shutil

df = pd.DataFrame(
    np.random.randint(0, 100, (100, 100)),
    columns=[f"col {i}" for i in range(1, 101)],
) # Build the dataframe we will pipeline.
pipeline = PandasQueryPipeline(df) # Build the pipeline.
pipeline.add_query(
    lambda df: df + 1,
    is_output=True,
    output_id=1,
) # Add the first query, specify that it is an output query, as well as specify an␣
↪output id.
```

(continues on next page)

(continued from previous page)

```
pipeline.add_query(
    lambda df: df.rename(columns={f"col {i}":f"col {i-1}" for i in range(1, 101)})
) # Add a second query.
pipeline.add_query(
    lambda df: df.drop(columns=['col 99']),
    is_output=True,
    output_id=2,
) # Add a third query, specify that it is an output query, and specify an output_id.
def postprocessing_func(df, output_id, partition_id):
    filepath = f"query_{output_id}/"
    os.makedirs(filepath, exist_ok=True)
    filepath += f"part-{partition_id:04d}.parquet"
    df.to_parquet(filepath)
    return df
result_dfs = pipeline.compute_batch(
    postprocessor=postprocessing_func,
    pass_partition_id=True,
    pass_output_id=True,
) # Begin computation, pass in a postprocessing function, and specify that partition ID␣
↪and
  # output ID should be passed to that postprocessing function.

print(os.system("ls query_1/")) # Should show `NPartitions.get()` parquet files - which
                                # correspond to partitions of the output of query 1.
print(os.system("ls query_2/")) # Should show `NPartitions.get()` parquet files - which
                                # correspond to partitions of the output of query 2.

for query_id, res_df in result_dfs.items():
    written_df = pd.read_parquet(f"query_{query_id}/")
    shutil.rmtree(f"query_{query_id}/") # Clean up
    print(f"Written and Computed DF are " +
          f"{'equal' if res_df.equals(written_df) else 'not equal'} for query {query_id}
↪")
```

Batch Pipelining with Fan Out

If the input dataframe to a query is small (consisting of only one partition), it is possible to induce additional parallelism using the `fan_out` argument. The `fan_out` argument replicates the input partition, applies the query to each replica, and then coalesces all of the replicas back to one partition using the `reduce_fn` that must also be specified when `fan_out` is `True`.

It is possible to control the parallelism via the `num_partitions` parameter passed to the constructor of the `PandasQueryPipeline`. This parameter designates the desired number of partitions, and defaults to `NPartitions.get()` when not specified. During fan out, the input partition is replicated `num_partitions` times. In the previous examples, `num_partitions` was not specified, and so defaulted to `NPartitions.get()`.

The example below demonstrates the usage of `fan_out` and `num_partitions`. We first demonstrate an example of a function that would benefit from this computation pattern:

```
import glob
from PIL import Image
import torchvision.transforms as T
```

(continues on next page)

(continued from previous page)

```
import torchvision

transforms = T.Compose([T.ToTensor()])
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
model.eval()
COCO_INSTANCE_CATEGORY_NAMES = [
    '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop sign',
    'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
    'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/A', 'N/A',
    'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
    'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket',
    'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
    'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
    'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table',
    'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell␣
↪phone',
    'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A', 'book',
    'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'
]

def contains_cat(image, model):
    image = transforms(image)
    labels = [COCO_INSTANCE_CATEGORY_NAMES[i] for i in model([image])[0]['labels']]
    return 'cat' in labels

def serial_query(df):
    """
    This function takes as input a dataframe with a single row corresponding to a folder
    containing images to parse. Each image in the folder is passed through a neural␣
↪network
    that detects whether it contains a cat, in serial, and a new column is computed for␣
↪the
    dataframe that counts the number of images containing cats.

    Parameters
    ----------
    df : a dataframe
        The dataframe to process

    Returns
    -------
    The same dataframe as before, with an additional column containing the count of␣
↪images
    containing cats.
    """
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
    model.eval()
    img_folder = df['images'][0]
    images = sorted(glob.glob(f"{img_folder}/*.jpg"))
    cats = 0
    for img in images:
```

(continues on next page)

(continued from previous page)

```
        cats = cats + 1 if contains_cat(Image.open(img), model) else cats
    df['cat_count'] = cats
    return df
```

To download the image files to test out this code, run the following bash script, which downloads the images from the fast-ai-coco S3 bucket to a folder called `images` in your current working directory:

```
aws s3 cp s3://fast-ai-coco/coco_tiny.tgz . --no-sign-request; tar -xf coco_tiny.tgz;␣
↪mkdir \
    images; mv coco_tiny/train/* images/; rm -rf coco_tiny; rm -rf coco_tiny.tgz
```

We can pipeline that code like so:

```
import modin.pandas as pd
from modin.experimental.batch import PandasQueryPipeline
from time import time
df = pd.DataFrame([['images']], columns=['images'])
pipeline = PandasQueryPipeline(df)
pipeline.add_query(serial_query, is_output=True)
serial_start = time()
df_with_cat_count = pipeline.compute_batch()[0]
serial_end = time()
print(f"Result of pipeline:\n{df_with_cat_count}")
```

We can induce *8x* parallelism into the pipeline above by combining the `fan_out` and `num_partitions` parameters like so:

```
import modin.pandas as pd
from modin.experimental.batch import PandasQueryPipeline
import shutil
from time import time
df = pd.DataFrame([['images']], columns=['images'])
desired_num_partitions = 8
def parallel_query(df, partition_id):
    """
    This function takes as input a dataframe with a single row corresponding to a folder
    containing images to parse. It parses `total_images/desired_num_partitions` images␣
↪every
    time it is called. A new column is computed for the dataframe that counts the number␣
↪of
    images containing cats.

    Parameters
    ----------
    df : a dataframe
        The dataframe to process
    partition_id : int
        The partition id of the dataframe that this function runs on.

    Returns
    -------
    The same dataframe as before, with an additional column containing the count of␣
↪images
```

(continues on next page)

(continued from previous page)

```
    containing cats.
    """
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
    model.eval()
    img_folder = df['images'][0]
    images = sorted(glob.glob(f"{img_folder}/*.jpg"))
    total_images = len(images)
    cats = 0
    start_index = partition_id * (total_images // desired_num_partitions)
    if partition_id == desired_num_partitions - 1: # Last partition must parse to end of␣
↪list
        images = images[start_index:]
    else:
        end_index = (partition_id + 1) * (total_images // desired_num_partitions)
        images = images[start_index:end_index]
    for img in images:
        cats = cats + 1 if contains_cat(Image.open(img), model) else cats
    df['cat_count'] = cats
    return df

def reduce_fn(dfs):
    """
    Coalesce the results of fanning out the `parallel_query` query.

    Parameters
    ----------
    dfs : a list of dataframes
        The resulting dataframes from fanning out `parallel_query`

    Returns
    -------
    A new dataframe whose `cat_count` column is the sum of the `cat_count` column of all
    dataframes in `dfs`
    """
    df = dfs[0]
    cat_count = df['cat_count'][0]
    for dataframe in dfs[1:]:
        cat_count += dataframe['cat_count'][0]
    df['cat_count'] = cat_count
    return df
pipeline = PandasQueryPipeline(df, desired_num_partitions)
pipeline.add_query(
    parallel_query,
    fan_out=True,
    reduce_fn=reduce_fn,
    is_output=True,
    pass_partition_id=True
)
parallel_start = time()
df_with_cat_count = pipeline.compute_batch()[0]
parallel_end = time()
print(f"Result of pipeline:\n{df_with_cat_count}")
```

(continues on next page)

(continued from previous page)

```
print(f"Total Time in Serial: {serial_end - serial_start}")
print(f"Total time with induced parallelism: {parallel_end - parallel_start}")
shutil.rmtree("images/") # Clean up
```

Batch Pipelining with Dynamic Repartitioning

Similarly, it is also possible to hint to the Pipeline API to repartition after a node completes computation. This is currently only supported if the input dataframe consists of only one partition. The number of partitions after repartitioning is controlled by the `num_partitions` parameter passed to the constructor of the `PandasQueryPipeline`.

The following example demonstrates how to use the `repartition_after` parameter.

```
import modin.pandas as pd
from modin.experimental.batch import PandasQueryPipeline
import numpy as np

small_df = pd.DataFrame([[1, 2, 3]]) # Create a small dataframe

def increase_dataframe_size(df):
    import pandas
    new_df = pandas.concat([df] * 1000)
    new_df = new_df.reset_index(drop=True) # Get a new range index that isn't duplicated
    return new_df

desired_num_partitions = 24 # We will repartition to 24 partitions

def add_partition_id_to_df(df, partition_id):
    import pandas
    new_col = pandas.Series([partition_id]*len(df), name="partition_id", index=df.index)
    return pandas.concat([df, new_col], axis=1)

pipeline = PandasQueryPipeline(small_df, desired_num_partitions)
pipeline.add_query(increase_dataframe_size, repartition_after=True)
pipeline.add_query(add_partition_id_to_df, pass_partition_id=True, is_output=True)
result_df = pipeline.compute_batch()[0]
print(f"Number of partitions passed to second query: " +
      f"{len(np.unique(result_df['partition_id'].values))}")
print(f"Result of pipeline:\n{result_df}")
```

10.3.7 Modin engines

As a rule, you don't have to worry about initialization of an execution engine as Modin itself automatically initializes one when performing the first operation. Also, Modin has a broad range of *configuration settings*, which you can use to configure an execution engine. If there is a reason to initialize an execution engine on your own and you are sure what to do, Modin will automatically attach to whichever engine is available. Below, you can find some examples on how to initialize a specific execution engine on your own.

Ray

You can initialize Ray engine with a specific number of CPUs (worker processes) to perform computation.

```
import ray
import modin.config as modin_cfg

ray.init(num_cpus=<N>)
modin_cfg.Engine.put("ray") # Modin will use Ray engine
modin_cfg.CpuCount.put(<N>)
```

To get more details on all possible parameters for initialization refer to Ray documentation.

Dask

You can initialize Dask engine with a specific number of worker processes and threads per worker to perform computation.

```
from distributed import Client
import modin.config as modin_cfg

client = Client(n_workers=<N1>, threads_per_worker=<N2>)
modin_cfg.Engine.put("dask") # # Modin will use Dask engine
modin_cfg.CpuCount.put(<N1>)
```

To get more details on all possible parameters for initialization refer to Dask Distributed documentation.

MPI through unidist

You can initialize MPI through unidist engine with a specific number of CPUs (worker processes) to perform computation.

```
import unidist
import unidist.config as unidist_cfg
import modin.config as modin_cfg

unidist_cfg.Backend.put("mpi")
unidist_cfg.CpuCount.put(<N>)
unidist.init()

modin_cfg.Engine.put("unidist") # # Modin will use MPI through unidist engine
modin_cfg.CpuCount.put(<N>)
```

To get more details on all possible parameters for initialization refer to unidist documentation.

Modin aims to not only optimize pandas, but also provide a comprehensive, integrated toolkit for data scientists. We are actively developing data science tools such as DataFrame spreadsheet integration, DataFrame algebra, progress bars, SQL queries on DataFrames, and more. Join us on Slack for the latest updates!

10.3.8 Modin engines

Modin supports a series of execution engines such as Ray, Dask, MPI through unidist, each of which might be a more beneficial choice for a specific scenario. When doing the first operation with Modin it automatically initializes one of the engines to further perform distributed/parallel computation. If you are familiar with a concrete execution engine, it is possible to initialize the engine on your own and Modin will automatically attach to it. Refer to *Modin engines* page for more details.

10.3.9 Additional APIs

Modin also supports these additional APIs on top of pandas to improve user experience.

- `to_pandas()` – convert a Modin DataFrame/Series to a pandas DataFrame/Series.
- `from_pandas()` – convert a pandas DataFrame to a Modin DataFrame.
- `to_ray()` – convert a Modin DataFrame/Series to a Ray Dataset.
- `from_ray()` – convert a Ray Dataset to a Modin DataFrame.
- `to_dask()` – convert a Modin DataFrame/Series to a Ray Dataset.
- `from_dask()` – convert a Modin DataFrame/Series to a Dask DataFrame/Series.
- `from_map()` – create a Modin DataFrame from map function applied to an iterable object.
- `from_arrow()` – convert an Arrow Table to a Modin DataFrame.
- `read_csv_glob()` – read multiple files in a directory.
- `read_sql()` – add optional parameters for the database connection.
- `read_custom_text()` – read custom text data from file.
- `read_pickle_glob()` – read multiple pickle files in a directory.
- `read_parquet_glob()` – read multiple parquet files in a directory.
- `read_json_glob()` – read multiple json files in a directory.
- `read_xml_glob()` – read multiple xml files in a directory.
- `to_pickle_glob()` – write to multiple pickle files in a directory.
- `to_parquet_glob()` – write to multiple parquet files in a directory.
- `to_json_glob()` – write to multiple json files in a directory.
- `to_xml_glob()` – write to multiple xml files in a directory.

10.3.10 DataFrame partitioning API

Modin DataFrame provides an API to directly access partitions: you can extract physical partitions from a `DataFrame`, modify their structure by reshuffling or applying some functions, and create a DataFrame from those modified partitions. Visit *pandas partitioning API* documentation to learn more.

10.3.11 Modin Spreadsheet API

The Spreadsheet API for Modin allows you to render the dataframe as a spreadsheet to easily explore your data and perform operations on a graphical user interface. The API also includes features for recording the changes made to the dataframe and exporting them as reproducible code. Built on top of Modin and SlickGrid, the spreadsheet interface is able to provide interactive response times even at a scale of billions of rows. See our Modin Spreadsheet API documentation for more details.

10.3.12 Progress Bar

Visual progress bar for Dataframe operations such as groupby and fillna, as well as for file reading operations such as read_csv. Built using the tqdm library and Ray execution engine. See Progress Bar documentation for more details.

In [6]:
```
df
```

Out[6]:

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	pickup_longitude	pickup_latitude	RateCodeID	store_and_fwc
0	2	2015-01-15 19:05:39	2015-01-15 19:23:42	1	1.59	-73.993896	40.750111	1	
1	1	2015-01-10 20:33:38	2015-01-10 20:53:28	1	3.30	-74.001648	40.724243	1	
2	1	2015-01-10 20:33:38	2015-01-10 20:43:41	1	1.80	-73.963341	40.802788	1	
3	1	2015-01-10 20:33:39	2015-01-10 20:35:31	1	0.50	-74.009087	40.713818	1	
4	1	2015-01-10 20:33:39	2015-01-10 20:52:58	1	3.00	-73.971176	40.762428	1	
...	...	...	...	...	...	...	...	...	
12748981	1	2015-01-10 19:01:44	2015-01-10 19:05:40	2	1.00	-73.951988	40.786217	1	
12748982	1	2015-01-10 19:01:44	2015-01-10 19:07:26	2	0.80	-73.982742	40.728184	1	
12748983	1	2015-01-10 19:01:44	2015-01-10 19:15:01	1	3.40	-73.979324	40.749550	1	
12748984	1	2015-01-10 19:01:44	2015-01-10 19:17:03	1	1.30	-73.999565	40.738483	1	
12748985	1	2015-01-10 19:01:45	2015-01-10 19:07:33	1	0.70	-73.960350	40.766399	1	

12748986 rows x 19 columns

In [7]:
```
df.groupby("passenger_count").count()
```

Estimated completion of line 1: 100% 12/12 [00:06<00:00, 6.64s/it]

Out[7]:

passenger_count	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	trip_distance	pickup_longitude	pickup_latitude	RateCodeID	store_and_fwd_flag	drop
0	6565	6565	6565	6565	6565	6565	6565	6565	
1	8993870	8993870	8993870	8993870	8993870	8993870	8993870	8993870	
2	1814594	1814594	1814594	1814594	1814594	1814594	1814594	1814594	
3	528486	528486	528486	528486	528486	528486	528486	528486	
4	253228	253228	253228	253228	253228	253228	253228	253228	
5	697645	697645	697645	697645	697645	697645	697645	697645	

10.3.13 Dataframe Algebra

A minimal set of operators that can be composed to express any dataframe query for use in query planning and optimization. See our paper for more information, and full documentation is coming soon!

10.3.14 Distributed XGBoost on Modin

Modin provides an implementation of distributed XGBoost machine learning algorithm on Modin DataFrames. See our *Distributed XGBoost on Modin documentation* for details about installation and usage, as well as Modin XGBoost architecture documentation for information about implementation and internal execution flow.

10.3.15 Logging with Modin

Modin logging offers users greater insight into their queries by logging internal Modin API calls, partition metadata, and system memory. Logging is disabled by default, but when it is enabled, log files are written to a local *.modin* directory at the same directory level as the notebook/script used to run Modin. See our *Logging with Modin documentation* for usage information.

10.3.16 Batch Pipeline API

Modin provides an experimental batched API that pipelines row parallel queries. See our *Batch Pipline API Usage Guide* for a walkthrough on how to use this feature, as well as Batch Pipeline API documentation for more information about the API.

10.3.17 Fuzzydata Testing

An experimental GitHub Action on pull request has been added to Modin, which automatically runs the Modin codebase against *fuzzydata*, a random dataframe workflow generator. The resulting workflow that was used to test Modin codebase can be downloaded as an artifact from the GitHub Actions tab for further inspection. See fuzzydata for more details.

10.4 Optimization Notes

Modin has chosen default values for a lot of the configurations here that provide excellent performance in most cases. This page is for those who love to optimize their code and those who are curious about existing optimizations within Modin. Here you can find more information about Modin's optimizations both for a pipeline of operations as well as for specific operations. If you want to go ahead and tune the Modin behavior on your own, refer to *Modin Configuration Settings* page for the full set of configurations available in Modin.

10.4.1 Range-partitioning in Modin

Modin utilizes a range-partitioning approach for specific operations, significantly enhancing parallelism and reducing memory consumption in certain scenarios. Range-partitioning is typically engaged for operations that has key columns (to group on, to merge on, etc).

You can enable range-partitioning by specifying `cfg.RangePartitioning` *configuration variable:*

```
import modin.pandas as pd
import modin.config as cfg

cfg.RangePartitioning.put(True) # past this point methods that support range-partitioning
                                # will use it

pd.DataFrame(...).groupby(...).mean() # use range-partitioning for groupby.mean()

cfg.Range-partitioning.put(False)

pd.DataFrame(...).groupby(...).mean() # use MapReduce implementation for groupby.mean()
```

Building range-partitioning assumes data reshuffling, which may result into breaking the original order of rows, for some operation, it will mean that the result will be different from Pandas.

Range-partitioning is not a silver bullet, meaning that enabling it is not always beneficial. Below you find a link to the list of operations that have support for range-partitioning and practical advices on when one should enable it: operations that support range-partitioning.

10.4.2 Understanding Modin's partitioning mechanism

Modin's partitioning is crucial for performance; so we recommend expert users to understand Modin's partitioning mechanism and how to tune it in order to achieve better performance.

How Modin partitions a dataframe

Modin uses a partitioning scheme that partitions a dataframe along both axes, resulting in a matrix of partitions. The row and column chunk sizes are computed independently based on the length of the appropriate axis and Modin's special *configuration variables* (`NPartitions` and `MinPartitionSize`):

- `NPartitions` is the maximum number of splits along an axis; by default, it equals to the number of cores on your local machine or cluster of nodes.
- `MinPartitionSize` is the minimum number of rows/columns to do a split. For instance, if `MinPartitionSize` is 32, the column axis will not be split unless the amount of columns is greater than 32. If it is is greater, for example, 34, then the column axis is sliced into two partitions: containing 32 and 2 columns accordingly.

Beware that `NPartitions` specifies a limit for the number of partitions *along a single axis*, which means, that the actual limit for the entire dataframe itself is the square of `NPartitions`.

Full-axis functions

Some of the aggregation functions require knowledge about the entire axis, for example at `.apply(foo, axis=0)` the passed function `foo` expects to receive data for the whole column at once.

When a full-axis function is applied, the partitions along this axis are collected at a single worker that processes the function. After the function is done, the partitioning of the data is back to normal.

Note that the amount of remote calls is equal to the number of partitions, which means that since the number of partitions is decreased for full-axis functions it also decreases the potential for parallelism.

Also note, that reduce functions such as `.sum()`, `.mean()`, `.max()`, etc, are not considered to be full-axis, so they do not suffer from the decreasing level of parallelism.

How to tune partitioning

Configure Modin's default partitioning scheme

As you can see from the examples above, the more the dataframe's shape is closer to a square, the closer the number of partitions to the square of `NPartitions`. In the case of `NPartitions` equals to the number of workers, that means that a single worker is going to process multiple partitions at once, which slows down overall performance.

If your workflow mainly operates with wide dataframes and non-full-axis functions, it makes sense to reduce the `NPartitions` value so a single worker would process a single partition.

Copy-pastable example, showing how tuning `NPartitions` value for wide frames may improve performance on your machine:

```
from multiprocessing import cpu_count
from modin.distributed.dataframe.pandas import unwrap_partitions
import modin.config as cfg
import modin.pandas as pd
import numpy as np
import timeit

# Generating data for a square-like dataframe
data = np.random.randint(0, 100, size=(5000, 5000))

# Explicitly setting `NPartitions` to its default value
cfg.NPartitions.put(cpu_count())

# Each worker processes `cpu_count()` amount of partitions
df = pd.DataFrame(data)
print(f"NPartitions: {cfg.NPartitions.get()}")
# Getting raw partitions to count them
partitions_shape = np.array(unwrap_partitions(df)).shape
print(
    f"The frame has {partitions_shape[0]}x{partitions_shape[1]}={np.prod(partitions_
↪shape)} partitions "
    f"when the CPU has only {cpu_count()} cores."
)
print(f"10 times of .abs(): {timeit.timeit(lambda: df.abs(), number=10)}s.")
```

(continues on next page)

(continued from previous page)

```
# Possible output:
#   NPartitions: 112
#   The frame has 112x112=12544 partitions when the CPU has only 112 cores.
#   10 times of .abs(): 23.64s.

# Taking a square root of the the current `cpu_count` to make more even partitioning
cfg.NPartitions.put(int(cpu_count() ** 0.5))

# Each worker processes a single partition
df = pd.DataFrame(data)
print(f"NPartitions: {cfg.NPartitions.get()}")
# Getting raw partitions to count them
partitions_shape = np.array(unwrap_partitions(df)).shape
print(
    f"The frame has {partitions_shape[0]}x{partitions_shape[1]}={np.prod(partitions_
↪shape)} "
    f"when the CPU has {cpu_count()} cores."
)
print(f"10 times of .abs(): {timeit.timeit(lambda: df.abs(), number=10)}s.")
# Possible output:
#   NPartitions: 10
#   The frame has 10x10=100 partitions when the CPU has 112 cores.
#   10 times of .abs(): 0.25s.
```

Manually trigger repartitioning

If you're getting unexpectedly poor performance, although you configured `MODIN_NPARTITIONS` correctly, then this might be caused by unbalanced partitioning that occurred during the workflow's execution.

Modin's idealogy is to handle partitioning internally and not let users worry about the possible consequences of applying a lot of "bad" operations that may affect DataFrame's partitioning. We're constantly making efforts to find and fix cases where partitioning may cause a headache for users.

However, if you feel that you're dealing with unbalanced partitioning you may try to call an internal `modin.pandas.dataframe.DataFrame._repartition()` method on your `DataFrame` in order to manually trigger partitions rebalancing and see whether it improves performance for your case.

`DataFrame.`**`_repartition`**`(`*axis: Optional[int] = None*`)` → Self

Repartitioning Modin objects to get ideal partitions inside.

Allows to improve performance where the query compiler can't improve yet by doing implicit repartitioning.

Parameters

axis (*{0, 1, None}, optional*) – The axis along which the repartitioning occurs. *None* is used for repartitioning along both axes.

Returns

The repartitioned dataframe or series, depending on the original type.

Return type

DataFrame or *Series*

An actual use-case for this method may be the following:

```
import modin.pandas as pd
import timeit

df = pd.DataFrame({"col0": [1, 2, 3, 4]})

# Appending a lot of columns may result into unbalanced partitioning
for i in range(1, 128):
    df[f"col{i}"] = pd.Series([1, 2, 3, 4])

print(
    "DataFrame with unbalanced partitioning:",
    timeit.timeit(lambda: df.sum(), number=10)
) # 1.44s

df = df._repartition()
print(
    "DataFrame after '._repartition()':",
    timeit.timeit(lambda: df.sum(), number=10)
) # 0.21s.
```

10.4.3 Avoid iterating over Modin DataFrame

Use `df.apply()` or other aggregation methods when possible instead of iterating over a dataframe. For-loops don't scale and forces the distributed data to be collected back at the driver.

Copy-pastable example, showing how replacing a for-loop to the equivalent `.apply()` may improve performance:

```
import modin.pandas as pd
import numpy as np
from timeit import default_timer as timer

data = np.random.randint(1, 100, (2 ** 10, 2 ** 2))

md_df = pd.DataFrame(data)

result = []
t1 = timer()
# Iterating over a dataframe forces to collect distributed data to the driver and doesn't␣
↪scale
for idx, row in md_df.iterrows():
    result.append((row[1] + row[2]) / row[3])
print(f"Filling a list by iterating a Modin frame: {timer() - t1:.2f}s.")
# Possible output: 36.15s.

t1 = timer()
# Using `.apply()` perfectly scales to all axis-partitions
result = md_df.apply(lambda row: (row[1] + row[2]) / row[3], axis=1).to_numpy().tolist()
print(f"Filling a list by using '.apply()' and converting the result to a list: {timer()␣
↪- t1:.2f}s.")
# Possible output: 0.22s.
```

10.4.4 Use Modin's Dataframe Algebra API to implement custom parallel functions

Modin provides a set of low-level parallel-implemented operators which can be used to build most of the aggregation functions. These operators are present in the algebra module. Modin DataFrame allows users to use their own aggregations built with this module. Visit the DataFrame's algebra page of the documentation for the steps to do it.

10.4.5 Avoid mixing pandas and Modin DataFrames

Although Modin is considered to be a drop-in replacement for pandas, Modin and pandas are not intended to be used together in a single flow. Passing a pandas DataFrame as an argument for a Modin's DataFrame method may either slowdown the function (because it has to process non-distributed object) or raise an error. You would also get an undefined behavior if you pass a Modin DataFrame as an input to pandas methods, since pandas identifies Modin's objects as a simple iterable, and so can't leverage its benefits as a distributed dataframe.

Copy-pastable example, showing how mixing pandas and Modin DataFrames in a single flow may bottleneck performance:

```
import modin.pandas as pd
import numpy as np
import timeit
import pandas

data = np.random.randint(0, 100, (2 ** 20, 2 ** 2))

md_df, md_df_copy = pd.DataFrame(data), pd.DataFrame(data)
pd_df, pd_df_copy = pandas.DataFrame(data), pandas.DataFrame(data)

print("concat modin frame + pandas frame:")
# Concatenating modin frame + pandas frame using modin '.concat()'
# This case is bad because Modin have to process non-distributed pandas object
time = timeit.timeit(lambda: pd.concat([md_df, pd_df]), number=10)
print(f"\t{time}s.\n")
# Possible output: 0.44s.

print("concat modin frame + modin frame:")
# Concatenating modin frame + modin frame using modin '.concat()'
# This is an ideal case, Modin is being used as intended
time = timeit.timeit(lambda: pd.concat([md_df, md_df_copy]), number=10)
print(f"\t{time}s.\n")
# Possible output: 0.05s.

print("concat pandas frame + pandas frame:")
# Concatenating pandas frame + pandas frame using pandas '.concat()'
time = timeit.timeit(lambda: pandas.concat([pd_df, pd_df_copy]), number=10)
print(f"\t{time}s.\n")
# Possible output: 0.31s.

print("concat pandas frame + modin frame:")
# Concatenating pandas frame + modin frame using pandas '.concat()'
time = timeit.timeit(lambda: pandas.concat([pd_df, md_df]), number=10)
print(f"\t{time}s.\n")
# Possible output: TypeError
```

10.4.6 Operation-specific optimizations

merge

`merge` operation in Modin uses the broadcast join algorithm: combining a right Modin DataFrame into a pandas DataFrame and broadcasting it to the row partitions of the left Modin DataFrame. In order to minimize interprocess communication cost when doing an inner join you may want to swap left and right DataFrames.

```
import modin.pandas as pd
import numpy as np

left_data = np.random.randint(0, 100, size=(2**8, 2**8))
right_data = np.random.randint(0, 100, size=(2**12, 2**12))

left_df = pd.DataFrame(left_data)
right_df = pd.DataFrame(right_data)
%timeit left_df.merge(right_df, how="inner", on=10)
3.59 s  107 ms per loop (mean  std. dev. of 7 runs, 1 loop each)

%timeit right_df.merge(left_df, how="inner", on=10)
1.22 s  40.1 ms per loop (mean  std. dev. of 7 runs, 1 loop each)
```

Note that result columns order may differ for first and second `merge`.

10.5 Benchmarking Modin

10.5.1 Summary

To benchmark a single Modin function, often turning on the *configuration variable* variable `BenchmarkMode` will suffice.

There is no simple way to benchmark more complex Modin workflows, though benchmark mode or calling `modin.utils.execute` on Modin objects may be useful. The *Modin logs* may help you identify bottlenecks in your code, and they may also help profile the execution of each Modin function.

10.5.2 Modin's execution and benchmark mode

Most of Modin's execution happens asynchronously, i.e. in separate processes that run independently of the main program flow. Some execution is also lazy, meaning that it doesn't start immediately once the user calls a Modin function. While Modin provides the same API as pandas, lazy and asynchronous execution can often make it hard to tell how much time each Modin function call takes, as well as to compare Modin's performance to pandas and other similar libraries.

Note: All examples in this doc use the system specified at the bottom of this page.

Consider the following ipython script:

```
import modin.pandas as pd
from modin.config import MinPartitionSize
import time
```

(continues on next page)

(continued from previous page)

```
import ray

# Look at the Ray documentation with respect to the Ray configuration suited to you most.
ray.init()
df = pd.DataFrame(list(range(MinPartitionSize.get() * 2)))
%time result = df.map(lambda x: time.sleep(0.1) or x)
%time print(result)
```

Modin takes just 2.68 milliseconds for the `map`, and 3.78 seconds to print the result. However, if we run this script in pandas by replacing `import modin.pandas as pd` with `import pandas as pd`, the `map` takes 6.63 seconds, and printing the result takes just 5.53 milliseconds.

Both pandas and Modin start executing the `map` as soon as the interpreter evalutes it. While pandas blocks until the `map` has finished, Modin just kicks off asynchronous functions in remote ray processes. Printing the function result is fairly fast in pandas and Modin, but before Modin can print the data, it has to wait until all the remote functions complete.

To time how long Modin takes for a single operation, you should typically use benchmark mode. Benchmark mode will wait for all asynchronous remote execution to complete. You can turn on benchmark mode on at any point as follows:

```
from modin.config import BenchmarkMode
BenchmarkMode.put(True)
```

Rerunning the script above with benchmark mode on, the Modin `map` takes 3.59 seconds, and the `print` takes 183 milliseconds. These timings better reflect where Modin is spending its execution time.

10.5.3 A caveat about benchmark mode

While benchmark code is often good for measuring the performance of a single Modin function call, it can underestimate Modin's performance in cases where Modin's asynchronous execution improves Modin's performance. Consider the following script with benchmark mode on:

```
import numpy as np
import time
import ray
from io import BytesIO

import modin.pandas as pd
from modin.config import BenchmarkMode, MinPartitionSize

BenchmarkMode.put(True)

start = time.time()
df = pd.DataFrame(list(range(MinPartitionSize.get())), columns=['A'])
result1 = df.map(lambda x: time.sleep(0.2) or x + 1)
result2 = df.map(lambda x: time.sleep(0.2) or x + 2)
result1.to_parquet(BytesIO())
result2.to_parquet(BytesIO())
end = time.time()
print(f'map and write to parquet took {end - start} seconds.')
```

The script does two slow `map` on a dataframe and then writes each result to a buffer. The whole script takes 13 seconds with benchmark mode on, but just 7 seconds with benchmark mode off. Because Modin can run the `map` asynchronously, it can start writing the first result to its buffer while it's still computing the second result. With benchmark mode on, Modin has to execute every function synchronously instead.

10.5.4 How to benchmark complex workflows

Typically, to benchmark Modin's overall performance on your workflow, you should start by looking at end-to-end performance with benchmark mode off. It's common for Modin worfklows to end with writing results to one or more files, or with printing some Modin objects to an interactive console. Such end points are natural ways to make sure that all of the Modin execution that you require is complete.

To measure more fine-grained performance, it can be helpful to turn benchmark mode on, but beware that doing so may reduce your script's overall performance and thus may not reflect where Modin is normally spending execution time, as pointed out above.

Turning on *Modin logging* and using the Modin logs can also help you profile your workflow. The Modin logs can also give a detailed break down of the performance of each Modin function at each Modin *layer*. Log mode is more useful when used in conjuction with benchmark mode.

Sometimes, if you don't have a natural end-point to your workflow, you can just call `modin.utils.execute` on the workflow's final Modin objects. That will typically block on any asynchronous computation:

```
import time
import ray
from io import BytesIO

import modin.pandas as pd
from modin.config import MinPartitionSize, NPartitions
import modin.utils

MinPartitionSize.put(32)
NPartitions.put(16)

def slow_add_one(x):
  if x == 5000:
    time.sleep(10)
  return x + 1

# Look at the Ray documentation with respect to the Ray configuration suited to you most.
ray.init()
df1 = pd.DataFrame(list(range(10_000)), columns=['A'])
result = df1.map(slow_add_one)
# %time modin.utils.execute(result)
%time result.to_parquet(BytesIO())
```

Writing the result to a buffer takes 9.84 seconds. However, if you uncomment the `%time modin.utils.execute(result)` before the `to_parquet` call, the `to_parquet` takes just 23.8 milliseconds!

Note: If you see any Modin documentation touting Modin's speed without using benchmark mode or otherwise guaranteeing that Modin is finishing all asynchronous and deferred computation, you should file an issue on the Modin GitHub. It's not fair to compare the speed of an async Modin function call to an equivalent synchronous call using another library.

10.5.5 Appendix: System Information

The example scripts here were run on the following system:

- **OS Platform and Distribution (e.g., Linux Ubuntu 16.04)**: macOS Monterey 12.4
- **Modin version**: d6d503ac7c3028d871c34d9e99e925ddb0746df6
- **Ray version**: 2.0.0
- **Python version**: 3.10.4
- **Machine**: MacBook Pro (16-inch, 2019)
- **Processor**: 2.3 GHz 8-core Intel Core i9 processor
- **Memory**: 16 GB 2667 MHz DDR4

10.6 Third Party Library Integrations

Modin is a drop-in replacement for Pandas, so we want it to interoperate with third-party libraries just as Pandas does. To see where Modin performs well and where it needs to improve, we've selected a number of important machine learning + visualization + statistics libraries, and then looked at examples (from their documentation, if possible) about how they work with Pandas. Then we ran those same workflows with Modin, and tracked what worked, and what failed.

In the table below, you'll see, for each third-party library we tested, the number of successful test calls / total test calls, and a qualitative description of how both Pandas and Modin integrate with that library.

In the deeper dive, you can view the Jupyter notebook we have used to test API calls and the corresponding Github issues filed. If you come across other issues/ examples in your own workflows we encourage you to file an issue or contribute a PR!

Note: These interoperability metrics are preliminary and not all APIs for each library have been tested. Feel free to add more!

10.6.1 Modin Interoperability by Library

Library	API successes / calls	Interoperability
seaborn	73% (11/15)	**Pandas**: Accepts Pandas DataFrames as inputs for producing plot **Modin**: Mostly accepts Modin DataFrames as inputs for producing plots, but fails completely in some cases (pairplot, lmplot), and in others (catplot, objects.Plot) only works for some parameter combinations
plotly	78% (7 / 9)	**Pandas**: Accepts Pandas DataFrames as inputs for producing plots, including specifying X and Y parameters as df columns **Modin**: Mostly accepts Modin DataFrames as inputs for producing plots (the exception is choropleth), but fails when specifying X and Y parameters as df columns
matplotlib	100% (5 / 5)	**Pandas**: Accepts Pandas DataFrames as inputs for producing plots like scatter, barh, etc. **Modin**: Accepts Modin DataFrames as inputs for producing plots like scatter, barh, etc.
altair	0% (0 / 1)	**Pandas**: Accepts Pandas DataFrames as inputs for producing charts through Chart **Modin**: Does not accept Modin DataFrames as inputs for producing charts through Chart
bokeh	0% (0 / 1)	**Pandas**: Loads Pandas DataFrames through ColumnDataSource **Modin**: Does not load Modin DataFrames through ColumnDataSource
sklearn	100% (6 / 6)	**Pandas**: Many functions take Pandas DataFrames as inputs **Modin**: Many functions take Modin DataFrames as inputs
Hugging Face (Transformers, Datasets)	100% (2 / 2)	**Pandas**: Loads Pandas DataFrames into Datasets, and processes Pandas DataFrame rows as inputs using Transformers.InputExample (deprecated) **Modin**: Loads Modin DataFrames into Datasets (though slowly), and processes Modin DataFrame rows as inputs through Transformers.InputExample (deprecated)
Tensorflow	75% (3 / 4)	**Pandas**: Converts Pandas dataframes to tensors **Modin**: Converts Modin DataFrames to tensors, but specialized APIs like Keras might not work yet
NLTK	100% (1 / 1)	**Pandas**: Performs transformations like tokenization on Pandas DataFrames **Modin**: Performs transformations like tokenization on Modin DataFrames
XGBoost	100% (1 / 1)	**Pandas**: Loads Pandas DataFrames through the DMatrix function **Modin**: Loads Modin DataFrames through the DMatrix function
statsmodels	50% (1 / 2)	**Pandas**: Can accept Pandas DataFrames when fitting models **Modin**: Sometimes accepts Modin DataFrames when fitting models (e.g., formula.api.ols), but does not in others (e.g., api.OLS)

10.6.2 A Deeper Dive

seaborn

Jupyter Notebook

Github Issues

- https://github.com/modin-project/modin/issues/5435
- https://github.com/modin-project/modin/issues/5433

plotly

Jupyter Notebook

Github Issues

- https://github.com/modin-project/modin/issues/5447
- https://github.com/modin-project/modin/issues/5445

matplotlib

Jupyter Notebook

altair

Jupyter Notebook

Github Issues

- https://github.com/modin-project/modin/issues/5438

bokeh

Jupyter Notebook

Github Issues

- https://github.com/modin-project/modin/issues/5437

sklearn

Jupyter Notebook

Hugging Face

Jupyter Notebook

Tensorflow

Jupyter Notebook

Github Issues

- https://github.com/modin-project/modin/issues/5439

NLTK

Jupyter Notebook

XGBoost

Jupyter Notebook

statsmodels

Jupyter Notebook

Github Issues

- https://github.com/modin-project/modin/issues/5440

10.6.3 Appendix: System Information

The example scripts here were run on the following system:

- **OS Platform and Distribution (e.g., Linux Ubuntu 16.04)**: macOS Big Sur 11.5.2
- **Modin version**: 0.18.0+3.g4114183f
- **Ray version**: 2.0.1
- **Python version**: 3.9.7.final.0
- **Machine**: MacBook Pro (16-inch, 2019)
- **Processor**: 2.3 GHz 8-core Intel Core i9 processor
- **Memory**: 16 GB 2667 MHz DDR4

CHAPTER

ELEVEN

SUPPORTED APIS

For your convenience, we have compiled a list of currently implemented APIs and methods available in Modin. This documentation is updated as new methods and APIs are merged into the main branch, and not necessarily correct as of the most recent release.

To view the docs for the most recent release, check that you're viewing the stable version of the docs.

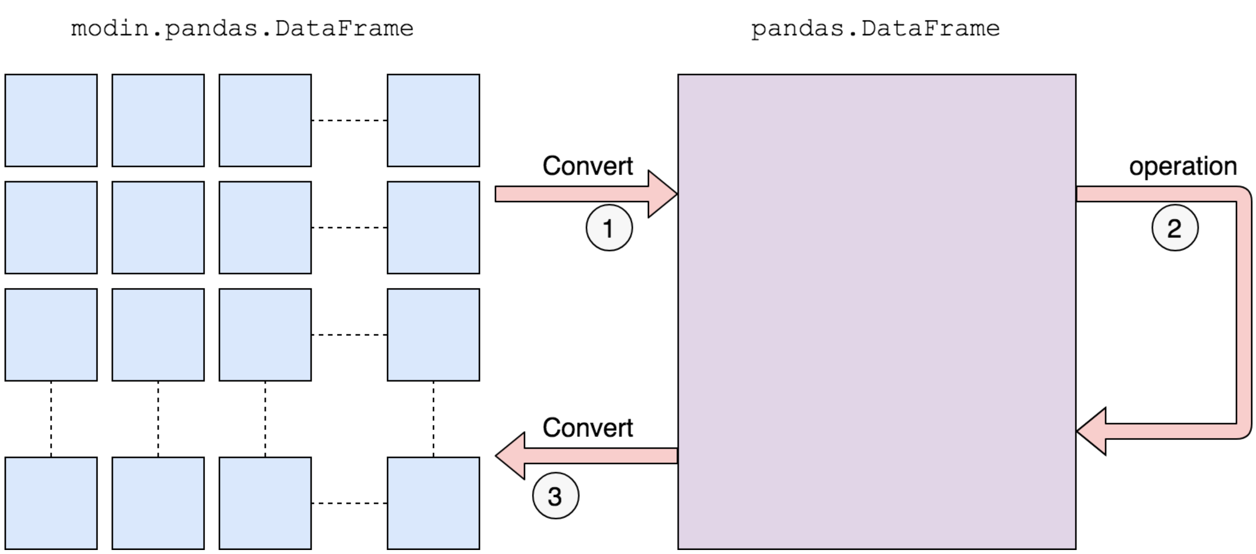
In order to install the latest version of Modin, follow the directions found on the *installation page*.

11.1 Questions on implementation details

If you have a question about the implementation details or would like more information about an API or method in Modin, please contact the Modin developer mailing list.

11.1.1 Defaulting to pandas

Currently Modin does not support distributed execution for all methods from pandas API. The remaining unimplemented methods are being executed in a mode called "default to pandas". This allows users to continue using Modin even though their workloads contain functions not yet implemented in Modin. Here is a diagram of how we convert to pandas and perform the operation:



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 pan-**

das. 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.

The exact methods we have implemented are listed in the respective subsections:

- *DataFrame*
- *Series*
- *utilities*
- *I/O*

We have taken a community-driven approach to implementing new methods. We did a study on pandas usage to learn what the most-used APIs are. Modin currently supports **93%** of the pandas API based on our study of pandas usage, and we are actively expanding the API. **To request implementation, file an issue at https://github.com/modin-project/modin/issues or send an email to feature_requests@modin.org.**

11.1.2 `pd.DataFrame` supported APIs

The following table lists both implemented and not implemented methods. If you have need of an operation that is listed as not implemented, feel free to open an issue on the GitHub repository, or give a thumbs up to already created issues. Contributions are also welcome!

The following table is structured as follows: The first column contains the method name. The second column contains link to a description of corresponding pandas method. The third column is a flag for whether or not there is an implementation in Modin for the method in the left column. `Y` stands for yes, `N` stands for no, `P` stands for partial (meaning some parameters may not be supported yet), and `D` stands for default to pandas.

DataFrame method	pandas Doc link	Implemented? (Y/N/P/D)	Notes for Current implementation
`T`	T	Y	
`abs`	abs	Y	
`add`	add	Y	**Ray** and **Dask**: Shuffles data in operations between DataFrames.
`add_prefix`	add_prefix	Y	
`add_suffix`	add_suffix	Y	
`agg` / `aggregate`	agg / aggregate	P	• Dictionary `func` parameter defaults to pandas • Numpy operations default to pandas
`align`	align	D	
`all`	all	Y	
`any`	any	Y	
`apply`	apply	Y	See `agg`
`applymap`	applymap	Y	
`asfreq`	asfreq	D	
`asof`	asof	Y	
`assign`	assign	Y	
`astype`	astype	Y	
`at`	at	Y	

continues on next page

Table 1 – continued from previous page

at_time	at_time	Y	
axes	axes	Y	
between_time	between_time	Y	
bfill	bfill	Y	
bool	bool	Y	
boxplot	boxplot	D	
clip	clip	Y	
combine	combine	Y	
combine_first	combine_first	Y	
compare	compare	Y	
copy	copy	Y	
corr	corr	P	Correlation floating point precision may slightly differ from pandas. For now pearson method is available only. For other methods and for numeric_only defaults to pandas.
corrwith	corrwith	D	
count	count	Y	
cov	cov	P	Covariance floating point precision may slightly differ from pandas. For numeric_only defaults to pandas.
cummax	cummax	Y	
cummin	cummin	Y	
cumprod	cumprod	Y	
cumsum	cumsum	Y	
describe	describe	Y	
diff	diff	Y	
div	div	Y	See add
divide	divide	Y	See add
dot	dot	Y	
drop	drop	Y	
droplevel	droplevel	Y	
drop_duplicates	drop_duplicates	D	
dropna	dropna	Y	
dtypes	dtypes	Y	
duplicated	duplicated	Y	
empty	empty	Y	
eq	eq	Y	See add
equals	equals	Y	Requires shuffle, can be further optimized
eval	eval	Y	
ewm	ewm	D	
expanding	expanding	D	
explode	explode	Y	
ffill	ffill	Y	

continues on next page

Table 1 – continued from previous page

fillna	fillna	P	value parameter of type DataFrame defaults to pandas.
filter	filter	Y	
first	first	Y	
first_valid_index	first_valid_index	Y	
floordiv	floordiv	Y	See add
from_dict	from_dict	D	
from_records	from_records	D	
ge	ge	Y	See add
get	get	Y	
groupby	groupby	Y	Not yet optimized for all operations.
gt	gt	Y	See add
head	head	Y	
hist	hist	D	
iat	iat	Y	
idxmax	idxmax	Y	
idxmin	idxmin	Y	
iloc	iloc	Y	
infer_objects	infer_objects	Y	
info	info	Y	
insert	insert	Y	
interpolate	interpolate	D	
isetitem	isetitem	D	
isin	isin	Y	
isna	isna	Y	
isnull	isnull	Y	
items	items	Y	
iterrows	iterrows	P	Modin does not parallelize iteration in Python
itertuples	itertuples	P	Modin does not parallelize iteration in Python
join	join	P	When on is set to right or outer or when validate is given defaults to pandas
keys	keys	Y	
kurt	kurt	Y	
kurtosis	kurtosis	Y	
last	last	Y	
last_valid_index	last_valid_index	Y	
le	le	Y	See add
loc	loc	P	We do not support: boolean array, callable.
lt	lt	Y	See add
mask	mask	D	
max	max	Y	
mean	mean	P	Modin defaults to pandas if given the level param.
median	median	P	Modin defaults to pandas if given the level param.

continues on next page

Table 1 – continued from previous page

melt	melt	Y	
memory_usage	memory_usage	Y	
merge	merge	P	Implemented the following cases: left_index=True and right_index=True, how=left and how=inner for all values of parameters except left_index=True and right_index=False or left_index=False and right_index=True. Defaults to pandas otherwise.
min	min	Y	
mod	mod	Y	See add
mode	mode	Y	
mul	mul	Y	See add
multiply	multiply	Y	See add
ndim	ndim	Y	
ne	ne	Y	See add
nlargest	nlargest	Y	
notna	notna	Y	
notnull	notnull	Y	
nsmallest	nsmallest	Y	
nunique	nunique	Y	
pct_change	pct_change	D	
pipe	pipe	Y	
pivot	pivot	Y	
pivot_table	pivot_table	Y	
plot	plot	D	
pop	pop	Y	
pow	pow	Y	See add
prod	prod	Y	
product	product	Y	
quantile	quantile	Y	
query	query	P	Local variables not yet supported
radd	radd	Y	See add
rank	rank	Y	
rdiv	rdiv	Y	See add
reindex	reindex	Y	Shuffles data
reindex_like	reindex_like	D	
rename	rename	Y	
rename_axis	rename_axis	Y	
reorder_levels	reorder_levels	Y	
replace	replace	Y	
resample	resample	Y	

continues on next page

Table 1 – continued from previous page

reset_index	reset_index	P	**Ray** and **Dask**: D when names or allow_duplicates is non-default
rfloordiv	rfloordiv	Y	See add
rmod	rmod	Y	See add
rmul	rmul	Y	See add
rolling	rolling	Y	
round	round	Y	
rpow	rpow	Y	See add
rsub	rsub	Y	See add
rtruediv	rtruediv	Y	See add
sample	sample	Y	
select_dtypes	select_dtypes	Y	
sem	sem	P	Modin defaults to pandas if given the level param.
set_axis	set_axis	Y	
set_index	set_index	Y	
shape	shape	Y	
shift	shift	Y	
size	size	Y	
skew	skew	P	Modin defaults to pandas if given the level param
sort_index	sort_index	Y	
sort_values	sort_values	Y	Shuffles data. Order of indexes that have the same sort key is not guaranteed to be the same across sorts
sparse	sparse	N	
squeeze	squeeze	Y	
stack	stack	Y	
std	std	P	Modin defaults to pandas if given the level param.
style	style	D	
sub	sub	Y	See add
subtract	subtract	Y	See add
sum	sum	Y	
swapaxes	swapaxes	Y	
swaplevel	swaplevel	Y	
tail	tail	Y	
take	take	Y	
to_clipboard	to_clipboard	D	
to_csv	to_csv	Y	
to_dict	to_dict	D	
to_excel	to_excel	D	
to_feather	to_feather	D	
to_gbq	to_gbq	D	
to_hdf	to_hdf	D	
to_html	to_html	D	

continues on next page

Table 1 – continued from previous page

to_json	to_json	D	Experimental implementation: DataFrame.modin.to_json_glob
to_xml	to_xml	D	Experimental implementation: DataFrame.modin.to_xml_glob
to_latex	to_latex	D	
to_orc	to_orc	D	
to_parquet	to_parquet	P	**Ray/Dask/Unidist**: Parallel implementation only if path parameter is a string. In that case, the `path` parameter specifies a directory where one file is written per row partition of the Modin dataframe. Experimental implementation: DataFrame.modin.to_parquet_glob
to_period	to_period	D	
to_pickle	to_pickle	D	Experimental implementation: DataFrame.modin.to_pickle_glob
to_records	to_records	D	
to_sql	to_sql	Y	
to_stata	to_stata	D	
to_string	to_string	D	
to_timestamp	to_timestamp	D	
to_xarray	to_xarray	D	
transform	transform	Y	
transpose	transpose	Y	
truediv	truediv	Y	See `add`
truncate	truncate	Y	
tz_convert	tz_convert	Y	
tz_localize	tz_localize	Y	
unstack	unstack	Y	
update	update	Y	
values	values	Y	
value_counts	value_counts	D	
var	var	P	Modin defaults to pandas if given the `level` param.
where	where	Y	

11.1.3 pd.Series supported APIs

The following table lists both implemented and not implemented methods. If you have need of an operation that is listed as not implemented, feel free to open an issue on the GitHub repository, or give a thumbs up to already created issues. Contributions are also welcome!

The following table is structured as follows: The first column contains the method name. The second column is a flag for whether or not there is an implementation in Modin for the method in the left column. Y stands for yes, N stands for no, P stands for partial (meaning some parameters may not be supported yet), and D stands for default to pandas. To learn more about the implementations that default to pandas, see the related section on *Defaulting to pandas*.

Series method	Modin Implementation? (Y/N/P/D)	Notes for Current implementation
abs	Y	
add	Y	
add_prefix	Y	
add_suffix	Y	
agg	Y	
aggregate	Y	
align	D	
all	Y	
any	Y	
apply	Y	
argmax	Y	
argmin	Y	
argsort	D	
array	D	
asfreq	D	
asobject	D	
asof	Y	
astype	Y	
at	Y	
at_time	Y	
autocorr	Y	
axes	Y	
base	D	
between	D	
between_time	Y	
bfill	Y	
bool	Y	
cat	D	
clip	Y	
combine	Y	
combine_first	Y	
compare	Y	
compress	D	
copy	Y	
corr	Y	Correlation floating point precision may slightly differ from pa
count	Y	
cov	Y	Covariance floating point precision may slightly differ from pa
cummax	Y	
cummin	Y	
cumprod	Y	

Table 2 – continued from previous page

cumsum	Y	
data	D	
describe	Y	
diff	Y	
div	Y	See add
divide	Y	See add
divmod	Y	
dot	Y	
drop	Y	
drop_duplicates	Y	
droplevel	Y	
dropna	Y	
dt	Y	
dtype	Y	
dtypes	Y	
duplicated	Y	
empty	Y	
eq	Y	See add
equals	Y	
ewm	D	
expanding	D	
explode	Y	
factorize	D	
ffill	Y	
fillna	Y	
filter	Y	
first	Y	
first_valid_index	Y	
flags	D	
floordiv	Y	See add
from_array	D	
ftype	Y	
ge	Y	See add
get	Y	
get_dtype_counts	Y	
get_ftype_counts	Y	
get_value	D	
get_values	D	
groupby	D	
gt	Y	See add
hasnans	Y	
head	Y	
hist	D	
iat	Y	
idxmax	Y	
idxmin	Y	
iloc	Y	
imag	D	
index	Y	
infer_objects	Y	

Table 2 – continued from previous page

interpolate	D	
is_monotonic_decreasing	Y	
is_monotonic_increasing	Y	
is_unique	Y	
isin	Y	
isna	Y	
isnull	Y	
item	Y	
items	Y	
itemsize	D	
keys	Y	
kurt	Y	
kurtosis	Y	
last	Y	
last_valid_index	Y	
le	Y	See add
loc	Y	
lt	Y	See add
map	Y	
mask	D	
max	Y	
mean	P	Modin defaults to pandas if given the level param.
median	P	Modin defaults to pandas if given the level param.
memory_usage	Y	
min	Y	
mod	Y	See add
mode	Y	
mul	Y	See add
multiply	Y	See add
name	Y	
nbytes	D	
ndim	Y	
ne	Y	See add
nlargest	Y	
nonzero	Y	
notna	Y	
notnull	Y	
nsmallest	Y	
nunique	Y	
pct_change	D	
pipe	Y	
plot	D	
pop	Y	
pow	Y	See add
prod	Y	
product	Y	
ptp	D	
put	D	
quantile	Y	
radd	Y	See add

Table 2 – continued from previous page

rank	Y	
ravel	Y	
rdiv	Y	See add
rdivmod	Y	
real	D	
reindex	Y	
reindex_like	Y	
rename	Y	
rename_axis	Y	
reorder_levels	D	
repeat	D	
replace	Y	
resample	Y	
reset_index	P	**Ray** and **Dask**: D when names or allow_duplicates is non-
rfloordiv	Y	See add
rmod	Y	See add
rmul	Y	See add
rolling	Y	
round	Y	
rpow	Y	See add
rsub	Y	See add
rtruediv	Y	See add
sample	Y	
searchsorted	Y	
sem	P	Modin defaults to pandas if given the level param.
set_axis	Y	
set_value	D	
shape	Y	
shift	Y	
size	Y	
skew	P	Modin defaults to pandas if given the level param.
sort_index	Y	
sort_values	Y	Order of indexes that have the same sort key is not guaranteed
sparse	Y	
squeeze	Y	
std	P	Modin defaults to pandas if given the level param.
str	Y	
strides	D	
sub	Y	See add
subtract	Y	See add;
sum	Y	
swapaxes	Y	
swaplevel	Y	
tail	Y	
take	Y	
to_clipboard	D	
to_csv	D	
to_dict	D	
to_excel	D	
to_frame	Y	

Table 2 – continued from previous page

to_hdf	D	
to_json	D	
to_latex	D	
to_list	D	
to_numpy	D	
to_period	D	
to_pickle	D	
to_sql	Y	
to_string	D	
to_timestamp	D	
to_xarray	D	
tolist	D	
transform	Y	
transpose	Y	
truediv	Y	See add
truncate	Y	
tz_convert	Y	
tz_localize	Y	
unique	Y	
unstack	Y	
update	Y	
valid	D	
value_counts	Y	The indices order of resulting object may differ from pandas.
values	Y	
var	P	Modin defaults to pandas if given the `level` param.
view	D	
where	Y	

11.1.4 pandas Utilities Supported

If you run `import modin.pandas as pd`, the following operations are available from `pd.<op>`, e.g. `pd.concat`. If you do not see an operation that pandas enables and would like to request it, feel free to open an issue. Make sure you tell us your primary use-case so we can make it happen faster!

The following table is structured as follows: The first column contains the method name. The second column is a flag for whether or not there is an implementation in Modin for the method in the left column. `Y` stands for yes, `N` stands for no, `P` stands for partial (meaning some parameters may not be supported yet), and `D` stands for default to pandas.

Utility method	Modin Implementation? (Y/N/P/D)	Notes for Current implementation
pd.concat	Y	
pd.eval	Y	
pd.unique	Y	
`pd.value_counts`	Y	The indices order of resulting object may differ from pandas.
pd.cut	D	
pd.to_numeric	D	
pd.factorize	D	
pd.from_dummies	D	
pd.qcut	D	
`pd.match`	D	
pd.to_datetime	D	
pd.get_dummies	Y	
pd.date_range	D	
pd.bdate_range	D	
pd.to_timedelta	D	
`pd.options`	Y	

Other objects & structures

This list is a list of objects not currently distributed by Modin. All of these objects are compatible with the distributed components of Modin. If you are interested in contributing a distributed version of any of these objects, feel free to open a pull request.

- Panel
- Index
- MultiIndex
- CategoricalIndex
- DatetimeIndex
- Timedelta
- Timestamp
- NaT
- PeriodIndex
- Categorical
- Interval
- UInt8Dtype
- UInt16Dtype
- UInt32Dtype
- UInt64Dtype
- SparseDtype
- Int8Dtype
- Int16Dtype

- Int32Dtype
- Int64Dtype
- CategoricalDtype
- DatetimeTZDtype
- IntervalDtype
- PeriodDtype
- RangeIndex
- TimedeltaIndex
- IntervalIndex
- IndexSlice
- TimeGrouper
- Grouper
- array
- Period
- DateOffset
- ExcelWriter
- SparseArray

11.1.5 `pd.read_<file>` and I/O APIs

A number of IO methods default to pandas. We have parallelized `read_csv`, `read_parquet` and some more (see table), though many of the remaining methods can be relatively easily parallelized. Some of the operations default to the pandas implementation, meaning it will read in serially as a single, non-distributed DataFrame and distribute it. Performance will be affected by this.

The following table is structured as follows: The first column contains the method name. The second column is a flag for whether or not there is an implementation in Modin for the method in the left column. `Y` stands for yes, `N` stands for no, `P` stands for partial (meaning some parameters may not be supported yet), and `D` stands for default to pandas.

Note: Support for fully asynchronous reading has been added for the following functions: `read_csv`, `read_fwf`, `read_table`, `read_custom_text`. This mode is disabled by default, one can enable it using `MODIN_ASYNC_READ_MODE=True` environment variable. Some parameter combinations are not supported and the function will be executed in synchronous mode.

IO method	Modin Implementation? (Y/N/P/D)	Notes for Current implementation
read_csv	Y	
read_fwf	Y	
read_table	Y	
read_parquet	P	Parameters besides `filters` and `storage_options` passed via `**kwargs` are not supported. `use_nullable_dtypes` == True is not supported. Experimental implementation: read_parquet_glob
read_json	P	Implemented for `lines=True` Experimental implementation: read_json_glob
read_xml	D	Experimental implementation: read_xml_glob
read_html	D	
read_clipboard	D	
read_excel	D	
read_hdf	D	
read_feather	Y	
read_stata	D	
read_sas	D	
read_pickle	D	Experimental implementation: read_pickle_glob
read_sql	Y	

11.1.6 Pandas backwards compatibility mode

Modin verions 0.16 and 0.17, but no later minor versions, had limited support for running with legacy pandas versions. The latest version of Modin no longer has such support.

Motivation for compatibility mode

Modin aims to keep compatibility with latest pandas release, hopefully catching up each release within a few days.

However, due to certain restrictions like need to use Python 3.6 it forces some users to use older pandas (1.1.x for Python 3.6, specifically), which normally would mean they're bound to be using ancient Modin as well.

To overcome this, Modin has special "compatibility mode" where some basic functionality works, but please note that the support is "best possible effort" (e.g. not all older bugs are worth fixing).

Known issues with pandas 1.1.x

- `pd.append()` does not preserve the order of columns in older pandas while Modin does
- `.astype()` produces different error type on incompatible dtypes
- `read_csv()` does not support reading from ZIP file *with compression* in parallel mode
- `read_*` do not support `storage_option` named argument
- `to_csv()` does not support binary mode for output file
- `read_excel()` does not support `.xlsx` files
- `read_fwf()` has a bug with list of skiprows and non-None nrows: pandas-dev#10261
- `.agg(int-value)` produces TypeError in older pandas but Modin raises AssertionError
- `Series.reset_index(drop=True)` does not ignore `name` in older pandas while Modin ignores it

- `.sort_index(ascending=None)` does not raise ValueError in older pandas while Modin raises it

Please keep in mind that there are probably more issues which are not yet uncovered!

CHAPTER

TWELVE

DEVELOPMENT

12.1 Contributing

12.1.1 Getting Started

If you're interested in getting involved in the development of Modin, but aren't sure where start, take a look at the issues tagged Good first issue or Documentation. These are issues that would be good for getting familiar with the codebase and better understanding some of the more complex components of the architecture. There is documentation here about the *architecture* that you will want to review in order to get started.

Also, feel free to join the discussions on the developer mailing list.

If you want a quick guide to getting your development environment setup, please use the contributing instructions on GitHub.

12.1.2 Certificate of Origin

To keep a clear track of who did what, we use a *sign-off* procedure (same requirements for using the signed-off-by process as the Linux kernel has https://www.kernel.org/doc/html/v4.17/process/submitting-patches.html) on patches or pull requests that are being sent. The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as an open-source patch. The rules are pretty simple: if you can certify the below:

CERTIFICATE OF ORIGIN V 1.1

"By making a contribution to this project, I certify that:

1.) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or 2.) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or 3.) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it. 4.) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved."

```
This is my commit message

Signed-off-by: Awesome Developer <developer@example.org>
```

Code without a proper signoff cannot be merged into the main branch. Note: You must use your real name (sorry, no pseudonyms or anonymous contributions.)

The text can either be manually added to your commit body, or you can add either `-s` or `--signoff` to your usual `git commit` commands:

```
git commit --signoff -m "This is my commit message"
git commit -s -m "This is my commit message"
```

This will use your default git configuration which is found in .git/config. To change this, you can use the following commands:

```
git config --global user.name "Awesome Developer"
git config --global user.email "awesome.developer.@example.org"
```

If you have authored a commit that is missing the signed-off-by line, you can amend your commits and push them to GitHub.

```
git commit --amend --signoff
```

If you've pushed your changes to GitHub already you'll need to force push your branch after this with `git push -f`.

12.1.3 Commit Message formatting

We request that your first commit follow a particular format, and we **require** that your PR title follow the format. The format is:

```
FEAT-#9999: Add `DataFrame.rolling` functionality, to enable rolling window operations
```

The `FEAT` component represents the type of commit. This component of the commit message can be one of the following:

- FEAT: A new feature that is added
- DOCS: Documentation improvements or updates
- FIX: A bugfix contribution
- REFACTOR: Moving or removing code without change in functionality
- TEST: Test updates or improvements
- PERF: Performance enhancements

The `#9999` component of the commit message should be the issue number in the Modin GitHub issue tracker: https://github.com/modin-project/modin/issues. This is important because it links commits to their issues.

The commit message should follow a colon (:) and be descriptive and succinct.

A Modin CI job on GitHub will enforce that your pull request title follows the format we suggest. Note that if you update the PR title, you have to push another commit (even if it's empty) or amend your last commit for the job to pick up the new PR title. Re-running the job in Github Actions won't work.

12.1.4 General Rules for committers

- Try to write a PR name as descriptive as possible.
- Try to keep PRs as small as possible. One PR should be making one semantically atomic change.
- Don't merge your own PRs even if you are technically able to do it.

12.1.5 Development Dependencies

We recommend doing development in a virtualenv or conda environment, though this decision is ultimately yours. You will want to run the following in order to install all of the required dependencies for running the tests and formatting the code:

```
conda env create --file environment-dev.yml
# or
pip install -r requirements-dev.txt
```

12.1.6 Code Formatting and Lint

We use black for code formatting. Before you submit a pull request, please make sure that you run the following from the project root:

```
black modin/ asv_bench/benchmarks scripts/doc_checker.py
```

We also use flake8 to check linting errors. Running the following from the project root will ensure that it passes the lint checks on Github Actions:

```
flake8 modin/ asv_bench/benchmarks scripts/doc_checker.py
```

We test that this has been run on our Github Actions test suite. If you do this and find that the tests are still failing, try updating your version of black and flake8.

12.1.7 Adding a test

If you find yourself fixing a bug or adding a new feature, don't forget to add a test to the test suite to verify its correctness! More on testing and the layout of the tests can be found in our testing documentation. We ask that you follow the existing structure of the tests for ease of maintenance.

12.1.8 Running the tests

To run the entire test suite, run the following from the project root:

```
pytest modin/pandas/test
```

The test suite is very large, and may take a long time if you run every test. If you've only modified a small amount of code, it may be sufficient to run a single test or some subset of the test suite. In order to run a specific test run:

```
pytest modin/pandas/test::test_new_functionality
```

The entire test suite is automatically run for each pull request.

12.1.9 Performance measurement

We use Asv tool for performance tracking of various Modin functionality. The results can be viewed here: Asv dashboard.

More information can be found in the Asv readme.

12.1.10 Building documentation

To build the documentation, please follow the steps below from the project root:

```
pip install -r docs/requirements-doc.txt
sphinx-build -b html docs docs/build
```

To visualize the documentation locally, run the following from *build* folder:

```
python -m http.server <port>
# python -m http.server 1234
```

then open the browser at *0.0.0.0:<port>* (e.g. *0.0.0.0:1234*).

12.1.11 Contributing a new execution framework or in-memory format

If you are interested in contributing support for a new execution framework or in-memory format, please make sure you understand the *architecture* of Modin.

The best place to start the discussion for adding a new execution framework or in-memory format is the developer mailing list.

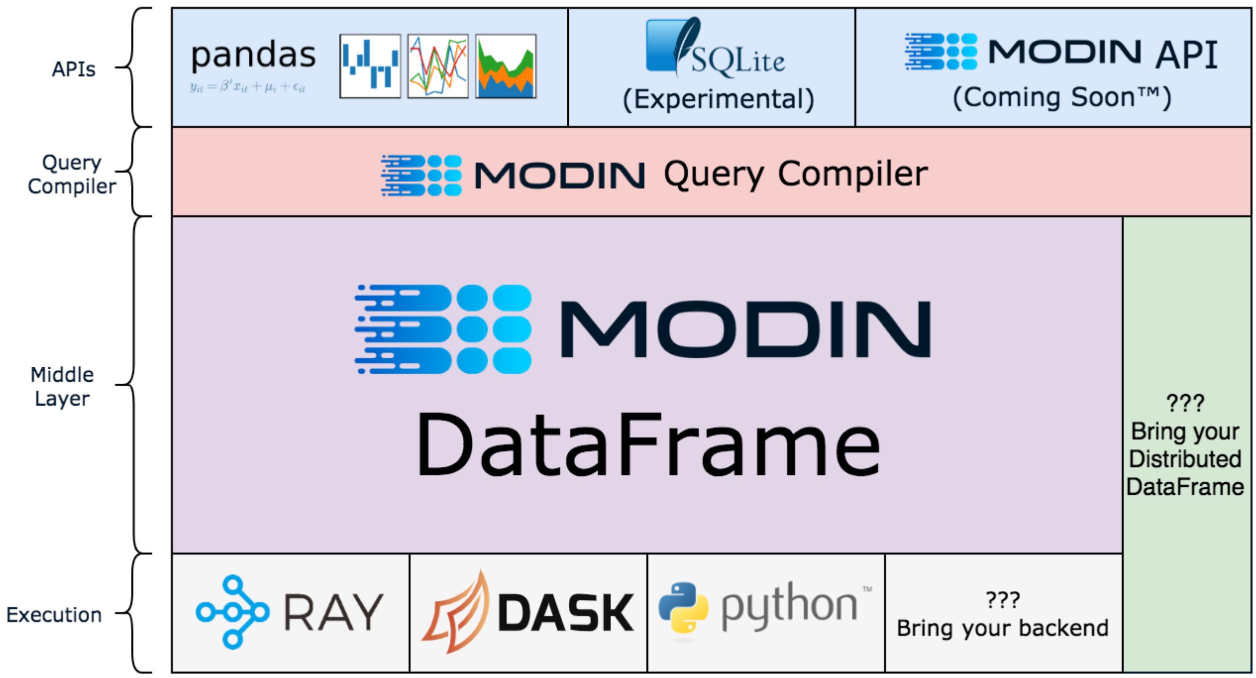
More docs on this coming soon…

12.2 System Architecture

In this section, we will lay out the overall system architecture for Modin, as well as go into detail about the component design, implementation and other important details. This document also contains important reference information for those interested in contributing new functionality, bugfixes and enhancements.

12.2.1 High-Level Architectural View

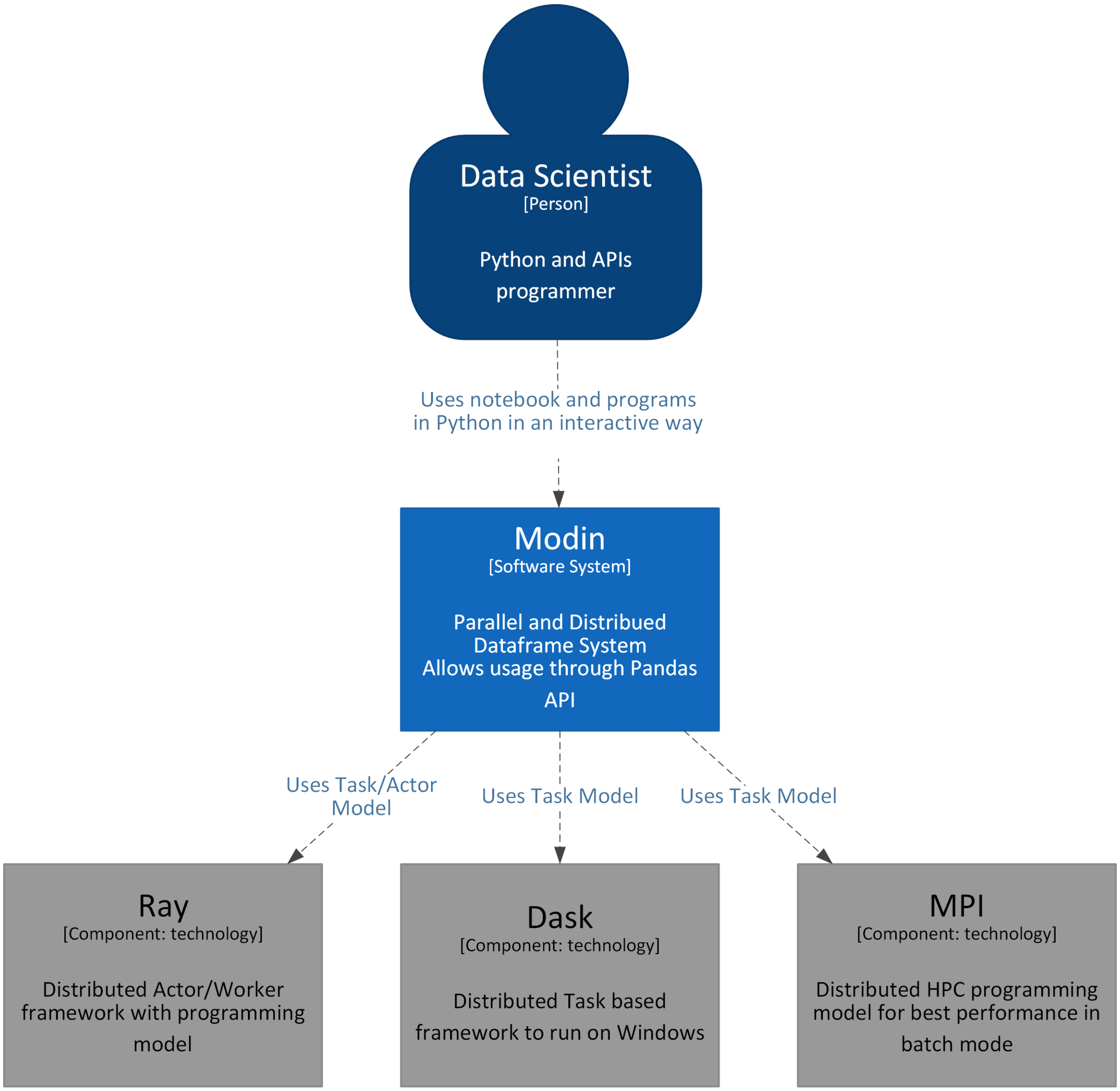
The diagram below outlines the general layered view to the components of Modin with a short description of each major section of the documentation following.



Modin is logically separated into different layers that represent the hierarchy of a typical Database Management System. Abstracting out each component allows us to individually optimize and swap out components without affecting the rest of the system. We can implement, for example, new compute kernels that are optimized for a certain type of data and can simply plug it in to the existing infrastructure by implementing a small interface. It can still be distributed by our choice of compute engine with the logic internally.

12.2.2 System View

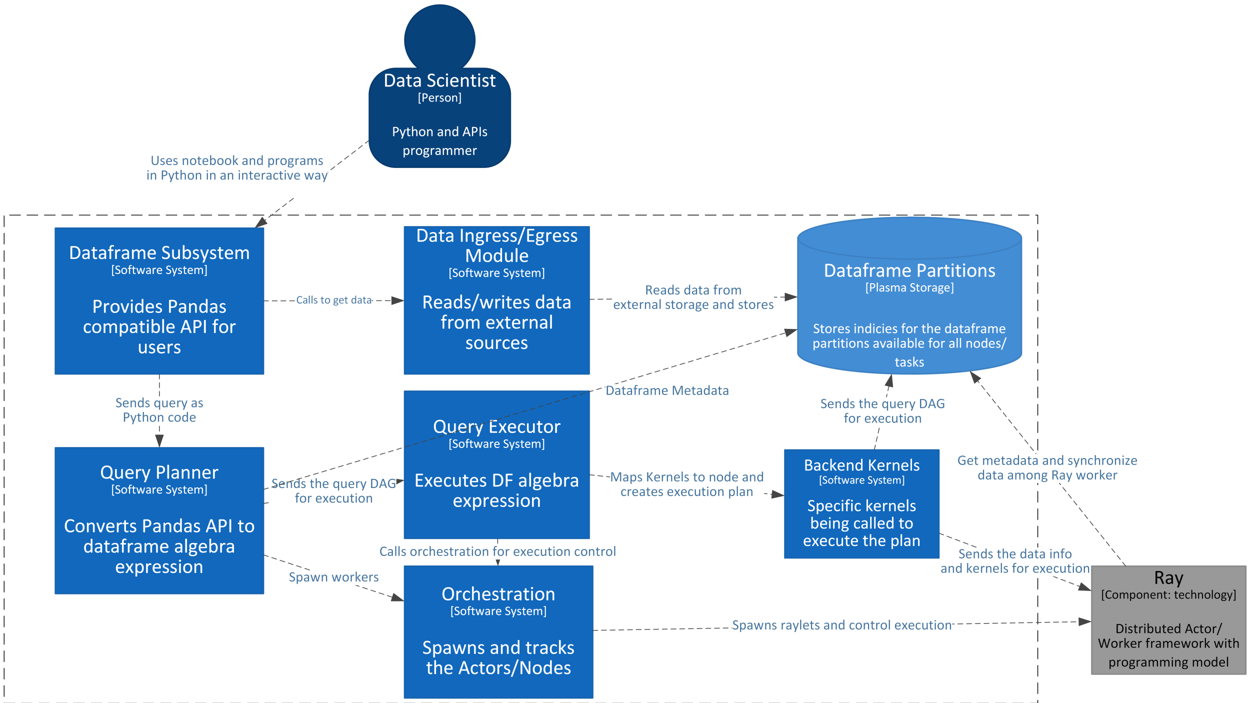
A top-down view of Modin's architecture is detailed below:



The user - Data Scientist interacts with the Modin system by sending interactive or batch commands through API and Modin executes them using various execution engines: Ray, Dask and MPI are currently supported.

12.2.3 Subsystem/Container View

If we click down to the next level of details we will see that inside Modin the layered architecture is implemented using several interacting components:



For the simplicity the other execution systems - Dask and MPI are omitted and only Ray execution is shown.

- Dataframe subsystem is the backbone of the dataframe holding and query compilation. It is responsible for dispatching the ingress/egress to the appropriate module, getting the pandas API and calling the query compiler to convert calls to the internal intermediate Dataframe Algebra.
- Data Ingress/Egress Module is working in conjunction with Dataframe and Partitions subsystem to read data split into partitions and send data into the appropriate node for storing.
- Query Planner is subsystem that translates the pandas API to intermediate Dataframe Algebra representation DAG and performs an initial set of optimizations.
- Query Executor is responsible for getting the Dataframe Algebra DAG, performing further optimizations based on a selected storage format and mapping or compiling the Dataframe Algebra DAG to and actual execution sequence.
- Storage formats module is responsible for mapping the abstract operation to an actual executor call, e.g. pandas, custom format.
- Orchestration subsystem is responsible for spawning and controlling the actual execution environment for the selected execution. It spawns the actual nodes, fires up the execution environment, e.g. Ray, monitors the state of executors and provides telemetry

12.2.4 Component View

User queries which perform data transformation, data ingress or data egress pass through the Modin components detailed below. The path the query takes is mostly similar across execution systems.

Data Transformation

Query Compiler

The Query Compiler receives queries from the pandas API layer. The API layer is responsible for ensuring a clean input to the Query Compiler. The Query Compiler must have knowledge of the compute kernels and in-memory format of the data in order to efficiently compile the query.

The Query Compiler is responsible for sending the compiled query to the Core Modin Dataframe. In this design, the Query Compiler does not have information about where or when the query will be executed, and gives the control of the partition layout to the Modin Dataframe.

In the interest of reducing the pandas API, the Query Compiler layer closely follows the pandas API, but cuts out a large majority of the repetition.

Core Modin Dataframe

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 can wait until another operation triggers computation. In the future, we plan to add additional query planning and laziness to Modin to ensure that queries are performed efficiently.

The structure of the Core Modin Dataframe is extensible, such that any operation that could be better optimized for a given execution can be overridden and optimized in that way.

This layer has a significantly reduced API from the QueryCompiler and the user-facing API. Each of these APIs represents a single way of performing a given operation or behavior.

Core Modin Dataframe API

More documentation can be found internally in the code. This API is not complete, but represents an overwhelming majority of operations and behaviors.

This API can be implemented by other distributed/parallel DataFrame libraries and plugged in to Modin as well. Create an issue or discuss on our Slack for more information!

The Core Modin Dataframe is responsible for the data layout and shuffling, partitioning, and serializing the tasks that get sent to each partition. Other implementations of the Modin Dataframe interface will have to handle these as well.

Partition Manager

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.

This abstraction separates the actual data movement and function application from the Dataframe layer to keep the Core Dataframe API small and separately optimize the data movement and metadata management.

Partitions

Partitions are responsible for managing a subset of the Dataframe. As mentioned below, 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, allowing Modin to exploit potential optimizations across both. These optimizations are explained further on the pages specific to the execution framework.

Execution Engine

This layer performs computation on partitions of the data. The Modin Dataframe is designed to work with task parallel frameworks, but integration with data parallel frameworks should be possible with some effort.

Storage Format

The storage format describes the in-memory partition type. The base storage format in Modin is pandas. In the default case, the Modin Dataframe operates on partitions that contain `pandas.DataFrame` objects.

Data Ingress

Note: Data ingress operations (e.g. `read_csv`) in Modin load data from the source into partitions and vice versa for data egress (e.g. `to_csv`) operation. Improved performance is achieved by reading/writing in partitions in parallel.

Data ingress starts with a function in the pandas API layer (e.g. `read_csv`). Then the user's query is passed to the Factory Dispatcher, which defines a factory specific for the execution. The factory for execution contains an IO class (e.g. `PandasOnRayIO`) whose responsibility is to perform a parallel read/write from/to a file. This IO class contains class methods with interfaces and names that are similar to pandas IO functions (e.g. `PandasOnRayIO.read_csv`). The IO class declares the Modin Dataframe and Query Compiler classes specific for the execution engine and storage format to ensure the correct object is constructed. It also declares IO methods that are mix-ins containing a combination of the engine-specific class for deploying remote tasks, the class for parsing the given file format and the class handling the chunking of the format-specific file on the head node (see dispatcher classes implementation details). The output from the IO class data ingress function is a Modin Dataframe.

Data Egress

Data egress operations (e.g. `to_csv`) are similar to data ingress operations up to execution-specific IO class functions construction. Data egress functions of the IO class are defined slightly different from data ingress functions and created only specifically for the engine since partitions already have information about its storage format. Using the IO class, data is exported from partitions to the target file.

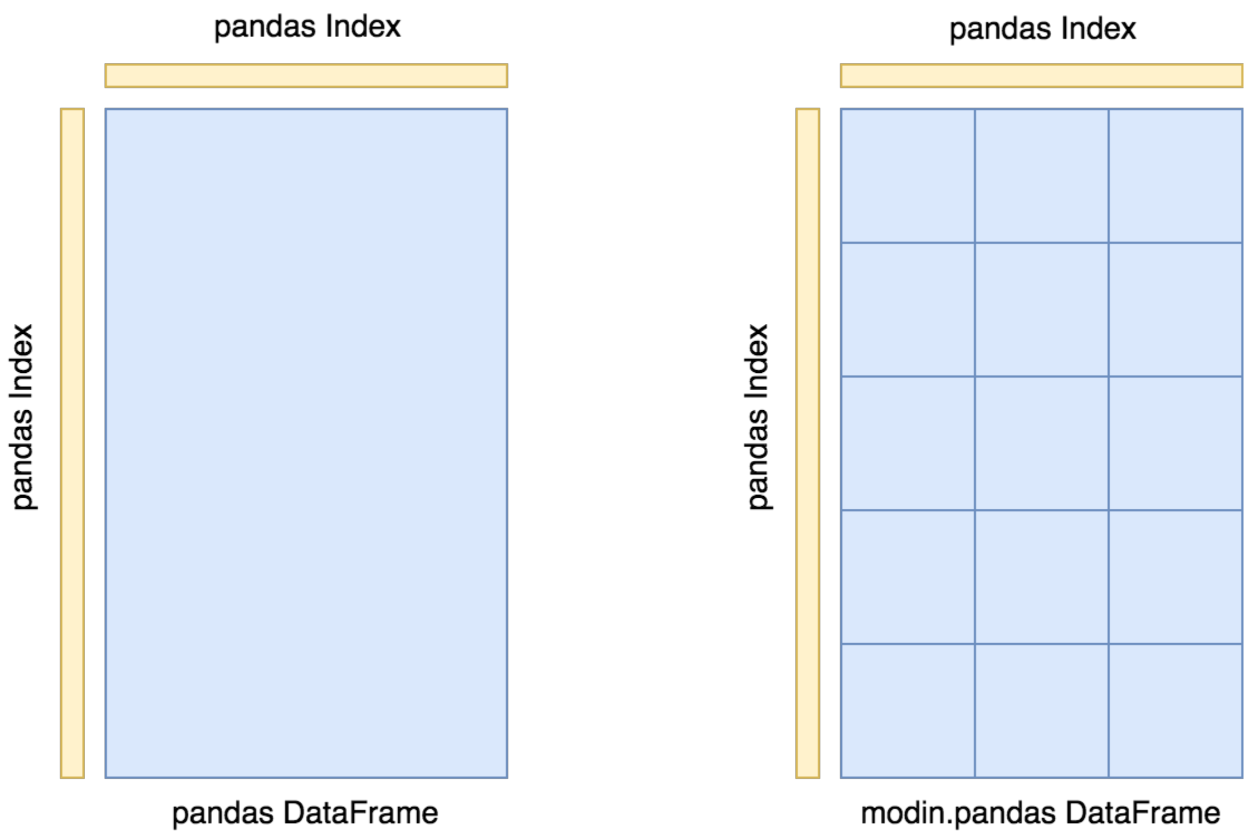
Supported Execution Engines and Storage Formats

This is a list of execution engines and in-memory formats supported in Modin. If you would like to contribute a new execution engine or in-memory format, please see the documentation page on *contributing*.

- ***pandas on Ray***
  - Uses the Ray execution framework.
  - The storage format is *pandas* and the in-memory partition type is a pandas DataFrame.
  - For more information on the execution path, see the pandas on Ray page.
- ***pandas on Dask***
  - Uses the Dask Futures execution framework.
  - The storage format is *pandas* and the in-memory partition type is a pandas DataFrame.
  - For more information on the execution path, see the pandas on Dask page.
- ***pandas on MPI***
  - Uses MPI through the Unidist execution framework.
  - The storage format is *pandas* and the in-memory partition type is a pandas DataFrame.
  - For more information on the execution path, see the pandas on Unidist page.
- ***pandas on Python***
  - Uses native python execution - mainly used for debugging.
  - The storage format is *pandas* and the in-memory partition type is a pandas DataFrame.
  - For more information on the execution path, see the pandas on Python page.
- **cuDF on Ray (experimental)**
  - Uses the Ray execution framework.
  - The storage format is *cudf* and the in-memory partition type is a cuDF DataFrame.
  - For more information on the execution path, see the cuDF on Ray page.

12.2.5 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. The following figure illustrates this concept.



Currently, the main in-memory format of each partition is a pandas DataFrame (pandas storage format).

12.2.6 Index

We currently use the `pandas.Index` object for indexing both columns and rows. In the future, we will implement a distributed, pandas-compatible Index object in order to remove this scaling limitation from the system. Most workloads will not be affected by this scalability limit since it only appears when operating on more than 10's of billions of columns or rows. **Important note**: If you are using the default index (`pandas.RangeIndex`) there is a fixed memory overhead (~200 bytes) and there will be no scalability issues with the index.

12.2.7 API

The API is the outer-most layer that faces users. The following classes contain Modin's implementation of the pandas API:

Base pandas Dataset API

The class implements functionality that is common to Modin's pandas API for both `DataFrame` and `Series` classes.

Public API

class `modin.pandas.base.`**`BasePandasDataset`**

Implement most of the common code that exists in DataFrame/Series.

Since both objects share the same underlying representation, and the algorithms are the same, we use this object to define the general behavior of those objects and then use those objects to define the output type.

Notes

See pandas API documentation for pandas.DataFrame, pandas.Series for more.

`abs``()` → Self

Return a *BasePandasDataset* with absolute numeric value of each element.

Notes

See pandas API documentation for pandas.DataFrame.abs, pandas.Series.abs for more.

`add``(`*other*, *axis='columns'*, *level=None*, *fill_value=None*`)` → Self

Return addition of *BasePandasDataset* and *other*, element-wise (binary operator *add*).

Notes

See pandas API documentation for pandas.DataFrame.add, pandas.Series.add for more.

`agg``(`*func=None*, *axis=0*, **args*, ***kwargs*`)` → *DataFrame* | *Series* | Scalar

Aggregate using one or more operations over the specified axis.

Notes

See pandas API documentation for pandas.DataFrame.aggregate, pandas.Series.aggregate for more.

`aggregate``(`*func=None*, *axis=0*, **args*, ***kwargs*`)` → *DataFrame* | *Series* | Scalar

Aggregate using one or more operations over the specified axis.

Notes

See pandas API documentation for pandas.DataFrame.aggregate, pandas.Series.aggregate for more.

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=None*, *fill_value=None*, *method=_NoDefault.no_default*, *limit=_NoDefault.no_default*, *fill_axis=_NoDefault.no_default*, *broadcast_axis=_NoDefault.no_default*) → tuple[Self, Self]

Align two objects on their axes with the specified join method.

Notes

See pandas API documentation for pandas.DataFrame.align, pandas.Series.align for more.

all(*axis=0*, *bool_only=False*, *skipna=True*, ***kwargs*) → Self

Return whether all elements are True, potentially over an axis.

Notes

See pandas API documentation for pandas.DataFrame.all, pandas.Series.all for more.

any(***, *axis=0*, *bool_only=False*, *skipna=True*, ***kwargs*) → Self

Return whether any element is True, potentially over an axis.

Notes

See pandas API documentation for pandas.DataFrame.any, pandas.Series.any for more.

apply(*func*, *axis*, *raw*, *result_type*, *args*, ***kwds*) → BaseQueryCompiler

Apply a function along an axis of the *BasePandasDataset*.

Notes

See pandas API documentation for pandas.DataFrame.apply, pandas.Series.apply for more.

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*) → Self

Convert time series to specified frequency.

Notes

See pandas API documentation for pandas.DataFrame.asfreq, pandas.Series.asfreq for more.

asof(*where*, *subset=None*) → Self

Return the last row(s) without any NaNs before *where*.

Notes

See pandas API documentation for pandas.DataFrame.asof, pandas.Series.asof for more.

astype(*dtype*, *copy=None*, *errors='raise'*) → Self

Cast a Modin object to a specified dtype *dtype*.

Notes

See pandas API documentation for pandas.DataFrame.astype, pandas.Series.astype for more.

property at: _LocIndexer

Get a single value for a row/column label pair.

Notes

See pandas API documentation for pandas.DataFrame.at, pandas.Series.at for more.

at_time(*time*, *asof=False*, *axis=None*) → Self

Select values at particular time of day (e.g., 9:30AM).

Notes

See pandas API documentation for pandas.DataFrame.at_time, pandas.Series.at_time for more.

backfill(***, *axis=None*, *inplace=False*, *limit=None*, *downcast=_NoDefault.no_default*) → Self

Synonym for *DataFrame.bfill*.

Notes

See pandas API documentation for pandas.DataFrame.backfill, pandas.Series.backfill for more.

between_time(*start_time*, *end_time*, *inclusive='both'*, *axis=None*) → Self

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

Parameters

- **start_time** (*datetime.time or str*) – Initial time as a time filter limit.
- **end_time** (*datetime.time or str*) – End time as a time filter limit.
- **inclusive** (*{"both", "neither", "left", "right"}, default "both"*) – Include boundaries; whether to set each bound as closed or open.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – Determine range time on index or columns value. For *Series* this parameter is unused and defaults to 0.

Returns

Data from the original object filtered to the specified dates range.

Return type

Series or *DataFrame*

Raises

`TypeError` – If the index is not a `DatetimeIndex`

See also:

`at_time`
Select values at a particular time of the day.

`first`
Select initial periods of time series based on a date offset.

`last`
Select final periods of time series based on a date offset.

`DatetimeIndex.indexer_between_time`
Get just the index locations for values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

Notes

See pandas API documentation for pandas.DataFrame.between_time for more.

bfill(***, *axis=None*, *inplace=False*, *limit=None*, *limit_area=None*, *downcast=_NoDefault.no_default*) → Self

Synonym for *DataFrame.fillna* with `method='bfill'`.

Notes

See pandas API documentation for pandas.DataFrame.bfill, pandas.Series.bfill for more.

bool() → bool

Return the bool of a single element *BasePandasDataset*.

Notes

See pandas API documentation for pandas.DataFrame.bool, pandas.Series.bool for more.

clip(*lower=None*, *upper=None*, ***, *axis=None*, *inplace=False*, ***kwargs*) → Self

Trim values at input threshold(s).

Notes

See pandas API documentation for pandas.DataFrame.clip, pandas.Series.clip for more.

combine(*other*, *func*, *fill_value=None*, ***kwargs*) → Self

Perform combination of *BasePandasDataset*-s according to *func*.

Notes

See pandas API documentation for pandas.DataFrame.combine, pandas.Series.combine for more.

combine_first(*other*) → Self

Update null elements with value in the same location in *other*.

Notes

See pandas API documentation for pandas.DataFrame.combine_first, pandas.Series.combine_first for more.

convert_dtypes(*infer_objects: bool = True*, *convert_string: bool = True*, *convert_integer: bool = True*, *convert_boolean: bool = True*, *convert_floating: bool = True*, *dtype_backend: DtypeBackend = 'numpy_nullable'*) → Self

Convert columns to best possible dtypes using dtypes supporting `pd.NA`.

Notes

See pandas API documentation for pandas.DataFrame.convert_dtypes, pandas.Series.convert_dtypes for more.

copy(*deep=True*) → Self

Make a copy of the object's metadata.

Notes

See pandas API documentation for pandas.DataFrame.copy, pandas.Series.copy for more.

`count`(*axis=0*, *numeric_only=False*) → *Series* | Scalar

Count non-NA cells for *BasePandasDataset*.

Notes

See pandas API documentation for pandas.DataFrame.count, pandas.Series.count for more.

`cummax`(*axis=None*, *skipna=True*, **args*, ***kwargs*) → Self

Return cumulative maximum over a *BasePandasDataset* axis.

Notes

See pandas API documentation for pandas.DataFrame.cummax, pandas.Series.cummax for more.

`cummin`(*axis=None*, *skipna=True*, **args*, ***kwargs*) → Self

Return cumulative minimum over a *BasePandasDataset* axis.

Notes

See pandas API documentation for pandas.DataFrame.cummin, pandas.Series.cummin for more.

`cumprod`(*axis=None*, *skipna=True*, **args*, ***kwargs*) → Self

Return cumulative product over a *BasePandasDataset* axis.

Notes

See pandas API documentation for pandas.DataFrame.cumprod, pandas.Series.cumprod for more.

`cumsum`(*axis=None*, *skipna=True*, **args*, ***kwargs*) → Self

Return cumulative sum over a *BasePandasDataset* axis.

Notes

See pandas API documentation for pandas.DataFrame.cumsum, pandas.Series.cumsum for more.

`describe`(*percentiles=None*, *include=None*, *exclude=None*) → Self

Generate descriptive statistics.

Notes

See pandas API documentation for pandas.DataFrame.describe, pandas.Series.describe for more.

`diff`(*periods=1*, *axis=0*) → Self

First discrete difference of element.

Notes

See pandas API documentation for pandas.DataFrame.diff, pandas.Series.diff for more.

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get floating division of *BasePandasDataset* and *other*, element-wise (binary operator *truediv*).

Notes

See pandas API documentation for pandas.DataFrame.truediv, pandas.Series.truediv for more.

divide(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get floating division of *BasePandasDataset* and *other*, element-wise (binary operator *truediv*).

Notes

See pandas API documentation for pandas.DataFrame.truediv, pandas.Series.truediv for more.

drop(*labels=None*, ***, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*) → Self

Drop specified labels from *BasePandasDataset*.

Notes

See pandas API documentation for pandas.DataFrame.drop, pandas.Series.drop for more.

drop_duplicates(*keep='first'*, *inplace=False*, ***kwargs*) → Self

Return *BasePandasDataset* with duplicate rows removed.

Notes

See pandas API documentation for pandas.DataFrame.drop_duplicates, pandas.Series.drop_duplicates for more.

droplevel(*level*, *axis=0*) → Self

Return *BasePandasDataset* with requested index / column level(s) removed.

Notes

See pandas API documentation for pandas.DataFrame.droplevel, pandas.Series.droplevel for more.

dropna(***, *axis: Axis = 0*, *how: str | lib.NoDefault = _NoDefault.no_default*, *thresh: int | lib.NoDefault = _NoDefault.no_default*, *subset: IndexLabel = None*, *inplace: bool = False*, *ignore_index: bool = False*) → Self

Remove missing values.

Notes

See pandas API documentation for pandas.DataFrame.dropna, pandas.Series.dropna for more.

eq(*other*, *axis='columns'*, *level=None*) → Self

Get equality of *BasePandasDataset* and *other*, element-wise (binary operator *eq*).

Notes

See pandas API documentation for pandas.DataFrame.eq, pandas.Series.eq for more.

ewm(*com: float | None = None*, *span: float | None = None*, *halflife: float | TimedeltaConvertibleTypes | None = None*, *alpha: float | None = None*, *min_periods: int | None = 0*, *adjust: bool = True*, *ignore_na: bool = False*, *axis: Axis = _NoDefault.no_default*, *times: str | np.ndarray | BasePandasDataset | None = None*, *method: str = 'single'*) → pandas.core.window.ewm.ExponentialMovingWindow

Provide exponentially weighted (EW) calculations.

Notes

See pandas API documentation for pandas.DataFrame.ewm, pandas.Series.ewm for more.

expanding(*min_periods=1*, *axis=_NoDefault.no_default*, *method='single'*) → Expanding

Provide expanding window calculations.

Notes

See pandas API documentation for pandas.DataFrame.expanding, pandas.Series.expanding for more.

explode(*column*, *ignore_index: bool = False*) → Self

Transform each element of a list-like to a row.

Notes

See pandas API documentation for pandas.DataFrame.explode, pandas.Series.explode for more.

ffill(***, *axis=None*, *inplace=False*, *limit=None*, *limit_area=None*, *downcast=_NoDefault.no_default*) → Self | None

Synonym for *DataFrame.fillna* with `method='ffill'`.

Notes

See pandas API documentation for pandas.DataFrame.ffill, pandas.Series.ffill for more.

fillna(*squeeze_self*, *squeeze_value*, *value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=_NoDefault.no_default*) → Self | None

Fill NA/NaN values using the specified method.

Parameters

- **squeeze_self** (*bool*) – If True then self contains a Series object, if False then self contains a DataFrame object.

- **squeeze_value** (*bool*) – If True then value contains a Series object, if False then value contains a DataFrame object.
- **value** (*scalar, dict, Series, or DataFrame, default: None*) – Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.
- **method** (*{'backfill', 'bfill', 'pad', 'ffill', None}, default: None*) – Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use next valid observation to fill gap.
- **axis** (*{None, 0, 1}, default: None*) – Axis along which to fill missing values.
- **inplace** (*bool, default: False*) – If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).
- **limit** (*int, default: None*) – If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.
- **downcast** (*dict, default: None*) – A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns
Object with missing values filled or None if `inplace=True`.

Return type
Series, *DataFrame* or None

Notes

See pandas API documentation for pandas.DataFrame.fillna, pandas.Series.fillna for more.

filter(*items=None*, *like=None*, *regex=None*, *axis=None*) → Self

Subset the *BasePandasDataset* rows or columns according to the specified index labels.

Notes

See pandas API documentation for pandas.DataFrame.filter, pandas.Series.filter for more.

first(*offset*) → Self | None

Select initial periods of time series data based on a date offset.

Notes

See pandas API documentation for pandas.DataFrame.first, pandas.Series.first for more.

first_valid_index() → int

Return index for first non-NA value or None, if no non-NA value is found.

Notes

See pandas API documentation for pandas.DataFrame.first_valid_index, pandas.Series.first_valid_index for more.

property flags

Get the properties associated with this pandas object.

The available flags are

- `Flags.allows_duplicate_labels`

See also:

Flags
Flags that apply to pandas objects.

DataFrame.attrs
Global metadata applying to this dataset.

Notes

See pandas API documentation for pandas.DataFrame.flags, pandas.Series.flags for more. "Flags" differ from "metadata". Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `DataFrame.attrs`.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2]})
>>> df.flags
<Flags(allows_duplicate_labels=True)>
```

Flags can be get or set using `.`

```
>>> df.flags.allows_duplicate_labels
True
>>> df.flags.allows_duplicate_labels = False
```

Or by slicing with a key

```
>>> df.flags["allows_duplicate_labels"]
False
>>> df.flags["allows_duplicate_labels"] = True
```

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get integer division of *BasePandasDataset* and *other*, element-wise (binary operator *floordiv*).

Notes

See pandas API documentation for pandas.DataFrame.floordiv, pandas.Series.floordiv for more.

ge(*other*, *axis='columns'*, *level=None*) → Self

Get greater than or equal comparison of *BasePandasDataset* and *other*, element-wise (binary operator *ge*).

Notes

See pandas API documentation for pandas.DataFrame.ge, pandas.Series.ge for more.

get(*key*, *default=None*) → *DataFrame* | *Series* | Scalar

Get item from object for given key.

Notes

See pandas API documentation for pandas.DataFrame.get, pandas.Series.get for more.

gt(*other*, *axis='columns'*, *level=None*) → Self

Get greater than comparison of *BasePandasDataset* and *other*, element-wise (binary operator *gt*).

Notes

See pandas API documentation for pandas.DataFrame.gt, pandas.Series.gt for more.

head(*n=5*) → Self

Return the first *n* rows.

Notes

See pandas API documentation for pandas.DataFrame.head, pandas.Series.head for more.

property iat: _iLocIndexer

Get a single value for a row/column pair by integer position.

Notes

See pandas API documentation for pandas.DataFrame.iat, pandas.Series.iat for more.

idxmax(*axis=0*, *skipna=True*, *numeric_only=False*) → Self

Return index of first occurrence of maximum over requested axis.

Notes

See pandas API documentation for pandas.DataFrame.idxmax, pandas.Series.idxmax for more.

idxmin(*axis=0*, *skipna=True*, *numeric_only=False*) → Self

Return index of first occurrence of minimum over requested axis.

Notes

See pandas API documentation for pandas.DataFrame.idxmin, pandas.Series.idxmin for more.

property iloc: _iLocIndexer

Purely integer-location based indexing for selection by position.

Notes

See pandas API documentation for pandas.DataFrame.iloc, pandas.Series.iloc for more.

property index: Index

Get the index for this DataFrame.

Returns
The union of all indexes across the partitions.

Return type
pandas.Index

infer_objects(*copy=None*) → Self

Attempt to infer better dtypes for object columns.

Notes

See pandas API documentation for pandas.DataFrame.infer_objects, pandas.Series.infer_objects for more.

interpolate(*method='linear'*, ***, *axis=0*, *limit=None*, *inplace=False*, *limit_direction: Optional[str] = None*, *limit_area=None*, *downcast=_NoDefault.no_default*, ***kwargs*) → Self

Fill NaN values using an interpolation method.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

- **method** (*str, default 'linear'*) – Interpolation technique to use. One of:
  - 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
  - 'time': Works on daily and higher resolution data to interpolate given length of interval.
  - 'index', 'values': use the actual numerical values of the index.
  - 'pad': Fill in NaNs using existing values.
  - 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*, whereas 'spline' is passed to *scipy.interpolate.UnivariateSpline*. These methods use the numerical values of the index. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=5)`. Note that, *slinear* method in Pandas refers to the Scipy first order *spline* instead of Pandas first order *spline*.
  - 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima', 'cubicspline': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
  - 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives*.

- **axis** (*{{0 or 'index', 1 or 'columns', None}}, default None*) – Axis to interpolate along. For *Series* this parameter is unused and defaults to 0.
- **limit** (*int, optional*) – Maximum number of consecutive NaNs to fill. Must be greater than 0.
- **inplace** (*bool, default False*) – Update the data in place if possible.
- **limit_direction** (*{{'forward', 'backward', 'both'}}, Optional*) – Consecutive NaNs will be filled in this direction.

  If limit is specified:

  – If 'method' is 'pad' or 'ffill', 'limit_direction' must be 'forward'.

  – If 'method' is 'backfill' or 'bfill', 'limit_direction' must be 'backwards'.

  If 'limit' is not specified:

  – If 'method' is 'backfill' or 'bfill', the default is 'backward'

  – else the default is 'forward'

  raises ValueError if *limit_direction* is 'forward' or 'both' and
  method is 'backfill' or 'bfill'.

  raises ValueError if *limit_direction* is 'backward' or 'both' and
  method is 'pad' or 'ffill'.

- **limit_area** ({{*None*, 'inside', 'outside'}}, default None) – If limit is specified, consecutive NaNs will be filled with this restriction.

  – None: No fill restriction.

  – 'inside': Only fill NaNs surrounded by valid values (interpolate).

  – 'outside': Only fill NaNs outside valid values (extrapolate).

- **downcast** (*optional, 'infer' or None, defaults to None*) – Downcast dtypes if possible.

  Deprecated since version 2.1.0.

- ****kwargs** (*optional*) – Keyword arguments to pass on to the interpolating function.

Returns
Returns the same object type as the caller, interpolated at some or all NaN values or None if inplace=True.

Return type
Series or *DataFrame* or None

See also:

fillna
Fill missing values using different methods.

scipy.interpolate.Akima1DInterpolator
Piecewise cubic polynomials (Akima interpolator).

scipy.interpolate.BPoly.from_derivatives
Piecewise polynomial in the Bernstein basis.

scipy.interpolate.interp1d
Interpolate a 1-D function.

scipy.interpolate.KroghInterpolator
Interpolate polynomial (Krogh interpolator).

scipy.interpolate.PchipInterpolator
PCHIP 1-d monotonic cubic interpolation.

scipy.interpolate.CubicSpline
Cubic spline data interpolator.

Notes

See pandas API documentation for pandas.DataFrame.interpolate, pandas.Series.interpolate for more. The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation.

Examples

Filling in `NaN` in a `Series` via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```

Filling in `NaN` in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an `order` (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains `NaN`, because there is no entry before it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
```

(continues on next page)

(continued from previous page)

```
...                      (np.nan, 4.0, -4.0, 16.0)],
...                      columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

isin(*values*) → Self

Whether elements in *BasePandasDataset* are contained in *values*.

Notes

See pandas API documentation for pandas.DataFrame.isin, pandas.Series.isin for more.

isna() → Self

Detect missing values.

Notes

See pandas API documentation for pandas.DataFrame.isna, pandas.Series.isna for more.

isnull() → Self

Detect missing values.

Notes

See pandas API documentation for pandas.DataFrame.isna, pandas.Series.isna for more.

kurt(*axis=0*, *skipna=True*, *numeric_only=False*, ***kwargs*) → *Series* | float

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

- **axis** (*{index (0), columns (1)}*) – Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

  For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

  New in version 2.0.0.

- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.

- **numeric_only** (*bool, default False*) – Include only float, int, boolean columns. Not implemented for Series.

- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Examples

```
>>> s = pd.Series([1, 2, 2, 3], index=['cat', 'dog', 'dog', 'mouse
↪'])
>>> s
cat      1
dog      2
dog      2
mouse    3
dtype: int64
>>> s.kurt()
1.5
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 2, 3], 'b': [3, 4, 4, 4]},
...                   index=['cat', 'dog', 'dog', 'mouse'])
>>> df
       a   b
  cat  1   3
  dog  2   4
  dog  2   4
mouse  3   4
>>> df.kurt()
a   1.5
b   4.0
dtype: float64
```

With axis=None

```
>>> df.kurt(axis=None).round(6)
-0.988693
```

Using axis=1

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [3, 4], 'd':␣
↪[1, 2]},
...                   index=['cat', 'dog'])
>>> df.kurt(axis=1)
```

(continues on next page)

(continued from previous page)

```
cat   -6.0
dog   -6.0
dtype: float64
```

Return type
Series or scalar

Notes

See pandas API documentation for pandas.DataFrame.kurt for more.

kurtosis(*axis=0*, *skipna=True*, *numeric_only=False*, ***kwargs*) → *Series* | float

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

- **axis** (*{index (0), columns (1)}*) – Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

  For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

  New in version 2.0.0.

- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.
- **numeric_only** (*bool, default False*) – Include only float, int, boolean columns. Not implemented for Series.
- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Examples

```
>>> s = pd.Series([1, 2, 2, 3], index=['cat', 'dog', 'dog', 'mouse
↪'])
>>> s
cat    1
dog    2
dog    2
mouse  3
dtype: int64
>>> s.kurt()
1.5
```

With a DataFrame

```
>>> df = pd.DataFrame({'a': [1, 2, 2, 3], 'b': [3, 4, 4, 4]},
...                   index=['cat', 'dog', 'dog', 'mouse'])
>>> df
       a   b
  cat  1   3
```

(continues on next page)

(continued from previous page)

```
  dog   2   4
  dog   2   4
mouse   3   4
>>> df.kurt()
a   1.5
b   4.0
dtype: float64
```

With axis=None

```
>>> df.kurt(axis=None).round(6)
-0.988693
```

Using axis=1

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [3, 4], 'd':␣
↪[1, 2]},
...                   index=['cat', 'dog'])
>>> df.kurt(axis=1)
cat   -6.0
dog   -6.0
dtype: float64
```

Return type
Series or scalar

Notes

See pandas API documentation for pandas.DataFrame.kurt for more.

last(*offset*) → Self

Select final periods of time series data based on a date offset.

Notes

See pandas API documentation for pandas.DataFrame.last, pandas.Series.last for more.

last_valid_index() → int

Return index for last non-NA value or None, if no non-NA value is found.

Notes

See pandas API documentation for pandas.DataFrame.last_valid_index, pandas.Series.last_valid_index for more.

le(*other*, *axis='columns'*, *level=None*) → Self

Get less than or equal comparison of *BasePandasDataset* and *other*, element-wise (binary operator *le*).

Notes

See pandas API documentation for pandas.DataFrame.le, pandas.Series.le for more.

property loc: _LocIndexer

Get a group of rows and columns by label(s) or a boolean array.

Notes

See pandas API documentation for pandas.DataFrame.loc, pandas.Series.loc for more.

lt(*other*, *axis='columns'*, *level=None*) → Self

Get less than comparison of *BasePandasDataset* and *other*, element-wise (binary operator *lt*).

Notes

See pandas API documentation for pandas.DataFrame.lt, pandas.Series.lt for more.

mask(*cond*, *other=_NoDefault.no_default*, ***, *inplace: bool = False*, *axis: Optional[Axis] = None*, *level: Optional[Level] = None*) → Self | None

Replace values where the condition is True.

Notes

See pandas API documentation for pandas.DataFrame.mask, pandas.Series.mask for more.

max(*axis: Axis = 0*, *skipna=True*, *numeric_only=False*, ***kwargs*) → *Series* | None

Return the maximum of the values over the requested axis.

Notes

See pandas API documentation for pandas.DataFrame.max, pandas.Series.max for more.

mean(*axis: Axis = 0*, *skipna=True*, *numeric_only=False*, ***kwargs*) → *Series* | float

Return the mean of the values over the requested axis.

Notes

See pandas API documentation for pandas.DataFrame.mean, pandas.Series.mean for more.

median(*axis: Axis = 0*, *skipna=True*, *numeric_only=False*, ***kwargs*) → *Series* | float

Return the mean of the values over the requested axis.

Notes

See pandas API documentation for pandas.DataFrame.median, pandas.Series.median for more.

memory_usage(*index=True*, *deep=False*) → *Series* | None

Return the memory usage of the *BasePandasDataset*.

Notes

See pandas API documentation for pandas.DataFrame.memory_usage, pandas.Series.memory_usage for more.

min(*axis: Axis = 0*, *skipna: bool = True*, *numeric_only=False*, ***kwargs*) → *Series* | None

Return the minimum of the values over the requested axis.

Notes

See pandas API documentation for pandas.DataFrame.min, pandas.Series.min for more.

mod(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get modulo of *BasePandasDataset* and *other*, element-wise (binary operator *mod*).

Notes

See pandas API documentation for pandas.DataFrame.mod, pandas.Series.mod for more.

mode(*axis=0*, *numeric_only=False*, *dropna=True*) → Self

Get the mode(s) of each element along the selected axis.

Notes

See pandas API documentation for pandas.DataFrame.mode, pandas.Series.mode for more.

modin

alias of `ModinAPI`

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get multiplication of *BasePandasDataset* and *other*, element-wise (binary operator *mul*).

Notes

See pandas API documentation for pandas.DataFrame.mul, pandas.Series.mul for more.

multiply(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get multiplication of *BasePandasDataset* and *other*, element-wise (binary operator *mul*).

Notes

See pandas API documentation for pandas.DataFrame.mul, pandas.Series.mul for more.

ne(*other*, *axis='columns'*, *level=None*) → Self

Get Not equal comparison of *BasePandasDataset* and *other*, element-wise (binary operator *ne*).

Notes

See pandas API documentation for pandas.DataFrame.ne, pandas.Series.ne for more.

notna() → Self

Detect existing (non-missing) values.

Notes

See pandas API documentation for pandas.DataFrame.notna, pandas.Series.notna for more.

notnull() → Self

Detect existing (non-missing) values.

Notes

See pandas API documentation for pandas.DataFrame.notna, pandas.Series.notna for more.

nunique(*axis=0*, *dropna=True*) → *Series* | int

Return number of unique elements in the *BasePandasDataset*.

Notes

See pandas API documentation for pandas.DataFrame.nunique, pandas.Series.nunique for more.

pad(***, *axis=None*, *inplace=False*, *limit=None*, *downcast=_NoDefault.no_default*) → Self | None

Synonym for *DataFrame.ffill*.

Notes

See pandas API documentation for pandas.DataFrame.pad, pandas.Series.pad for more.

pct_change(*periods=1*, *fill_method=_NoDefault.no_default*, *limit=_NoDefault.no_default*, *freq=None*, ***kwargs*) → Self

Percentage change between the current and a prior element.

Notes

See pandas API documentation for pandas.DataFrame.pct_change, pandas.Series.pct_change for more.

pipe(*func: Callable[..., T] | tuple[Callable[..., T], str]*, **args*, ***kwargs*) → T

Apply chainable functions that expect *BasePandasDataset*.

Notes

See pandas API documentation for pandas.DataFrame.pipe, pandas.Series.pipe for more.

pop(*item*) → *Series* | Scalar

Return item and drop from frame. Raise KeyError if not found.

Notes

See pandas API documentation for pandas.DataFrame.pop, pandas.Series.pop for more.

pow(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get exponential power of *BasePandasDataset* and *other*, element-wise (binary operator *pow*).

Notes

See pandas API documentation for pandas.DataFrame.pow, pandas.Series.pow for more.

quantile(*q*, *axis*, *numeric_only*, *interpolation*, *method*) → *DataFrame* | *Series* | Scalar

Return values at the given quantile over requested axis.

Notes

See pandas API documentation for pandas.DataFrame.quantile, pandas.Series.quantile for more.

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Return addition of *BasePandasDataset* and *other*, element-wise (binary operator *radd*).

Notes

See pandas API documentation for pandas.DataFrame.radd, pandas.Series.radd for more.

rank(*axis=0*, *method: str = 'average'*, *numeric_only=False*, *na_option: str = 'keep'*, *ascending: bool = True*, *pct: bool = False*) → Self

Compute numerical data ranks (1 through n) along axis.

By default, equal values are assigned a rank that is the average of the ranks of those values.

Parameters

- **axis** (`{0 or 'index', 1 or 'columns'}, default 0`) – Index to direct ranking. For *Series* this parameter is unused and defaults to 0.
- **method** (`{'average', 'min', 'max', 'first', 'dense'}, default 'average'`) – How to rank the group of records that have the same value (i.e. ties):
  - average: average rank of the group

  - min: lowest rank in the group
  - max: highest rank in the group
  - first: ranks assigned in order they appear in the array
  - dense: like 'min', but rank always increases by 1 between groups.

- **numeric_only** (*bool, default False*) – For DataFrame objects, rank only numeric columns if set to True.

  Changed in version 2.0.0: The default value of `numeric_only` is now `False`.

- **na_option** (*{'keep', 'top', 'bottom'}, default 'keep'*) – How to rank NaN values:
  - keep: assign NaN rank to NaN values
  - top: assign lowest rank to NaN values
  - bottom: assign highest rank to NaN values

- **ascending** (*bool, default True*) – Whether or not the elements should be ranked in ascending order.

- **pct** (*bool, default False*) – Whether or not to display the returned rankings in percentile form.

Returns

Return a Series or DataFrame with data ranks as values.

Return type

same type as caller

See also:

core.groupby.DataFrameGroupBy.rank
Rank of values within each group.

core.groupby.SeriesGroupBy.rank
Rank of values within each group.

Examples

```
>>> df = pd.DataFrame(data={'Animal': ['cat', 'penguin', 'dog',
...                                    'spider', 'snake'],
...                         'Number_legs': [4, 2, 4, 8, np.nan]})
>>> df
    Animal  Number_legs
0      cat          4.0
1  penguin          2.0
2      dog          4.0
3   spider          8.0
4    snake          NaN
```

Ties are assigned the mean of the ranks (by default) for the group.

```
>>> s = pd.Series(range(5), index=list("abcde"))
>>> s["d"] = s["b"]
>>> s.rank()
```

(continues on next page)

(continued from previous page)

```
a    1.0
b    2.5
c    4.0
d    2.5
e    5.0
dtype: float64
```

The following example shows how the method behaves with the above parameters:

- default_rank: this is the default behaviour obtained without using any parameter.
- max_rank: setting `method = 'max'` the records that have the same values are ranked using the highest rank (e.g.: since 'cat' and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
- NA_bottom: choosing `na_option = 'bottom'`, if there are records with NaN values they are placed at the bottom of the ranking.
- pct_rank: when setting `pct = True`, the ranking is expressed as percentile rank.

```
>>> df['default_rank'] = df['Number_legs'].rank()
>>> df['max_rank'] = df['Number_legs'].rank(method='max')
>>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
>>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
>>> df
    Animal  Number_legs  default_rank  max_rank  NA_bottom  pct_rank
0      cat          4.0           2.5       3.0        2.5     0.625
1  penguin          2.0           1.0       1.0        1.0     0.250
2      dog          4.0           2.5       3.0        2.5     0.625
3   spider          8.0           4.0       4.0        4.0     1.000
4    snake          NaN           NaN       NaN        5.0       NaN
```

Notes

See pandas API documentation for pandas.DataFrame.rank for more.

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get floating division of *BasePandasDataset* and *other*, element-wise (binary operator *rtruediv*).

Notes

See pandas API documentation for pandas.DataFrame.rtruediv, pandas.Series.rtruediv for more.

reindex(*index=None*, *columns=None*, *copy=True*, ***kwargs*) → Self

Conform *BasePandasDataset* to new index with optional filling logic.

Notes

See pandas API documentation for pandas.DataFrame.reindex, pandas.Series.reindex for more.

rename_axis(*mapper=_NoDefault.no_default*, ***, *index=_NoDefault.no_default*, *columns=_NoDefault.no_default*, *axis=0*, *copy=None*, *inplace=False*) → *DataFrame* | *Series* | None

Set the name of the axis for the index or columns.

Notes

See pandas API documentation for pandas.DataFrame.rename_axis, pandas.Series.rename_axis for more.

reorder_levels(*order*, *axis=0*) → Self

Rearrange index levels using input order.

Notes

See pandas API documentation for pandas.DataFrame.reorder_levels, pandas.Series.reorder_levels for more.

resample(*rule*, *axis: Axis = _NoDefault.no_default*, *closed: Optional[str] = None*, *label: Optional[str] = None*, *convention: str = _NoDefault.no_default*, *kind: Optional[str] = _NoDefault.no_default*, *on: Level = None*, *level: Level = None*, *origin: str | TimestampConvertibleTypes = 'start_day'*, *offset: Optional[TimedeltaConvertibleTypes] = None*, *group_keys=False*) → Resampler

Resample time-series data.

Notes

See pandas API documentation for pandas.DataFrame.resample, pandas.Series.resample for more.

reset_index(*level: IndexLabel = None*, ***, *drop: bool = False*, *inplace: bool = False*, *col_level: Hashable = 0*, *col_fill: Hashable = ''*, *allow_duplicates=_NoDefault.no_default*, *names: Hashable | Sequence[Hashable] = None*) → *DataFrame* | *Series* | None

Reset the index, or a level of it.

Notes

See pandas API documentation for pandas.DataFrame.reset_index, pandas.Series.reset_index for more.

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get integer division of *BasePandasDataset* and *other*, element-wise (binary operator *rfloordiv*).

Notes

See pandas API documentation for pandas.DataFrame.rfloordiv, pandas.Series.rfloordiv for more.

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get modulo of *BasePandasDataset* and *other*, element-wise (binary operator *rmod*).

Notes

See pandas API documentation for pandas.DataFrame.rmod, pandas.Series.rmod for more.

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Notes

See pandas API documentation for pandas.DataFrame.rmul, pandas.Series.rmul for more.

rolling(*window*, *min_periods: int | None = None*, *center: bool = False*, *win_type: str | None = None*, *on: str | None = None*, *axis: Axis = _NoDefault.no_default*, *closed: str | None = None*, *step: int | None = None*, *method: str = 'single'*) → Rolling | Window

Provide rolling window calculations.

Notes

See pandas API documentation for pandas.DataFrame.rolling, pandas.Series.rolling for more.

round(*decimals=0*, **args*, ***kwargs*) → Self

Round a *BasePandasDataset* to a variable number of decimal places.

Notes

See pandas API documentation for pandas.DataFrame.round, pandas.Series.round for more.

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get exponential power of *BasePandasDataset* and *other*, element-wise (binary operator *rpow*).

Notes

See pandas API documentation for pandas.DataFrame.rpow, pandas.Series.rpow for more.

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get subtraction of *BasePandasDataset* and *other*, element-wise (binary operator *rsub*).

Notes

See pandas API documentation for pandas.DataFrame.rsub, pandas.Series.rsub for more.

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get floating division of *BasePandasDataset* and *other*, element-wise (binary operator *rtruediv*).

Notes

See pandas API documentation for pandas.DataFrame.rtruediv, pandas.Series.rtruediv for more.

sample(*n: int | None = None*, *frac: float | None = None*, *replace: bool = False*, *weights=None*, *random_state: RandomState | None = None*, *axis: Axis | None = None*, *ignore_index: bool = False*) → Self

Return a random sample of items from an axis of object.

Notes

See pandas API documentation for pandas.DataFrame.sample, pandas.Series.sample for more.

sem(*axis: Axis = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only=False*, ***kwargs*) → *Series* | float

Return unbiased standard error of the mean over requested axis.

Notes

See pandas API documentation for pandas.DataFrame.sem, pandas.Series.sem for more.

set_axis(*labels*, ***, *axis: Axis = 0*, *copy=None*) → Self

Assign desired index to given axis.

Notes

See pandas API documentation for pandas.DataFrame.set_axis, pandas.Series.set_axis for more.

set_flags(***, *copy: bool = False*, *allows_duplicate_labels: Optional[bool] = None*) → Self

Return a new *BasePandasDataset* with updated flags.

Notes

See pandas API documentation for pandas.DataFrame.set_flags, pandas.Series.set_flags for more.

shift(*periods: int = 1*, *freq=None*, *axis: Axis = 0*, *fill_value: Hashable = _NoDefault.no_default*, *suffix=None*) → Self | *DataFrame*

Shift index by desired number of periods with an optional time *freq*.

Notes

See pandas API documentation for pandas.DataFrame.shift, pandas.Series.shift for more.

property size: int

Return an int representing the number of elements in this *BasePandasDataset* object.

Notes

See pandas API documentation for pandas.DataFrame.size, pandas.Series.size for more.

skew(*axis: Axis = 0*, *skipna: bool = True*, *numeric_only=False*, ***kwargs*) → *Series* | float

Return unbiased skew over requested axis.

Notes

See pandas API documentation for pandas.DataFrame.skew, pandas.Series.skew for more.

sort_index(***, *axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *ignore_index: bool = False*, *key: Optional[IndexKeyFunc] = None*) → Self | None

Sort object by labels (along an axis).

Notes

See pandas API documentation for pandas.DataFrame.sort_index, pandas.Series.sort_index for more.

sort_values(*by*, ***, *axis=0*, *ascending=True*, *inplace: bool = False*, *kind='quicksort'*, *na_position='last'*, *ignore_index: bool = False*, *key: Optional[IndexKeyFunc] = None*) → Self | None

Sort by the values along either axis.

Notes

See pandas API documentation for pandas.DataFrame.sort_values, pandas.Series.sort_values for more.

std(*axis: Axis = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only=False*, ***kwargs*) → *Series* | float

Return sample standard deviation over requested axis.

Notes

See pandas API documentation for pandas.DataFrame.std, pandas.Series.std for more.

sub(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get subtraction of *BasePandasDataset* and *other*, element-wise (binary operator *sub*).

Notes

See pandas API documentation for pandas.DataFrame.sub, pandas.Series.sub for more.

subtract(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get subtraction of *BasePandasDataset* and *other*, element-wise (binary operator *sub*).

Notes

See pandas API documentation for pandas.DataFrame.sub, pandas.Series.sub for more.

swapaxes(*axis1*, *axis2*, *copy=None*) → Self

Interchange axes and swap values axes appropriately.

Notes

See pandas API documentation for pandas.DataFrame.swapaxes, pandas.Series.swapaxes for more.

swaplevel(*i=-2*, *j=-1*, *axis=0*) → Self

Swap levels *i* and *j* in a *MultiIndex*.

Notes

See pandas API documentation for pandas.DataFrame.swaplevel, pandas.Series.swaplevel for more.

tail(*n=5*) → Self

Return the last *n* rows.

Notes

See pandas API documentation for pandas.DataFrame.tail, pandas.Series.tail for more.

take(*indices*, *axis=0*, ***kwargs*) → Self

Return the elements in the given *positional* indices along an axis.

Notes

See pandas API documentation for pandas.DataFrame.take, pandas.Series.take for more.

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

Copy object to the system clipboard.

Notes

See pandas API documentation for pandas.DataFrame.to_clipboard, pandas.Series.to_clipboard for more.

to_csv(*path_or_buf=None*, *sep=','*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression='infer'*, *quoting=None*, *quotechar='"'*, *lineterminator=None*, *chunksize=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*, *errors: str = 'strict'*, *storage_options: StorageOptions = None*) → str | None

Write object to a comma-separated values (csv) file.

Parameters

- **path_or_buf** (*str, path object, file-like object, or None, default None*) – String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, the result is returned as a string. If a non-binary file object is passed, it should be opened with *newline=''*, disabling universal newlines. If a binary file object is passed, *mode* might need to contain a *'b'*.
- **sep** (*str, default ','*) – String of length 1. Field delimiter for the output file.
- **na_rep** (*str, default ''*) – Missing data representation.
- **float_format** (*str, Callable, default None*) – Format string for floating point numbers. If a Callable is given, it takes precedence over other numeric formatting parameters, like decimal.
- **columns** (*sequence, optional*) – Columns to write.
- **header** (*bool or list of str, default True*) – Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.
- **index** (*bool, default True*) – Write row names (index).
- **index_label** (*str or sequence, or False, default None*) – Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.
- **mode** (*{'w', 'x', 'a'}, default 'w'*) – Forwarded to either *open(mode=)* or *fsspec.open(mode=)* to control the file opening. Typical values include:
  - 'w', truncate the file first.
  - 'x', exclusive creation, failing if the file already exists.
  - 'a', append to the end of file if it exists.
- **encoding** (*str, optional*) – A string representing the encoding to use in the output file, defaults to 'utf-8'. *encoding* is not supported if *path_or_buf* is a non-binary file object.
- **compression** (*str or dict, default 'infer'*) – For on-the-fly compression of the output data. If 'infer' and 'path_or_buf' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdCompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

  New in version 1.5.0: Added support for *.tar* files.

  May be a dict with key 'method' as compression mode and other entries as additional compression options if compression mode is 'zip'.

  Passing compression options as keys in dict is supported for compression modes 'gzip', 'bz2', 'zstd', and 'zip'.

- **quoting** (*optional constant from csv module*) – Defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.
- **quotechar** (*str, default '"'*) – String of length 1. Character used to quote fields.
- **lineterminator** (*str, optional*) – The newline character or character sequence to use in the output file. Defaults to *os.linesep*, which depends on the OS in which this method is called ('\n' for linux, '\r\n' for Windows, i.e.).

  Changed in version 1.5.0: Previously was line_terminator, changed for consistency with read_csv and the standard library 'csv' module.
- **chunksize** (*int or None*) – Rows to write at a time.
- **date_format** (*str, default None*) – Format string for datetime objects.
- **doublequote** (*bool, default True*) – Control quoting of *quotechar* inside a field.
- **escapechar** (*str, default None*) – String of length 1. Character used to escape *sep* and *quotechar* when appropriate.
- **decimal** (*str, default '.'*) – Character recognized as decimal separator. E.g. use ',' for European data.
- **errors** (*str, default 'strict'*) – Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.
- **storage_options** (*dict, optional*) – Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

Returns
If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

Return type
None or str

See also:

`read_csv`
Load a CSV file into a DataFrame.

`to_excel`
Write DataFrame to an Excel file.

Examples

Create 'out.csv' containing 'df' without indices

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_csv('out.csv', index=False)
```

Create 'out.zip' containing 'out.csv'

```
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
>>> compression_opts = dict(method='zip',
...                         archive_name='out.csv')
>>> df.to_csv('out.zip', index=False,
...           compression=compression_opts)
```

To write a csv file to a new folder or nested folder you will first need to create it using either Pathlib or os:

```
>>> from pathlib import Path
>>> filepath = Path('folder/subfolder/out.csv')
>>> filepath.parent.mkdir(parents=True, exist_ok=True)
>>> df.to_csv(filepath)
```

```
>>> import os
>>> os.makedirs('folder/subfolder', exist_ok=True)
>>> df.to_csv('folder/subfolder/out.csv')
```

Notes

See pandas API documentation for pandas.DataFrame.to_csv, pandas.Series.to_csv for more.

to_dict(*orient='dict'*, *into=<class 'dict'>*, *index=True*) → dict

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

Parameters

- **orient** (*str {'dict', 'list', 'series', 'split', 'tight', 'records', 'index'}*) – Determines the type of the values of the dictionary.
  - 'dict' (default) : dict like {column -> {index -> value}}
  - 'list' : dict like {column -> [values]}
  - 'series' : dict like {column -> Series(values)}
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'tight' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values], 'index_names' -> [index.names], 'column_names' -> [column.names]}
  - 'records' : list like [{column -> value}, … , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}

  New in version 1.4.0: 'tight' as an allowed value for the `orient` argument
- **into** (*class, default dict*) – The collections.abc.MutableMapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.
- **index** (*bool, default True*) – Whether to include the index item (and index_names item if *orient* is 'tight') in the returned dictionary. Can only be `False` when *orient* is 'split' or 'tight'.

  New in version 2.0.0.

Returns
Return a collections.abc.MutableMapping object representing the DataFrame. The resulting transformation depends on the *orient* parameter.

Return type
dict, list or collections.abc.MutableMapping

See also:

DataFrame.from_dict
Create a DataFrame from a dictionary.

DataFrame.to_json
Convert a DataFrame to JSON format.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['row1', 'row2'])
>>> df
      col1  col2
row1     1  0.50
row2     2  0.75
>>> df.to_dict()
{'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 0.5, 'row2': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': row1    1
         row2    2
Name: col1, dtype: int64,
'col2': row1    0.50
        row2    0.75
Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1, 'col2': 0.5}, {'col1': 2, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'row1': {'col1': 1, 'col2': 0.5}, 'row2': {'col1': 2, 'col2': 0.75}}
```

```
>>> df.to_dict('tight')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]], 'index_names': [None], 'column_names': [None]}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('row1', 1), ('row2', 2)])),
             ('col2', OrderedDict([('row1', 0.5), ('row2', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2, 'col2': 0.75})]
```

Notes

See pandas API documentation for pandas.DataFrame.to_dict, pandas.Series.to_dict for more.

to_excel(*excel_writer*, *sheet_name='Sheet1'*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *inf_rep='inf'*, *freeze_panes=None*, *storage_options: Optional[dict[str, Any]] = None*, *engine_kwargs=None*) → None

Write object to an Excel sheet.

Notes

See pandas API documentation for pandas.DataFrame.to_excel, pandas.Series.to_excel for more.

to_hdf(*path_or_buf*, *key: str*, *mode: Literal['a', 'w', 'r+'] = 'a'*, *complevel: int | None = None*, *complib: Literal['zlib', 'lzo', 'bzip2', 'blosc'] | None = None*, *append: bool = False*, *format: Literal['fixed', 'table'] | None = None*, *index: bool = True*, *min_itemsize: int | dict[str, int] | None = None*, *nan_rep=None*, *dropna: bool | None = None*, *data_columns: Literal[True] | list[str] | None = None*, *errors: str = 'strict'*, *encoding: str = 'UTF-8'*) → None

Write the contained data to an HDF5 file using HDFStore.

Notes

See pandas API documentation for pandas.DataFrame.to_hdf, pandas.Series.to_hdf for more.

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression='infer'*, *index=None*, *indent=None*, *storage_options: StorageOptions = None*, *mode='w'*) → str | None

Convert the object to a JSON string.

Notes

See pandas API documentation for pandas.DataFrame.to_json, pandas.Series.to_json for more.

to_latex(*buf=None*, *columns=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*, *caption=None*, *label=None*, *position=None*) → str | None

Render object to a LaTeX tabular, longtable, or nested table.

Notes

See pandas API documentation for pandas.DataFrame.to_latex, pandas.Series.to_latex for more.

to_markdown(*buf=None*, *mode: str = 'wt'*, *index: bool = True*, *storage_options: Optional[dict[str, Any]] = None*, ***kwargs*) → str

Print *BasePandasDataset* in Markdown-friendly format.

Notes

See pandas API documentation for pandas.DataFrame.to_markdown, pandas.Series.to_markdown for more.

to_numpy(*dtype=None*, *copy=False*, *na_value=_NoDefault.no_default*) → ndarray

Convert the *BasePandasDataset* to a NumPy array or a Modin wrapper for NumPy array.

Notes

See pandas API documentation for pandas.DataFrame.to_numpy, pandas.Series.to_numpy for more.

to_period(*freq=None*, *axis=0*, *copy=None*) → Self

Convert *BasePandasDataset* from DatetimeIndex to PeriodIndex.

Notes

See pandas API documentation for pandas.DataFrame.to_period, pandas.Series.to_period for more.

to_pickle(*path*, *compression: Optional[Union[Literal['infer', 'gzip', 'bz2', 'zip', 'xz', 'zstd', 'tar'], dict[str, Any]]] = 'infer'*, *protocol: int = 5*, *storage_options: Optional[dict[str, Any]] = None*) → None

Pickle (serialize) object to file.

Notes

See pandas API documentation for pandas.DataFrame.to_pickle, pandas.Series.to_pickle for more.

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*, *method=None*) → int | None

Write records stored in a *BasePandasDataset* to a SQL database.

Notes

See pandas API documentation for pandas.DataFrame.to_sql, pandas.Series.to_sql for more.

to_string(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *max_rows=None*, *min_rows=None*, *max_cols=None*, *show_dimensions=False*, *decimal='.'*, *line_width=None*, *max_colwidth=None*, *encoding=None*) → str | None

Render a *BasePandasDataset* to a console-friendly tabular output.

Notes

See pandas API documentation for pandas.DataFrame.to_string, pandas.Series.to_string for more.

to_timestamp(*freq=None*, *how='start'*, *axis=0*, *copy=None*) → Self

Cast to DatetimeIndex of timestamps, at *beginning* of period.

Notes

See pandas API documentation for pandas.DataFrame.to_timestamp, pandas.Series.to_timestamp for more.

to_xarray()

Return an xarray object from the *BasePandasDataset*.

Notes

See pandas API documentation for pandas.DataFrame.to_xarray, pandas.Series.to_xarray for more.

transform(*func*, *axis=0*, **args*, ***kwargs*) → Self

Call `func` on self producing a *BasePandasDataset* with the same axis shape as self.

Notes

See pandas API documentation for pandas.DataFrame.transform, pandas.Series.transform for more.

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*) → Self

Get floating division of *BasePandasDataset* and *other*, element-wise (binary operator *truediv*).

Notes

See pandas API documentation for pandas.DataFrame.truediv, pandas.Series.truediv for more.

truncate(*before=None*, *after=None*, *axis=None*, *copy=None*) → Self

Truncate a *BasePandasDataset* before and after some index value.

Notes

See pandas API documentation for pandas.DataFrame.truncate, pandas.Series.truncate for more.

tz_convert(*tz*, *axis=0*, *level=None*, *copy=None*) → Self

Convert tz-aware axis to target time zone.

Notes

See pandas API documentation for pandas.DataFrame.tz_convert, pandas.Series.tz_convert for more.

tz_localize(*tz*, *axis=0*, *level=None*, *copy=None*, *ambiguous='raise'*, *nonexistent='raise'*) → Self

Localize tz-naive index of a *BasePandasDataset* to target time zone.

Notes

See pandas API documentation for pandas.DataFrame.tz_localize, pandas.Series.tz_localize for more.

value_counts(*subset: Sequence[Hashable] | None = None*, *normalize: bool = False*, *sort: bool = True*, *ascending: bool = False*, *dropna: bool = True*) → *Series*

Return a Series containing the frequency of each distinct row in the Dataframe.

Parameters

- **subset** (*label or list of labels, optional*) – Columns to use when counting unique combinations.
- **normalize** (*bool, default False*) – Return proportions rather than frequencies.
- **sort** (*bool, default True*) – Sort by frequencies when True. Sort by DataFrame column values when False.
- **ascending** (*bool, default False*) – Sort in ascending order.
- **dropna** (*bool, default True*) – Don't include counts of rows that contain NA values.

  New in version 1.3.0.

Return type

Series

See also:

Series.value_counts

Equivalent method on Series.

Notes

See pandas API documentation for pandas.DataFrame.value_counts for more. The returned Series will have a MultiIndex with one level per input column but an Index (non-multi) for a single label. By default, rows that contain any NA values are omitted from the result. By default, the resulting Series will be in descending order so that the first element is the most frequently-occurring row.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
...                    'num_wings': [2, 0, 0, 0]},
...                   index=['falcon', 'dog', 'cat', 'ant'])
>>> df
        num_legs  num_wings
falcon         2          2
dog            4          0
cat            4          0
ant            6          0
```

```
>>> df.value_counts()
num_legs  num_wings
4         0            2
2         2            1
6         0            1
Name: count, dtype: int64
```

```
>>> df.value_counts(sort=False)
num_legs  num_wings
2         2            1
4         0            2
6         0            1
Name: count, dtype: int64
```

```
>>> df.value_counts(ascending=True)
num_legs  num_wings
2         2            1
6         0            1
4         0            2
Name: count, dtype: int64
```

```
>>> df.value_counts(normalize=True)
num_legs  num_wings
4         0            0.50
2         2            0.25
6         0            0.25
Name: proportion, dtype: float64
```

With *dropna* set to *False* we can also count rows with NA values.

```
>>> df = pd.DataFrame({'first_name': ['John', 'Anne', 'John', 'Beth'],
...                    'middle_name': ['Smith', pd.NA, pd.NA, 'Louise']})
>>> df
  first_name middle_name
0       John       Smith
1       Anne        <NA>
2       John        <NA>
3       Beth      Louise
```

```
>>> df.value_counts()
first_name  middle_name
Beth        Louise         1
John        Smith          1
Name: count, dtype: int64
```

```
>>> df.value_counts(dropna=False)
first_name  middle_name
Anne        NaN            1
Beth        Louise         1
John        Smith          1
            NaN            1
Name: count, dtype: int64
```

```
>>> df.value_counts("first_name")
first_name
John    2
Anne    1
Beth    1
Name: count, dtype: int64
```

property values: ndarray

Return a NumPy representation of the *BasePandasDataset*.

Notes

See pandas API documentation for pandas.DataFrame.values, pandas.Series.values for more.

var(*axis: Axis = 0*, *skipna: bool = True*, *ddof: int = 1*, *numeric_only=False*, ***kwargs*) → *Series* | float

Return unbiased variance over requested axis.

Notes

See pandas API documentation for pandas.DataFrame.var, pandas.Series.var for more.

xs(*key*, *axis=0*, *level=None*, *drop_level: bool = True*) → Self

Return cross-section from the Series/DataFrame.

Notes

See pandas API documentation for pandas.DataFrame.xs, pandas.Series.xs for more.

DataFrame Module Overview

Modin's `pandas.DataFrame` API

Modin's `pandas.DataFrame` API is backed by a distributed object providing an identical API to pandas. After the user calls some `DataFrame` function, this call is internally rewritten into a representation that can be processed in parallel by the partitions. These results can be e.g., reduced to single output, identical to the single threaded pandas `DataFrame` method output.

Public API

class `modin.pandas.dataframe.`**`DataFrame`**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=None*, *query_compiler: BaseQueryCompiler = None*)

Modin distributed representation of `pandas.DataFrame`.

Internally, the data can be divided into partitions along both columns and rows in order to parallelize computations and utilize the user's hardware as much as possible.

Inherit common for `DataFrame`-s and `Series` functionality from the *BasePandasDataset* class.

Parameters

- **data** (`DataFrame`, `Series`, *pandas.DataFrame, ndarray, Iterable or dict, optional*) – Dict can contain `Series`, arrays, constants, dataclass or list-like objects. If data is a dict, column order follows insertion-order.
- **index** (*Index or array-like, optional*) – Index to use for resulting frame. Will default to `RangeIndex` if no indexing information part of input data and no index provided.
- **columns** (*Index or array-like, optional*) – Column labels to use for resulting frame. Will default to `RangeIndex` if no column labels are provided.
- **dtype** (*str, np.dtype, or pandas.ExtensionDtype, optional*) – Data type to force. Only a single dtype is allowed. If None, infer.
- **copy** (*bool, default: False*) – Copy data from inputs. Only affects `pandas.DataFrame` / 2d ndarray input.
- **query_compiler** (*BaseQueryCompiler, optional*) – A query compiler object to create the `DataFrame` from.

Notes

See pandas API documentation for pandas.DataFrame for more. `DataFrame` can be created either from passed *data* or *query_compiler*. If both parameters are provided, data source will be prioritized in the next order:

1) Modin `DataFrame` or `Series` passed with *data* parameter.
2) Query compiler from the *query_compiler* parameter.
3) Various pandas/NumPy/Python data structures passed with *data* parameter.

The last option is less desirable since import of such data structures is very inefficient, please use previously created Modin structures from the fist two options or import data using highly efficient Modin IO tools (for example `pd.read_csv`).

Usage Guide

The most efficient way to create Modin `DataFrame` is to import data from external storage using the highly efficient Modin IO methods (for example using `pd.read_csv`, see details for Modin IO methods in the IO page), but even if the data does not originate from a file, any pandas supported data type or `pandas.DataFrame` can be used. Internally, the `DataFrame` data is divided into partitions, which number along an axis usually corresponds to the number of the user's hardware CPUs. If needed, the number of partitions can be changed by setting `modin.config.NPartitions`.

Let's consider simple example of creation and interacting with Modin `DataFrame`:

```
import modin.config

# This explicitly sets the number of partitions
modin.config.NPartitions.put(4)

import modin.pandas as pd
import pandas

# Create Modin DataFrame from the external file
pd_dataframe = pd.read_csv("test_data.csv")
# Create Modin DataFrame from the python object
# data = {f'col{x}': [f'col{x}_{y}' for y in range(100, 356)] for x in range(4)}
# pd_dataframe = pd.DataFrame(data)
# Create Modin DataFrame from the pandas object
# pd_dataframe = pd.DataFrame(pandas.DataFrame(data))

# Show created DataFrame
print(pd_dataframe)

# List DataFrame partitions. Note, that internal API is intended for
# developers needs and was used here for presentation purposes
# only.
partitions = pd_dataframe._query_compiler._modin_frame._partitions
print(partitions)

# Show the first DataFrame partition
print(partitions[0][0].get())

Output:

# created DataFrame

        col0      col1      col2      col3
0    col0_100  col1_100  col2_100  col3_100
1    col0_101  col1_101  col2_101  col3_101
2    col0_102  col1_102  col2_102  col3_102
3    col0_103  col1_103  col2_103  col3_103
4    col0_104  col1_104  col2_104  col3_104
..        ...       ...       ...       ...
251  col0_351  col1_351  col2_351  col3_351
252  col0_352  col1_352  col2_352  col3_352
253  col0_353  col1_353  col2_353  col3_353
254  col0_354  col1_354  col2_354  col3_354
```

(continues on next page)

(continued from previous page)

```
255  col0_355  col1_355  col2_355  col3_355

[256 rows x 4 columns]

# List of DataFrame partitions

[[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e607f0>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e9a4f0>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e60820>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e609d0>]]

# The first DataFrame partition

        col0      col1      col2      col3
0   col0_100  col1_100  col2_100  col3_100
1   col0_101  col1_101  col2_101  col3_101
2   col0_102  col1_102  col2_102  col3_102
3   col0_103  col1_103  col2_103  col3_103
4   col0_104  col1_104  col2_104  col3_104
..       ...       ...       ...       ...
60  col0_160  col1_160  col2_160  col3_160
61  col0_161  col1_161  col2_161  col3_161
62  col0_162  col1_162  col2_162  col3_162
63  col0_163  col1_163  col2_163  col3_163
64  col0_164  col1_164  col2_164  col3_164

[65 rows x 4 columns]
```

As we show in the example above, Modin `DataFrame` can be easily created, and supports any input that pandas `DataFrame` supports. Also note that tuning of the `DataFrame` partitioning can be done by just setting a single config.

Series Module Overview

Modin's `pandas.Series` API

Modin's `pandas.Series` API is backed by a distributed object providing an identical API to pandas. After the user calls some `Series` function, this call is internally rewritten into a representation that can be processed in parallel by the partitions. These results can be e.g., reduced to single output, identical to the single threaded pandas `Series` method output.

Public API

class modin.pandas.series.**Series**(*data=None*, *index=None*, *dtype=None*, *name=None*, *copy=None*, *fastpath=_NoDefault.no_default*, *query_compiler: BaseQueryCompiler = None*)

Modin distributed representation of *pandas.Series*.

Internally, the data can be divided into partitions in order to parallelize computations and utilize the user's hardware as much as possible.

Inherit common for DataFrames and Series functionality from the *BasePandasDataset* class.

Parameters

- **data** (*modin.pandas.Series, array-like, Iterable, dict, or scalar value, optional*) – Contains data stored in Series. If data is a dict, argument order is maintained.
- **index** (*array-like or Index (1d), optional*) – Values must be hashable and have the same length as *data*.
- **dtype** (*str, np.dtype, or pandas.ExtensionDtype, optional*) – Data type for the output Series. If not specified, this will be inferred from *data*.
- **name** (*str, optional*) – The name to give to the Series.
- **copy** (*bool, default: False*) – Copy input data.
- **fastpath** (*bool, default: False*) – *pandas* internal parameter.
- **query_compiler** (*BaseQueryCompiler, optional*) – A query compiler object to create the Series from.

Notes

See pandas API documentation for pandas.Series for more.

Usage Guide

The most efficient way to create Modin `Series` is to import data from external storage using the highly efficient Modin IO methods (for example using `pd.read_csv`, see details for Modin IO methods in the IO page), but even if the data does not originate from a file, any pandas supported data type or `pandas.Series` can be used. Internally, the `Series` data is divided into partitions, which number along an axis usually corresponds to the number of the user's hardware CPUs. If needed, the number of partitions can be changed by setting `modin.config.NPartitions`.

Let's consider simple example of creation and interacting with Modin `Series`:

```
import modin.config

# This explicitly sets the number of partitions
modin.config.NPartitions.put(4)

import modin.pandas as pd
import pandas

# Create Modin Series from the external file
```

(continues on next page)

(continued from previous page)

```
pd_series = pd.read_csv("test_data.csv", header=None).squeeze()
# Create Modin Series from the python object
# pd_series = pd.Series([x for x in range(256)])
# Create Modin Series from the pandas object
# pd_series = pd.Series(pandas.Series([x for x in range(256)]))

# Show created `Series`
print(pd_series)

# List `Series` partitions. Note, that internal API is intended for
# developers needs and was used here for presentation purposes
# only.
partitions = pd_series._query_compiler._modin_frame._partitions
print(partitions)

# Show the first `Series` partition
print(partitions[0][0].get())

Output:

# created `Series`

0      100
1      101
2      102
3      103
4      104
    ...
251    351
252    352
253    353
254    354
255    355
Name: 0, Length: 256, dtype: int64

# List of `Series` partitions

[[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e607f0>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e9a4f0>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e60820>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e609d0>]]

# The first `Series` partition

    0
0   100
1   101
2   102
```

(continues on next page)

(continued from previous page)

```
3   103
4   104
..  ...
60  160
61  161
62  162
63  163
64  164

[65 rows x 1 columns]
```

As we show in the example above, Modin `Series` can be easily created, and supports any input that pandas `Series` supports. Also note that tuning of the `Series` partitioning can be done by just setting a single config.

12.2.8 Module/Class View

Modin's modules layout is shown below. Click on the links to deep dive into Modin's internal implementation details. The documentation covers most modules, with more docs being added everyday!

```
├───.github
├───asv_bench
├───ci
├───docker
├───docs
├───examples
├───modin
│   ├─── config
|   ├─── utils
│   ├───core
│   │   ├─── dataframe
│   │   │   ├─── algebra
│   │   │   ├─── base
│   │   │   └─── pandas
│   │   ├───execution
│   │   │   ├───dask
│   │   │   │   ├───common
│   │   │   │   └───implementations
│   │   │   │       └─── pandas_on_dask
│   │   │   ├─── dispatching
│   │   │   ├───python
│   │   │   │   └───implementations
│   │   │   │       └─── pandas_on_python
│   │   │   ├───ray
│   │   │   │   ├───common
│   │   │   │   ├─── generic
│   │   │   │   └───implementations
│   │   │   │       ├─── cudf_on_ray
│   │   │   │       └─── pandas_on_ray
│   │   │   └───unidist
│   │   │       ├───common
│   │   │       ├─── generic
│   │   │       └───implementations
│   │   │
```

```
│       │       │               └── pandas_on_unidist
│       │       ├── io
│       │       └── storage_formats
│       │               ├── base
│       │               ├──cudf
│       │               └── pandas
│       ├──distributed
│       │       ├──dataframe
│       │       │       └── pandas
│       ├──  experimental
│       │       ├──core
│       │       │       └── io
│       │       ├── pandas
│       │       ├── sklearn
│       │       ├──spreadsheet
│       │       ├── xgboost
│       │       └── batch
│       └──pandas
│               ├── dataframe
│               └── series
├──requirements
├──scripts
└──stress_tests
```

12.3 Partition API in Modin

When you are working with a *DataFrame*, you can unwrap its remote partitions to get the raw futures objects compatible with the execution engine (e.g. `ray.ObjectRef` for Ray). In addition to unwrapping of the remote partitions we also provide an API to construct a `modin.pandas.DataFrame` from raw futures objects.

12.3.1 Partition IPs

For finer grained placement control, Modin also provides an API to get the IP addresses of the nodes that hold each partition. You can pass the partitions having needed IPs to your function. It can help with minimizing of data movement between nodes.

12.3.2 Partition API implementations

By default, a *DataFrame* stores underlying partitions as pandas.DataFrame objects. You can find the specific implementation of Modin's Partition Interface in *pandas Partition API*.

12.3.3 Ray engine

However, it is worth noting that for Modin on Ray engine with pandas in-memory format IPs of the remote partitions may not match actual locations if the partitions are lower than 100 kB. Ray saves such objects (<= 100 kB, by default) in in-process store of the calling process (please, refer to Ray documentation for more information). We can't get IPs for such objects while maintaining good performance. So, you should keep in mind this for unwrapping of the remote partitions with their IPs. Several options are provided to handle the case in How to handle Ray objects that are lower 100 kB section.

12.3.4 Dask engine

There is no mentioned above issue for Modin on Dask engine with pandas in-memory format because Dask saves any objects in the worker process that processes a function (please, refer to Dask documentation for more information).

12.3.5 Unidist engine

Currently, Modin only supports MPI through unidist. There is no mentioned above issue for Modin on Unidist engine using MPI backend with pandas in-memory format because Unidist saves any objects in the MPI worker process that processes a function (please, refer to Unidist documentation for more information).

12.3.6 How to handle Ray objects that are lower than 100 kB

- If you are sure that each of the remote partitions being unwrapped is higher than 100 kB, you can just import Modin or perform ray.init() manually.
- If you don't know partition sizes you can pass the option _system_config={"max_direct_call_object_size": <nbytes>,}, where nbytes is threshold for objects that will be stored in in-process store, to ray.init().
- You can also start Ray as follows: ray start --head --system-config='{"max_direct_call_object_size":<nbytes>

Note that when specifying the threshold the performance of some Modin operations may change.

12.4 pandas on Ray

This section describes usage related documents for the pandas on Ray component of Modin.

Modin uses pandas as a primary memory format of the underlying partitions and optimizes queries ingested from the API layer in a specific way to this format. Thus, there is no need to care of choosing it but you can explicitly specify it anyway as shown below.

One of the execution engines that Modin uses is Ray. If you have Ray installed in your system, Modin also uses it by default to distribute computations.

If you want to be explicit, you could set the following environment variables:

```
export MODIN_ENGINE=ray
export MODIN_STORAGE_FORMAT=pandas
```

or turn it on in source code:

```
import modin.config as cfg
cfg.Engine.put('ray')
cfg.StorageFormat.put('pandas')
```

12.5 pandas on Dask

This section describes usage related documents for the pandas on Dask component of Modin.

Modin uses pandas as a primary memory format of the underlying partitions and optimizes queries ingested from the API layer in a specific way to this format. Thus, there is no need to care of choosing it but you can explicitly specify it anyway as shown below.

One of the execution engines that Modin uses is Dask. To enable the pandas on Dask execution you should set the following environment variables:

```
export MODIN_ENGINE=dask
export MODIN_STORAGE_FORMAT=pandas
```

or turn them on in source code:

```
import modin.config as cfg
cfg.Engine.put('dask')
cfg.StorageFormat.put('pandas')
```

12.6 pandas on Python

This section describes usage related documents for the pandas on Python component of Modin.

Modin uses pandas as the primary memory format of the underlying partitions and optimizes queries from the API layer in a specific way to this format. Since it is a default, you do not need to specify the pandas memory format, but we show how to explicitly set it below.

One of the execution engines that Modin uses is Python. This engine is sequential and used for debugging. To enable the pandas on Python execution you should set the following environment variables:

```
export MODIN_ENGINE=python
export MODIN_STORAGE_FORMAT=pandas
```

or turn a debug mode on:

```
export MODIN_DEBUG=True
export MODIN_STORAGE_FORMAT=pandas
```

or do the same in source code:

```
import modin.config as cfg
cfg.Engine.put('python')
cfg.StorageFormat.put('pandas')
```

```
import modin.config as cfg
cfg.IsDebug.put(True)
cfg.StorageFormat.put('pandas')
```

12.7 pandas on MPI through unidist

This section describes usage related documents for the pandas on MPI through unidist component of Modin.

Modin uses pandas as a primary memory format of the underlying partitions and optimizes queries ingested from the API layer in a specific way to this format. Thus, there is no need to care of choosing it but you can explicitly specify it anyway as shown below.

One of the execution engines that Modin uses is MPI through unidist. To enable the pandas on MPI through unidist execution you should set the following environment variables:

```
export MODIN_ENGINE=unidist
export MODIN_STORAGE_FORMAT=pandas
export UNIDIST_BACKEND=mpi
```

or turn it on in source code:

```
import modin.config as modin_cfg
import unidist.config as unidist_cfg

modin_cfg.Engine.put('unidist')
modin_cfg.StorageFormat.put('pandas')
unidist_cfg.Backend.put('mpi')
```

To run a python application you should use `mpiexec -n 1 python <script.py>` command.

```
mpiexec -n 1 python script.py
```

For more information on how to run a python application with unidist on MPI backend please refer to Unidist on MPI section of the unidist documentation.

As of unidist 0.5.0 there is support for a shared object store for MPI backend. The feature allows to improve performance in the workloads, where workers use same data multiple times by reducing data copies. You can enable the feature by setting the following environment variable:

```
export UNIDIST_MPI_SHARED_OBJECT_STORE=True
```

or turn it on in source code:

```
import unidist.config as unidist_cfg

unidist_cfg.MpiSharedObjectStore.put(True)
```

CHAPTER

THIRTEEN

ECOSYSTEM

There is a constantly growing number of users and packages using pandas to address their specific needs in data preparation, analysis and visualization. pandas is being used ubiquitously and is a good choise to handle small-sized data. However, pandas scales poorly and is non-interactive on moderate to large datasets. Modin provides a drop-in replacement API for pandas and scales computation across nodes and CPUs available. What you need to do to switch to Modin is just replace a single line of code.

```
# import pandas as pd
import modin.pandas as pd
```

While most packages can consume a pandas DataFrame and operate it efficiently, this is not the case with a Modin DataFrame due to its distributed nature. Thus, some packages may lack support for handling Modin DataFrame(s) correctly and, moreover, efficiently. Modin implements such methods as `__array__`, `__dataframe__`, etc. to facilitate other libraries to consume a Modin DataFrame. If you feel that a certain library can operate efficiently with a specific format of data, it is possible to convert a Modin DataFrame to the format preferred.

13.1 to_pandas

You can refer to pandas ecosystem page to get more details on where pandas can be used and what libraries it powers.

```
from modin.pandas.io import to_pandas

pandas_df = to_pandas(modin_df)
```

13.2 to_numpy

You can refer to NumPy ecosystem section of NumPy documentation to get more details on where NumPy can be used and what libraries it powers.

```
from modin.pandas.io import to_numpy

numpy_arr = to_numpy(modin_df)
```

13.3 to_ray

You can refer to Ray Data page to get more details on where Ray Dataset can be used and what libraries it powers.

```
from modin.pandas.io import to_ray

ray_dataset = to_ray(modin_df)
```

13.4 to_dask

You can refer to Dask DataFrame page to get more details on where Dask DataFrame can be used and what libraries it powers.

```
from modin.pandas.io import to_dask

dask_df = to_dask(modin_df)
```

CHAPTER

FOURTEEN

CONTACT

14.1 Slack

Join our Slack community to connect with Modin users and contributors, discuss, and ask questions about all things Modin-related.

14.2 Mailing List

General questions, potential contributors, and ideas can be directed to the developer mailing list. It is an open Google Group, so feel free to join anytime! If you are unsure about where to ask or post something, the mailing list is a good place to ask as well.

14.3 Issues

Bug reports and feature requests can be directed to the issues page of the Modin GitHub repo.

CHAPTER

FIFTEEN

SCALE YOUR PANDAS WORKFLOW BY CHANGING A SINGLE LINE OF CODE

Modin uses Ray, Dask or Unidist to provide an effortless way to speed up your pandas notebooks, scripts, and libraries. Unlike other distributed DataFrame libraries, Modin provides seamless integration and compatibility with existing pandas code. Even using the DataFrame constructor is identical.

```
import modin.pandas as pd
import numpy as np

frame_data = np.random.randint(0, 100, size=(2**10, 2**8))
df = pd.DataFrame(frame_data)
```

It is not necessary to know in advance the available hardware resources in order to use Modin. Additionally, it is not necessary to specify how to distribute or place data. Modin acts as a drop-in replacement for pandas, which means that you can continue using your previous pandas notebooks, *unchanged*, while experiencing a considerable speedup thanks to Modin, even on a single machine. Once you've changed your import statement, you're ready to use Modin just like you would pandas.

CHAPTER

SIXTEEN

INSTALLATION AND CHOOSING YOUR COMPUTE ENGINE

Modin can be installed from PyPI:

```
pip install modin
```

If you don't have Ray, Dask or Unidist installed, you will need to install Modin with one of the targets:

```
pip install "modin[ray]" # Install Modin dependencies and Ray to run on Ray
pip install "modin[dask]" # Install Modin dependencies and Dask to run on Dask
pip install "modin[mpi]" # Install Modin dependencies and MPI to run on MPI through␣
↪unidist
pip install "modin[all]" # Install all of the above
```

Modin will automatically detect which engine you have installed and use that for scheduling computation!

If you want to choose a specific compute engine to run on, you can set the environment variable `MODIN_ENGINE` and Modin will do computation with that engine:

```
export MODIN_ENGINE=ray  # Modin will use Ray
export MODIN_ENGINE=dask  # Modin will use Dask
export MODIN_ENGINE=unidist # Modin will use Unidist
```

If you want to choose the Unidist engine, you should set the additional environment variable `UNIDIST_BACKEND`, because currently Modin only supports MPI through unidist:

```
export UNIDIST_BACKEND=mpi # Unidist will use MPI backend
```

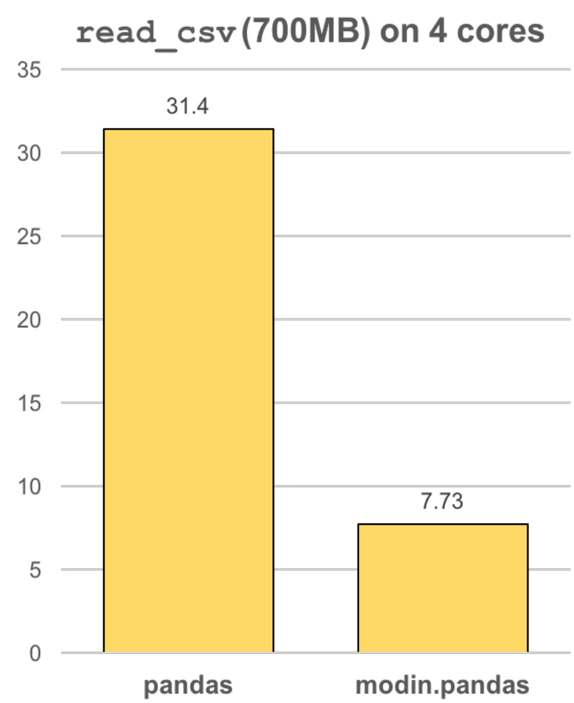
This can also be done within a notebook/interpreter before you import Modin:

```
import os

os.environ["MODIN_ENGINE"] = "ray"  # Modin will use Ray
os.environ["MODIN_ENGINE"] = "dask"  # Modin will use Dask

os.environ["MODIN_ENGINE"] = "unidist" # Modin will use Unidist
os.environ["UNIDIST_BACKEND"] = "mpi" # Unidist will use MPI backend

import modin.pandas as pd
```

CHAPTER

SEVENTEEN

FASTER PANDAS, EVEN ON YOUR LAPTOP



The `modin.pandas` DataFrame is an extremely light-weight parallel DataFrame. Modin transparently distributes the data and computation so that all you need to do is continue using the pandas API as you were before installing Modin. Unlike other parallel DataFrame systems, Modin is an extremely light-weight, robust DataFrame. Because it is so light-weight, Modin provides speed-ups of up to 4x on a laptop with 4 physical cores.

In pandas, you are only able to use one core at a time when you are doing computation of any kind. With Modin, you are able to use all of the CPU cores on your machine. Even in `read_csv`, we see large gains by efficiently distributing the work across your entire machine.

```python
import modin.pandas as pd

df = pd.read_csv("my_dataset.csv")
```

CHAPTER

EIGHTEEN

MODIN IS A DATAFRAME FOR DATASETS FROM 1MB TO 1TB+

We have focused heavily on bridging the solutions between DataFrames for small data (e.g. pandas) and large data. Often data scientists require different tools for doing the same thing on different sizes of data. The DataFrame solutions that exist for 1MB do not scale to 1TB+, and the overheads of the solutions for 1TB+ are too costly for datasets in the 1KB range. With Modin, because of its light-weight, robust, and scalable nature, you get a fast DataFrame at 1MB and 1TB+.

Modin is currently under active development. Requests and contributions are welcome!

If you are interested in learning more about Modin, please check out the *Getting Started* guide then refer to the *Developer Documentation* section, where you can find system architecture, internal implementation details, and other useful information. Also check out the Github to view open issues and make contributions.

INDEX

Symbols

D

E

F

G

O

P

S

U