PyCharm

The Python IDE for data science and web development

Download Follow ▼

Data Science

Polars vs. pandas: What's the Difference?

Jodie Burchell

July 4, 2024

Read this post in other languages:
Français, 日本語, 한국어, 简体中文

If you've been keeping up with the advances in Python dataframes in the past year, you couldn't help hearing about Polars, the powerful dataframe library designed for working with large datasets.

Polars vs. pandas: What’s the Difference?

Unlike other libraries for working with large datasets, such as Spark, Dask, and Ray, Polars is designed to be used on a single machine, prompting a lot of comparisons to pandas. However, Polars differs from pandas in a number of important ways, including how it works with data and what its optimal applications are. In the following article, we’ll explore the technical details that differentiate these two dataframe libraries and have a look at the strengths and limitations of each.

If you’d like to hear more about this from the creator of Polars, Ritchie Vink, you can also see our interview with him below!

Why use Polars over pandas?

In a word: performance. Polars was built from the ground up to be blazingly fast and can do common operations around 5–10 times faster than pandas. In addition, the memory requirement for Polars operations is significantly smaller than for pandas: pandas requires around 5 to 10 times as much RAM as the size of the dataset to carry out operations, compared to the 2 to 4 times needed for Polars.

You can get an idea of how Polars performs compared to other dataframe libraries here. As you can see, Polars is between 10 and 100 times as fast as pandas for common operations and is actually one of the fastest DataFrame libraries overall. Moreover, it can handle larger datasets than pandas can before running into out-of-memory errors.

Why is Polars so fast?

These results are extremely impressive, so you might be wondering: How can Polars get this sort of performance while still running on a single machine? The library was designed with performance in mind from the beginning, and this is achieved through a few different means.

Written in Rust

Or
th

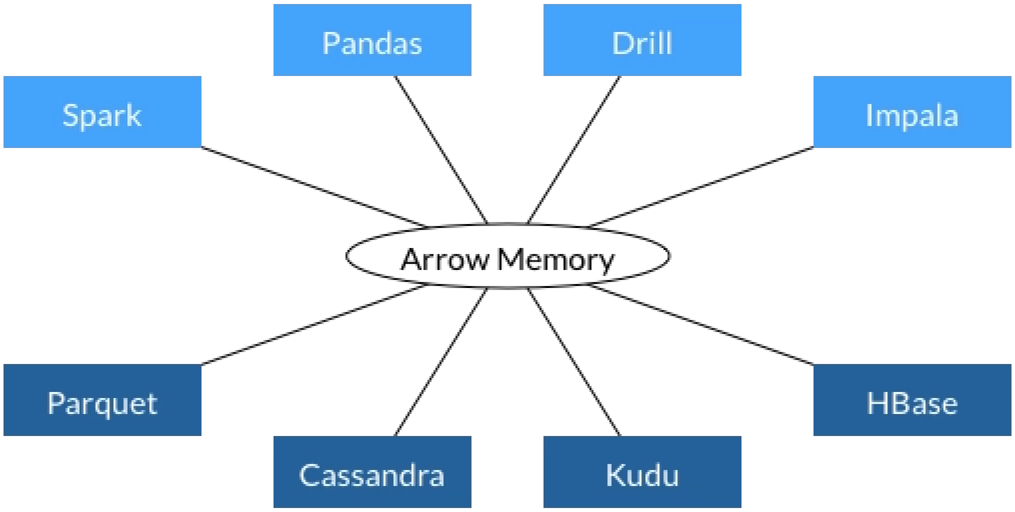
these being NumPy. While NumPy's core is written in C, it is still hamstrung by inherent problems with the way Python handles certain types in memory, such as strings for categorical data, leading to poor performance when handling these types (see this fantastic blog post from Wes McKinney for more details).

One of the other advantages of using Rust is that it allows for safe concurrency; that is, it is designed to make parallelism as predictable as possible. This means that Polars can safely use all of your machine's cores for even complex queries involving multiple columns, which led Ritchie Vink to describe Polar's performance as "embarrassingly parallel". This gives Polars a massive performance boost over pandas, which only uses one core to carry out operations. Check out this excellent talk by Nico Kreiling from PyCon DE this year, which goes into more detail about how Polars achieves this.

Based on Arrow

Another factor that contributes to Polars' impressive performance is Apache Arrow, a language-independent memory format. Arrow was actually co-created by Wes McKinney in response to many of the issues he saw with pandas as the size of data exploded. It is also the backend for pandas 2.0, a more performant version of pandas released in March of this year. The Arrow backends of the libraries do differ slightly, however: while pandas 2.0 is built on PyArrow, the Polars team built their own Arrow implementation.

One of the main advantages of building a data library on Arrow is interoperability. Arrow has been designed to standardize the in-memory data format used across libraries, and it is already used by a number of important libraries and databases, as you can see below.



Th

dif

avoids the need to serialize and deserialize the data). It is also more memory-efficient, as two processes can share the same data without needing to make a copy. As serialization/deserialization is estimated to represent 80–90% of the computing costs in data workflows, Arrow's common data format lends Polars significant performance gains.

Arrow also has built-in support for a wider range of data types than pandas. As pandas is based on NumPy, it is excellent at handling integer and float columns, but struggles with other data types. In contrast, Arrow has sophisticated support for datetime, boolean, binary, and even complex column types, such as those containing lists. In addition, Arrow is able to natively handle missing data, which requires a workaround in NumPy.

Finally, Arrow uses columnar data storage, which means that, regardless of the data type, all columns are stored in a continuous block of memory. This not only makes parallelism easier, but also makes data retrieval faster.

Query optimization

One of the other cores of Polars' performance is how it evaluates code. Pandas, by default, uses *eager execution*, carrying out operations in the order you've written them. In contrast, Polars has the ability to do both eager and *lazy execution*, where a query optimizer will evaluate all of the required operations and map out the most efficient way of executing the code. This can include, among other things, rewriting the execution order of operations or dropping redundant calculations. Take, for example, the following expression to get the mean of column `Number1` for each of the categories "A" and "B" in `Category`.

```
(
df
.groupby(by = "Category").agg(pl.col("Number1").mean())
.filter(pl.col("Category").is_in(["A", "B"]))
)
```

If this expression is eagerly executed, the `groupby` operation will be unnecessarily performed for the whole DataFrame, and then filtered by `Category`. With lazy execution, the DataFrame can be filtered and `groupby` performed on only the required data.

Expressive API

Finally, Polars has an extremely expressive API, meaning that basically any operation you want to perform can be expressed as a Polars method. In contrast, more complex operations in pandas often need to be passed to the `apply` method as a lambda expression. The problem

wi
ex
a columnar level and take advantage of another form of parallelism called SIMD.

When should you stick with pandas?

All of this sounds so amazing that you’re probably wondering why you would even bother with pandas anymore. The main reason is that pandas, having grown up with the rest of the Python data science ecosystem, still has the greatest interoperability with other packages that form part of the machine learning pipeline.

However, Polars is catching up, and its interoperability with these packages grows month by month. Polars is now compatible with a wide number of plotting libraries, including `plotly` , `matplotlib` (except when using Series), `seaborn` , `altair` and `hvplot` , meaning it now works well as an exploratory data analysis library.

It is also now possible to use Polars DataFrames as part of machine learning and deep learning pipelines. As of `scikit-learn` release 1.4.0., it is possible to output transformers as Polars DataFrames. It is also now possible to convert Polars DataFrames to PyTorch data types, including a PyTorch Tensor, a PolarsDataset (a frame-specialized TensorDataset), or a dictionary of Tensors. This can be achieved in Polars by calling the `to_torch` method on a DataFrame.

All of these are open source projects, and many hours have been put in by their maintainers to make these packages work together. A huge thank you to everyone working on these projects – it is certainly an exciting time in data science! (Also, thanks to Ritchie Vink for highlighting these advances!)

Tooling for Polars and pandas

After all of this, I am sure you are eager to try Polars yourself! PyCharm Professional for Data Science offers excellent tooling for working with both pandas and Polars in Jupyter notebooks. In particular, pandas and Polars DataFrames are displayed with interactive functionality, which makes exploring your data much quicker and more comfortable.

Some of my favorite features include the ability to scroll through all rows and columns of the DataFrame without truncation, get aggregations of DataFrame values in one click, and export the DataFrame in a huge range of formats (including Markdown!).

If you’re not yet using PyCharm, you can try it with a 30-day trial by following the link below.