modin Documentation

Release latest

Jul 16, 2019

Installation

Scale your pandas workflows by changing one line of code

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

Modin uses Ray 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.

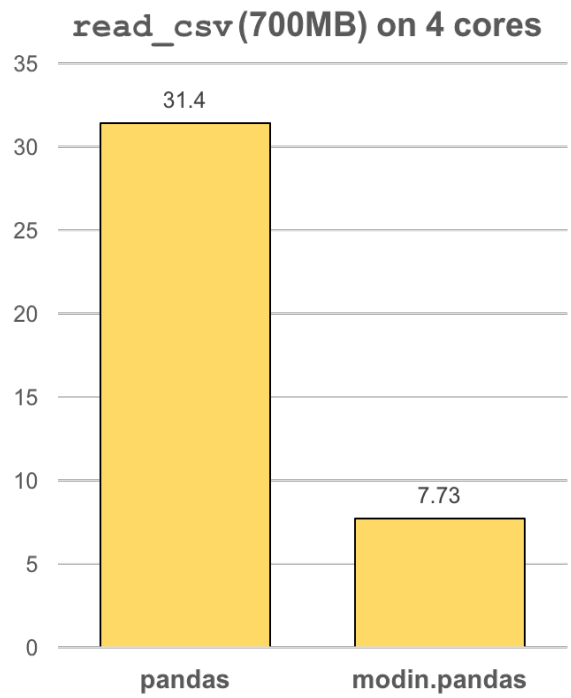
```
import modin.pandas as pd
import numpy as np

frame_data = np.random.randint(0, 100, size=(2**10, 2**8))
df = pd.DataFrame(frame_data)
```

To use Modin, you do not need to know how many cores your system has and you do not need to specify how to distribute the data. In fact, you can continue using your previous pandas notebooks while experiencing a considerable speedup from 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 1

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.

```
import modin.pandas as pd

df = pd.read_csv("my_dataset.csv")
```

CHAPTER 2

Modin is a DataFrame for datasets from 1KB 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 1KB 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 1KB and 1TB+.

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

2.1 Installation

There are a couple of ways to install Modin. Most users will want to install with `pip`, but some users may want to build from the master branch on the GitHub repo.

2.1.1 Installing with pip

Stable version

Modin can be installed with pip. To install the most stable release run the following:

```
pip install modin
```

Release candidates

Before most major releases, we will upload a release candidate to 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.

Installing specific dependency sets

Modin has a number of specific dependency sets for running Modin on different backends or for different functionalities of Modin. Here is a list of dependency sets for Modin:

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

2.1.2 Installing from the GitHub master branch

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

```
pip install git+https://github.com/modin-project/modin
```

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

2.1.3 Windows

For installation on Windows, we recommend using Windows Subsystem for Linux (WSL). This will allow you to use Linux commands on your Windows machine.

One of our dependencies is Ray. Ray is not yet supported natively on Windows, so in order to install it you need to use the WSL if you are on Windows.

Once you've installed WSL, you can install Modin in the WSL bash shell just like you would on Linux or Mac:

```
pip install modin
```

Once you've done this, Modin will be installed. However, it is important to note that you must execute *python*, *ipython* and *jupyter* from the WSL application.

2.1.4 Dependencies

Currently, Modin depends on pandas version 0.23.4. The API of pandas has a tendency to change some with each release, so we pin our current version to the most recent version to take advantage of the newest additions. This also typically means better performance and more correct code.

Modin also depends on Ray. Ray is a task-parallel execution framework for parallelizing new and existing applications with minor code changes. Currently, we depend on the most recent Ray release: https://pypi.org/project/ray/.

2.1.5 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:

```
git clone https://github.com/modin-project/modin.git
```

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

```
cd modin
pip install -e .
```

2.2 Using Modin

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

Currently, we have part of the pandas API implemented and are working toward full functional parity with pandas.

2.2.1 Using Modin on a Single Node

In order to use the most up-to-date version of Modin, please follow the instructions on the installation page

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

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

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

======================================================================
View the web UI at http://localhost:8889/notebooks/ray_ui36796.ipynb?
↪token=ac25867d62c4ae87941bc5a0ecd5f517dbf80bd8e9b04218
======================================================================
```

Once you have executed `import modin.pandas as pd`, you're ready to begin running your pandas pipeline as you were before.

2.2.2 APIs Supported

Please note, the API is not yet complete. For some methods, you may see the following:

```
NotImplementedError: To contribute to Modin, please visit github.com/modin-project/
↪modin.
```

We have compiled a list of currently supported methods.

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.

2.2.3 Using Modin on a Cluster (experimental)

Modin is able to utilize Ray's built-in autoscaled cluster. However, this usage is still under heavy development. To launch a Ray autoscaled cluster using Amazon Web Service (AWS), you can use the file *examples/cluster/aws_example.yaml* as the config file when launching an autoscaled Ray cluster. For the commands, refer to the autoscaler documentation.

We will provide a sample config file for private servers and other cloud service providers as we continue to develop and improve Modin's cluster support.

2.2.4 Advanced usage (experimental)

In some cases, it may be useful to customize your Ray environment. Below, we have listed a few ways you can solve common problems in data management with Modin by customizing your Ray environment. It is possible to use any of Ray's initialization parameters, which are all found in Ray's documentation.

```
import ray
ray.init()
import modin.pandas as pd
```

Modin will automatically connect to the Ray instance that is already running. This way, you can customize your Ray environment for use in Modin!

Exceeding memory (Out of core pandas)

Modin experimentally supports out of core operations. See more on the Out of Core documentation page.

Reducing or limiting the resources Modin can use

By default, Modin will use all of the resources available on your machine. It is possible, however, 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:

```
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).

2.2.5 Examples

You can find an example on our recent blog post or on the Jupyter Notebook that we used to create the blog post.

2.3 Out of Core in Modin (experimental)

If you are working with very large files or would like to exceed your memory, you may change the primary location of the DataFrame. If you would like to exceed memory, you can use your disk as an overflow for the memory. This API is experimental in the context of Modin. Please let us know what you think!

2.3.1 Install Modin out of core

Modin now comes with all the dependencies for out of core functionality by default! See the installation page for more information on installing Modin.

2.3.2 Starting Modin with out of core enabled

Out of core is detected from an environment variable set in bash.

```
export MODIN_OUT_OF_CORE=true
```

We also set up a way to tell Modin how much memory you'd like to use. Currently, this only accepts the number of bytes. This can only exceed your memory if you have enabled `MODIN_OUT_OF_CORE`.

[Optional]: Set a limit on the out of core space for Modin

Warning: Make sure you have enough space in your disk for however many bytes you request for your DataFrame

This limits the amount of memory that Modin can use.

Here is how you set `MODIN_MEMORY`:

```
export MODIN_MEMORY=200000000000 # Set the number of bytes to 200GB
```

The default for Modin is 8x the memory on the machine.

2.3.3 Running an example with out of core

Before you run this, please make sure you follow the instructions listed above.

```
import modin.pandas as pd
import numpy as np
frame_data = np.random.randint(0, 100, size=(2**20, 2**8)) # 2GB each
df = pd.DataFrame(frame_data).add_prefix("col")
big_df = pd.concat([df for _ in range(20)]) # 20x2GB frames
print(big_df)
nan_big_df = big_df.isna() # The performance here represents a simple map
print(big_df.apply(lambda col: col.sum())) # apply along an entire axis (columns in␣
↪this case)
```

This example creates a 40GB DataFrame from 20 identical 2GB DataFrames and performs various operations on them. Feel free to play around with this code and let us know what you think!

2.4 Modin Supported Methods

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 master branch, and not necessarily correct as of the most recent release. In order to install the latest version of Modin, follow the directions found on the installation page.

To see these lists, please visit the pages for your interested engine/backend combination:

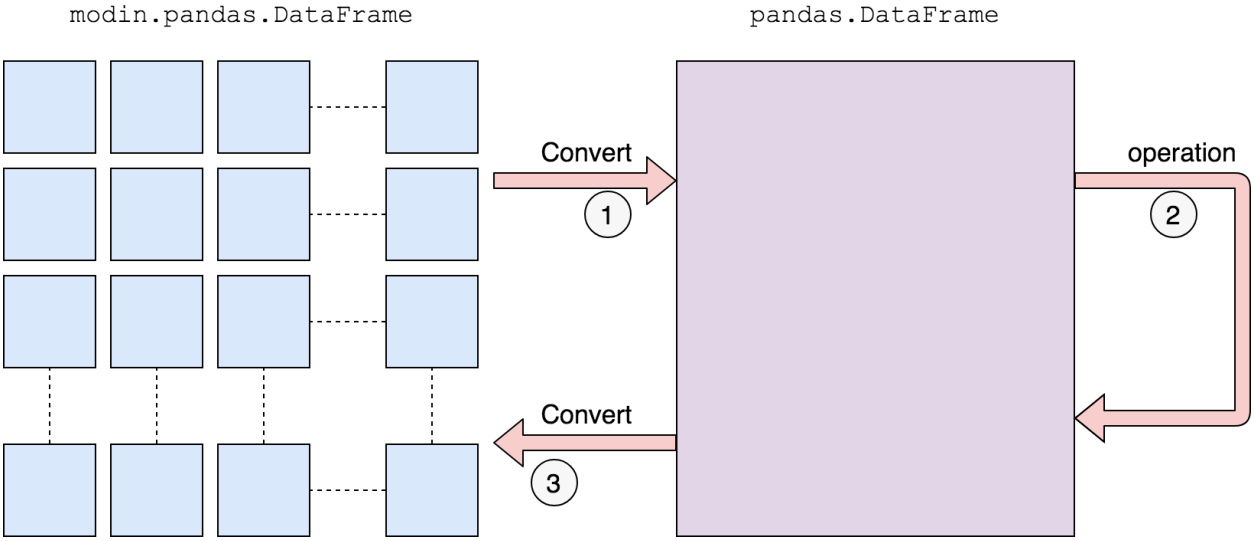
- Pandas on Ray
- Pandas on Dask
- Pyarrow on Ray

2.4.1 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.

2.4.2 Defaulting to pandas

The remaining unimplemented methods default to pandas. This allows users to continue using Modin even though their workloads contain functions not yet implemented in Modin. 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 pandas. Once the pandas operation has completed, we convert the DataFrame back into a partitioned Modin DataFrame. This way, operations performed after something defaults to pandas will be optimized with Modin.

2.5 Pandas on Ray

This section describes usage related documents for the Pandas on Ray component of Modin.

Currently, Modin support ~71% of the pandas API. 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.

Modin uses Pandas on Ray by default, but if you wanted to be explicit, you could set the following environment variables:

```
export MODIN_ENGINE=ray
export MODIN_BACKEND=pandas
```

2.6 Pandas on Dask

The Dask engine and documentation could use your help! Consider opening a pull request or an issue to contribute or ask clarifying questions.

2.7 Pyarrow on Ray

Coming Soon!

2.8 Contributing

2.8.1 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.

2.8.2 Development Dependencies

We recommend doing development in a virtualenv, 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:

```
pip install -U black flake8 pytest feather-format lxml openpyxl \
    xlrd numpy matplotlib --ignore-installed
```

2.8.3 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/
```

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

```
flake8 .
```

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

2.8.4 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.

2.8.5 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.

2.8.6 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.

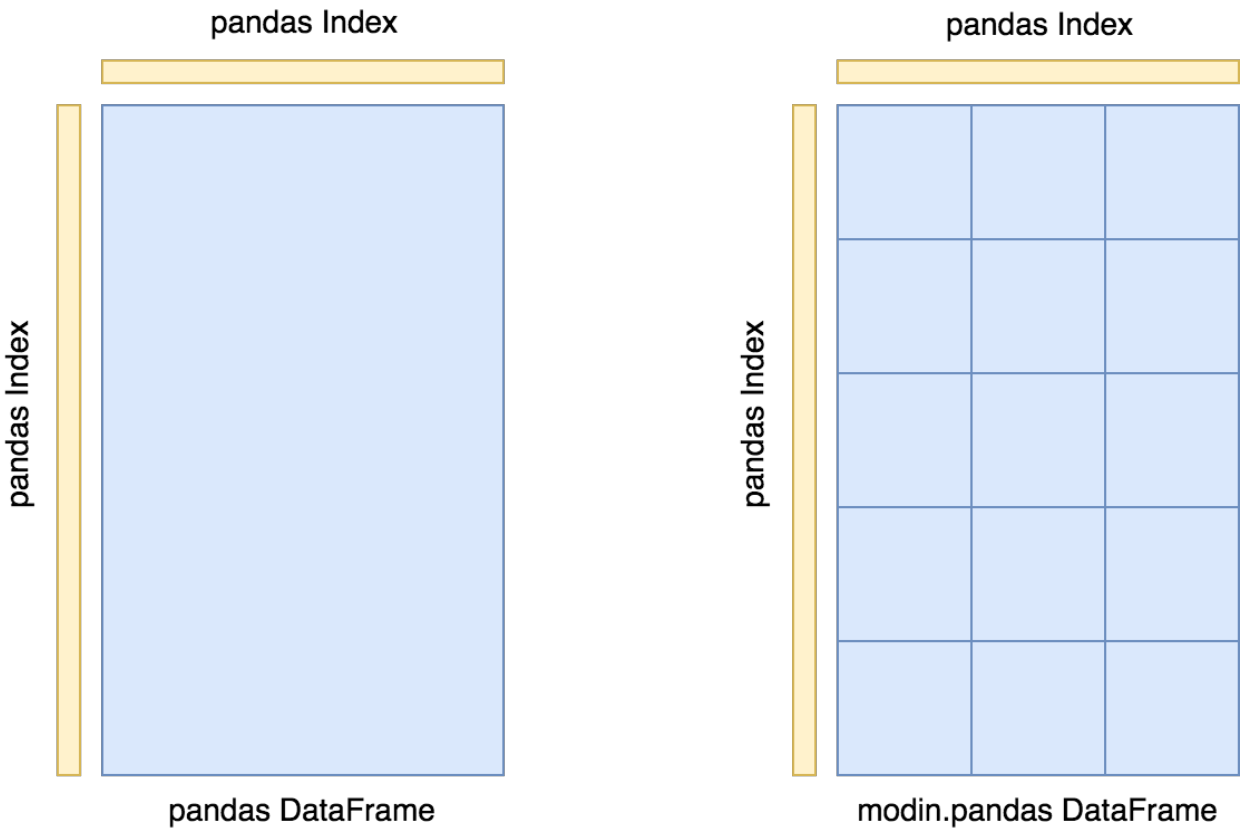
More docs on this coming soon...

2.9 Architecture

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

2.9.1 DataFrame Partitioning

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



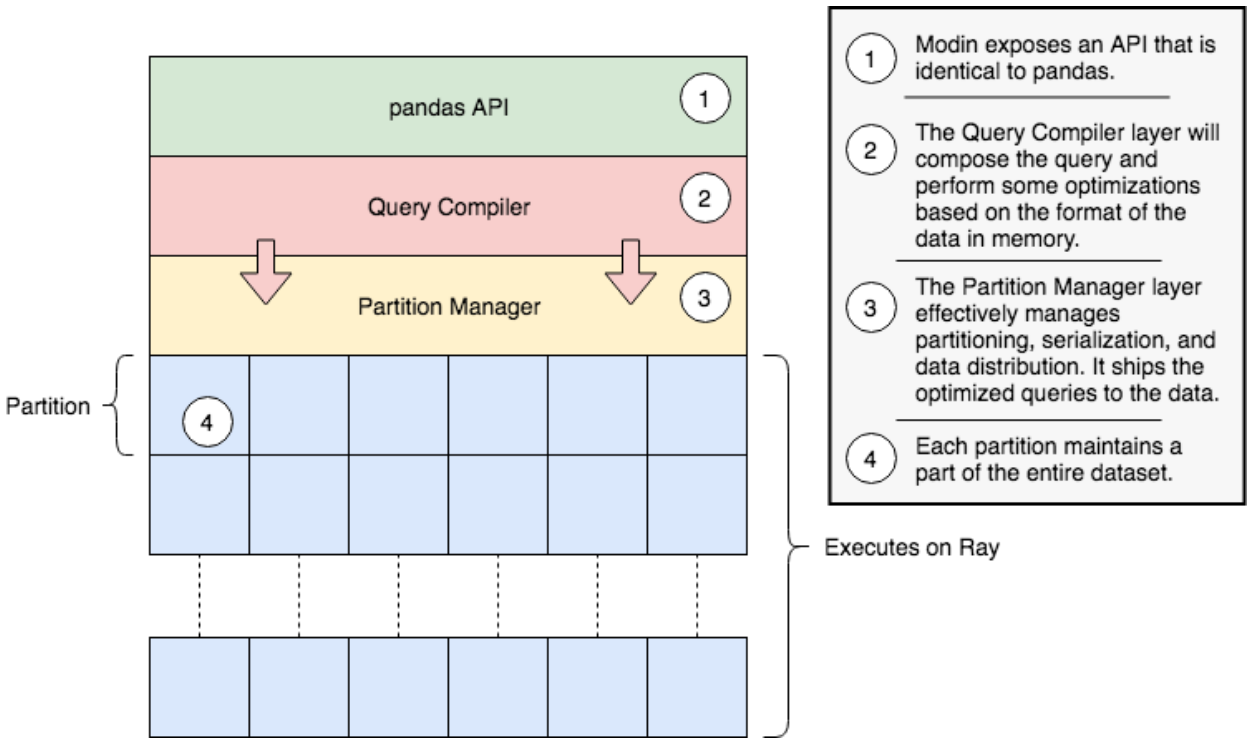
Currently, each partition's memory format is a pandas DataFrame. In the future, we will support additional in-memory formats for the backend, namely Arrow tables.

Index

We currently use the `pandas.Index` object for both indexing columns and rows. In the future, we will implement a distributed, pandas-compatible Index object in order remove this scaling limitation from the system. It does not start to become a problem until you are operating on more than 10's of billions of columns or rows, so most workloads will not be affected by this scalability limit. **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.

2.9.2 System Architecture

The figure below outlines the general architecture for the implementation of Modin.



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

Query Compiler

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

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

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

Partition Manager

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

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

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

Partition

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

Supported Execution Frameworks and Memory Formats

This is the list of execution frameworks and memory formats supported in Modin. If you would like to contribute a new execution framework or memory format, please see the documentation page on Contributing.

- **Pandas on Ray**
  - Uses the Ray execution framework.
  - The in-memory format is a pandas DataFrame.
- Coming Soon...

2.10 Pandas on Ray

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

2.10.1 Ray-specific optimizations

Ray is a high-performance task-parallel execution framework with Python and Java APIs. It uses the plasma store and serialization formats of Apache Arrow.

Normally, in order to start a Ray cluster, a user would have to use some of Ray's command line tools or call `ray.init`. Modin will automatically call `ray.init` for users who are running on a single node. Otherwise a Ray cluster must be setup before calling `import modin.pandas as pd`. More about running Modin in a cluster can be found in the using Modin documentation.

Serialization of tasks and parameters

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

Memory Management

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

2.10.2 Pandas-specific optimizations

Pandas on Ray can take advantage of some of the properties of pandas in order to optimize for both memory footprint and runtime.

Indexing

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

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

2.11 Troubleshooting

We hope your experience with Modin is bug-free, but there are some quirks about Modin that may require troubleshooting.

2.11.1 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.

Error During execution: `ArrowIOError: Broken Pipe`

One of the more frequently encountered issues is an `ArrowIOError: Broken Pipe`. This error can happen in a couple of different ways. One of the most common ways this is encountered is from pressing **CTRL + C** sending a `KeyboardInterrupt` to Modin. In Ray, when a `KeyboardInterrupt` is sent, Ray will shutdown. This causes the `ArrowIOError: Broken Pipe` because there is no longer an available plasma store for working on remote tasks. This is working as intended, as it is not yet possible in Ray to kill a task that has already started computation.

The other common way this `Error` is encountered is to let your computer go to sleep. As an optimization, Ray will shutdown whenever the computer goes to sleep. This will result in the same issue as above, because there is no longer a running instance of the plasma store.

Solution

Restart your interpreter or notebook kernel.

Avoiding this Error

Avoid using `KeyboardInterrupt` and keeping your notebook or terminal running while your machine is asleep. If you do `KeyboardInterrupt`, you must restart the kernel or interpreter.

Error during execution: `ArrowInvalid: Maximum size exceeded (2GB)`

Encountering this issue means that the limits of the Arrow plasma store have been exceeded by the partitions of your data. This can be encountered during shuffling data or operations that require multiple datasets. This will only affect extremely large DataFrames, and can potentially be worked around by setting the number of partitions. This error is being actively worked on and should be resolved in a future release.

Solution

```
import modin.pandas as pd
pd.DEFAULT_NPARTITIONS = 2 * pd.DEFAULT_NPARTITIONS
```

This will set the number of partitions to a higher count, and reduce the size in each. If this does not work for you, please open an issue.

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.

2.12 Contact

2.12.1 Mailing List

https://groups.google.com/forum/#!forum/modin-dev

General questions, potential contributors, and ideas should 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.

2.12.2 Issues

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

Bug reports and feature requests should be directed to the issues page of the Modin GitHub repo.

2.13 Modin SQL API

Modin's SQL API is currently a conceptual plan, Coming Soon!

2.13.1 Plans for future development

Our plans with the SQL API for Modin are to create an interface that allows you to intermix SQL and pandas operations without copying the entire dataset into a new structure between the two. This is possible due to the architecture of Modin. Currently, Modin has a query compiler that acts as an intermediate layer between the query language (e.g. SQL, pandas) and the execution (See architecture documentation for details).

We have implemented a simple example that can be found below. Feedback welcome!

```
>>> import modin.sql as sql
>>>
>>> conn = sql.connect("db_name")
>>> c = conn.cursor()
>>> c.execute("CREATE TABLE example (col1, col2, column 3, col4)")
>>> c.execute("INSERT INTO example VALUES ('1', 2.0, 'A String of information', True)
↪")
  col1  col2                 column 3  col4
0    1   2.0  A String of information  True

>>> c.execute("INSERT INTO example VALUES ('6', 17.0, 'A String of different␣
↪information', False)")
  col1  col2                           column 3   col4
0    1   2.0            A String of information   True
1    6  17.0  A String of different information  False
```