Polars: A Lightning-Fast DataFrame Library for Python and Rust

§

Made with Obsidian

Type deep-dive | Category data-science | Technologies Python | Website Post Link

Of all the libraries belonging to any Data Scientist's toolbox, `Pandas` may be the most important one; it's built on top of the `NumPy` package and provides data structures and methods tailored for data manipulation and analysis with a syntax similar to SQL queries.

The thing about `Pandas` is that it does not support parallelization natively, thus limiting its computation capabilities; some `Pandas` tasks can be parallelized by using `Dask` or other libraries, but this requires external handling and is not always the best solution.

Additionally, `Pandas` does not support *lazy execution*; this means that the code is run directly, and its results are returned immediately, which can result in running unnecessary code.

All these aspects make `Pandas` still attractive for relatively small computation tasks but somewhat unattractive for more extensive data set manipulation.

Meet `Polars`, a DataFrame library built on `Rust` from the ground up, presented in two flavours: A `Python` and a `Rust` API.

In this Deep Dive, we'll review `Polars` in detail using the `Polars` API for `Python`. We'll discuss its installation, core functionalities, basic syntax, some data transformations, reading and writing from and to different file formats, and more.

We'll be using Python scripts which can be found in the Deep Dive Repo.

§

Table of Contents

- Preface
- Preparing our environment
- Polars data structures
- Eager execution
- Lazy execution
- Reading and writing multiple file formats
  - Writing
  - Reading
- Basic operations
  - Exploratory methods

    - Indexing, selecting and filtering
        - Select
        - Filter
        - Filtering with multiple conditions
        - Filtering with advanced operators
    - Aggregations
    - Joins
    - Concatenations
    - Creating new columns
- Multi-threaded execution
- Schemas and data types
- Conclusions
- References

§

Preface

`Polars` is a DataFrame library/in-memory query engine written in `Rust`. It's built upon the safe Arrow2 implementation of the Apache Arrow specification, enabling efficient resource use and processing performance. By doing so, it also integrates seamlessly with other tools in the Arrow ecosystem.

Unlike tools such as `Dask`, which try to parallelize existing single-threaded libraries like `Numpy` and `Pandas`, `Polars` is designed for parallelization, resulting in breakneck processing speeds by default.

A `groupby` task performed on a 5GB dataset resulted in the following execution times:

Method	Version	Date Executed	Execution Time [s]
`DataFrames.jl`	1.1.1	May 15, 2021	9
`Polars`	0.8.8	June 30, 2021	11
`cuDF`	0.19.2	May 31, 2021	17
`Spark`	3.1.2	May 31, 2021	34
`Pandas`	1.2.5	June 30, 2021	70
`Arrow`	4.0.1	May 31, 2021	212

Table 1. Groupby Execution Times On 5 GB Data Set, H2O AI

A `join` task performed on a 5GB dataset resulted in the following execution times:

Method	Version	Date Executed	Execution Time [s]
`Polars`	0.8.8	June 30, 2021	43
`Spark`	3.1.2	May 31, 2021	332
`DataFrames.jl`	1.1.1	June 3, 2021	349
`Pandas`	1.2.5	June 30, 2021	628
`cuDF`	0.19.2	May 31, 2021	internal error
`Arrow`	4.0.1	May 31, 2021	not yet implemented

TABLE 2. JOIN EXECUTION TIMES ON 5 GB DATA SET, H2O AI

The full benchmark can be consulted here.

`Polars` for `Python` exposes a complete `Python` API, including the full set of features to manipulate DataFrames using an expression language similar to `Pandas`. It also has two different APIs:

- A lazy API
- An eager API

With *eager execution*, the code is run as soon as it's encountered; results are returned immediately. With *lazy execution*, the code is run until the result is required.

§

Preparing our environment

`Polars` is offered as a `Python` and a `Rust` package. In this segment, we'll only review the Python flavour; in a future iteration, we might review its Rust counterpart.

We're going to use the `Polars` package. More information about this package can be found in the `Polars` Official Web Page, in the `Polars` GitHub Repo, or the `Polars` Official Documentation for Python.

If we don't yet have it, we can install it:

CODE

```
pip install polars
```

We will also install some additional libraries, which are not directly related to `Polars` but will be helpful for some bonus content ahead.

CODE

```
pip install geopandas
pip install geopy
pip install folium
```

The convention is to import `Polars` using the `pl` alias, but we can select any alias we find more convenient. For our case, we'll be using the preferred alias. We'll also import some other modules which will come in handy:

CODE

```
import polars as pl
import pandas as pd
import numpy as np
import pyarrow
import os
import glob
from datetime import datetime

# Import bonus modules
import folium
from folium.plugins import FastMarkerCluster
```

As of the writing of this article, the `Polars` version downloaded is 0.16.9. We can confirm this by using the `__version__` method:

Code

```
print(pl.__version__)
```

We will also use the Airbnb Prices in European Cities data set by The Devastator. The complete set has 20 files, one for each European city.

We can first create a new folder, `datasets`, inside our project folder. We can then download the entire set as a `.zip` file, extract its contents, and move them to the newly created folder.

The `datasets` folder will contain 20 files weighing 10.2MB.

We can also create an outputs directory, where we will store our written files:

Code

```
mkdir datasets, outputs
```

We will define both directories as variables inside our script:

Code

```
rDir = 'datasets/'
wDir = 'outputs/'
```

With everything ready, we can now proceed to load our data sets and perform some basic operations.

§

Polars data structures

Similar to `Pandas`, `Polars` has two main data structures:

- `Series`: One-dimensional.

- `DataFrame` (With a `LazyFrame` variation for *lazy execution*): Can be one or two-dimensional.

We can define a `series` object by enclosing the values in square brackets `[]`:

Code

```
# Declare series
se = pl.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Check type
type(se)

# Print object
se
```

Output

```
polars.internals.series.series.Series

shape: (10,)
Series: '' [i64]
[
    1
    2
    3
    4
    5
    6
    7
    8
    9
    10
]
```

We can define a DataFrame object by enclosing our set of entries in curly brackets `{}`. Each dictionary key will correspond to a column name and each value to the column entries.

Code

```
# Declare DataFrame
df = pl.DataFrame({'name' : ['Jack', 'Charles', 'Clarice'],
                   'surname' : ['Kerouac', 'Bukowski', 'Lispector'],
                   'birth' : [datetime(1922, 3, 12),
                              datetime(1920, 8, 16),
                              datetime(1920, 12, 10)]
                   })

# Print object
df
```

Output

```
shape: (3, 3)
+---------+-----------+--------------------+
| name    | surname   | birth              |
| ---     | ---       | ---                |
| str     | str       | datetime[μs]       |
+==========================================+
| Jack    | Kerouac   | 1922-03-12 00:00:00 |
| Charles | Bukowski  | 1920-08-16 00:00:00 |
| Clarice | Lispector | 1920-12-10 00:00:00 |
+---------+-----------+--------------------+
```

§

Eager execution

We will start by executing `Polar` commands using *eager execution*. This is the default method and will run our code upon calling.

We can read one of our downloaded `.csv` files:

Code

```
df = pl.read_csv(os.path.join(rDir, 'berlin_weekends.csv'))
```

This method will read our file into a `polars.DataFrame` object:

Code

```
type(df)
```

Output

```
polars.internals.dataframe.frame.DataFrame
```

§

Lazy execution

As mentioned earlier, *lazy* operations don't execute until we call `collect`. This allows `Polars` to optimize/reorder the query, which may lead to faster queries or fewer type errors.

There are two main ways for *lazy-reading* a `.csv` file in `Polars`:

- Using `pl.scan_csv()`.
- Using `pl.read_csv().lazy()`.

Both methods perform the same operation; the main difference is that the first lazy-loads by default, while the second includes the `.lazy()` method to specify that we're *lazy-loading*.

We can read a `.csv` file using either of the two methods:

Code

```
# Reading a csv file using pl.scan_csv()
df_s = pl.scan_csv(os.path.join(rDir, 'berlin_weekends.csv'))

# Reading a csv file using pl.read_csv().lazy()
df_l = pl.read_csv(os.path.join(rDir, 'berlin_weekends.csv')).lazy()
```

As opposed to *eager execution*, this method will read our file into a `polars.LazyFrame` object:

Code

```
type(df_s), type(df_l)
```

Output

```
(polars.internals.lazyframe.frame.LazyFrame,
 polars.internals.lazyframe.frame.LazyFrame)
```

If we try to get the head of our object, we will actually be presented with its memory location, and not the first records.

Code

```
df_s.head()
```

Output

```
<polars.LazyFrame object at 0x22F0CCFFF50>
```

We can display the object graph, which is a diagram of how the execution will take place upon calling `collect`.

Code

```
df_s.show_graph()
```

Output

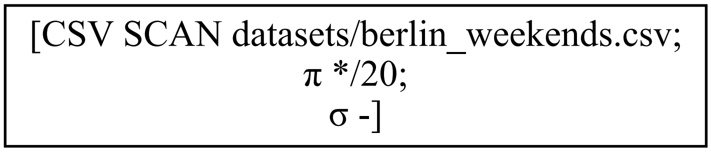

Figure 1: Execution Plan Graph For Lazy DataFrame Reading

We can include additional transformation steps to our object:

Code

```
df_s_filtered = (df_s.filter(pl.col("bedrooms") >= 2).
                 select(pl.col("metro_dist")).
                 sort("metro_dist")
                 )
```

And view its graph again:

Code

```
df_s_filtered.show_graph()
```

Output

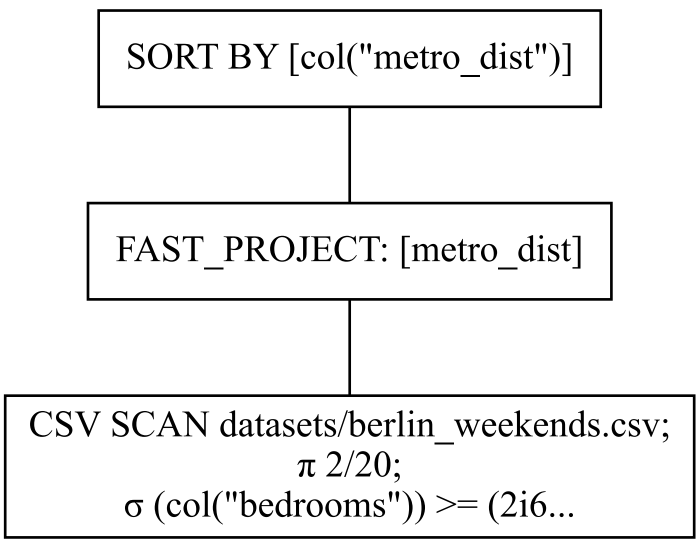

Figure 2: Execution Plan Graph For Several Lazy Transformations

We can see that the additional steps were added and are ready to be executed upon collecting our object.

We can also view this same information in text format:

Code

```
df_s_filtered.describe_optimized_plan()
```

Output

```
'SORT BY [col("metro_dist")]\n    FAST_PROJECT: [metro_dist]\n      CSV SCAN
datasets/berlin_weekends.csv\n      PROJECT 2/20 COLUMNS\n      SELECTION: [(col("bedrooms")) >=
(2i64)]\n'
```

Which is, of course, less neat than the previous graphical method.

We can finally call `collect()` :

Code

```
df_s_filtered.collect()
```

Output

```
shape: (124, 1)
┌────────────┐
│ metro_dist │
│ ---        │
│ f64        │
╞════════════╡
│ 0.088494   │
│ 0.110308   │
│ 0.12578    │
│ 0.131641   │
│ ...        │
│ 2.87661    │
│ 4.512803   │
│ 5.324563   │
│ 9.598773   │
└────────────┘
```

§

Reading and writing multiple file formats

1. Writing

As with `Pandas` , `Polars` can write to multiple file formats, the most common ones being:

- `.avro`
- `.csv`

- `.ipc`
- `.json`
- `.parquet`

To illustrate some examples, we will read our entire `weekdays` data set into a DataFrame object, and then write to the file formats above:

Code

```
# Read the entire weekdays data set
weekdays_files = glob.glob(os.path.join(rDir, "*weekdays.csv"))

weekdays_list = []

for filename in weekdays_files:
    city = re.search('datasets\\\(.*)_weekdays.csv', filename).group(1)
    df_weekdays = (pl.read_csv(filename).
                   drop(['']).
                    with_columns(pl.lit(city).
                    alias('city'))
                    )

    weekdays_list.append(df_weekdays)

df_weekdays = pl.concat(weekdays_list,
                        rechunk = True)

df_weekdays.shape

round(df_weekdays.estimated_size(unit='mb'), 4)
```

We should end up with a `Polars` DataFrame object of shape `(25500, 20)`:

Output

```
(25500, 20)

3.7998
```

Let us explain in detail by writing the pseudocode for the steps performed:

- Declare a list of `weekday` data set paths using the `glob.glob()` method.
- Create an empty DataFrame list `weekdays_list`.
- Iterate over the list.
- Extract the city using RegEx.
- Read each file using `pl.read_csv(filename)`.
- Drop the first column, which represents the index.
- Assign a new column to each DataFrame object containing its city name using `pl.lit(city).alias('city')`.
- Append each DataFrame to the DataFrame list `weekdays_list`.
- Concatenate all DataFrames in `weekdays_list`, passing `rechunk = True` as argument (*make sure that all data is in contiguous memory*).
- Get the object's shape.

- Get the object's estimated size in `mb` rounded to 4 decimal places.

Now, we can write our DataFrame to different file formats. The general syntax is `df.write_formatname(dir, args)`:

Code

```
# Write to csv
df_weekdays.write_csv(os.path.join(wDir, 'weekdays.csv'))

# Write to Parquet non-partitioned
df_weekdays.write_parquet(os.path.join(wDir, 'weekdays.parquet'))

# Write to Avro
df_weekdays.write_avro(os.path.join(wDir, 'weekdays.avro'))

# Write to JSON
df_weekdays.write_json(os.path.join(wDir, 'weekdays.json'))
```

2. Reading

Conversely, `Polars` can read all the file formats we wrote earlier. We'll skip the `.csv` file format since we already reviewed it. For the other cases, we can use the `pl.read_formatname()` syntax:

Code

```
# Write to csv
df_weekdays_csv = pl.read_csv(os.path.join(wDir, 'weekdays.csv'))

# Write to Parquet non-partitioned
df_weekdays_parquet = pl.read_parquet(os.path.join(wDir, 'weekdays.parquet'))

# Write to Avro
df_weekdays_avro = pl.read_avro(os.path.join(wDir, 'weekdays.avro'))

# Write to JSON
df_weekdays_json = pl.read_json(os.path.join(wDir, 'weekdays.json'))
```

We can confirm that our files were read successfully by selecting a given column and getting each object's head:

Code

```
df_weekdays_csv['realSum'].head(10)
df_weekdays_parquet['realSum'].head(10)
df_weekdays_avro['realSum'].head(10)
df_weekdays_json['realSum'].head(10)
```

Output

```
shape: (10,)
Series: 'realSum' [f64]
[
    194.033698
    344.245776
    264.101422
    433.529398
    485.552926
    552.808567
    215.124317
    2771.307384
    1001.80442
    276.521454
]

shape: (10,)
Series: 'realSum' [f64]
[
    194.033698
    344.245776
    264.101422
    433.529398
    485.552926
    552.808567
    215.124317
    2771.307384
    1001.80442
    276.521454
]

shape: (10,)
Series: 'realSum' [f64]
[
    194.033698
    344.245776
    264.101422
    433.529398
    485.552926
    552.808567
    215.124317
    2771.307384
    1001.80442
    276.521454
]

shape: (10,)
Series: 'realSum' [f64]
[
    194.033698
    344.245776
    264.101422
    433.529398
```

```
    485.552926
    552.808567
    215.124317
    2771.307384
    1001.80442
    276.521454
]
```

§

Basic operations

1. Exploratory methods

We can use a wide range of exploratory methods to take a first look at our data. We can display our DataFrame's shape, columns and first ten entries for the `realSum` column:

Code

```
df.shape
df.columns
df['realSum'].head(10)
df['realSum'].tail(10)
```

Output

```
(1200, 20)

['',
 'realSum',
 'room_type',
 'room_shared',
 'room_private',
 'person_capacity',
 'host_is_superhost',
 'multi',
 'biz',
 'cleanliness_rating',
 'guest_satisfaction_overall',
 'bedrooms',
 'dist',
 'metro_dist',
 'attr_index',
 'attr_index_norm',
 'rest_index',
 'rest_index_norm',
 'lng',
 'lat']

shape: (10,)
Series: 'realSum' [f64]
[
    185.799757
    387.49182
    194.914462
    171.777134
    207.768533
    162.428718
    521.875292
    155.417407
    171.777134
    147.237543
]

shape: (10,)
Series: 'realSum' [f64]
[
    162.428718
    231.840703
    127.605871
    175.049079
    156.585959
    84.83687
    134.617182
    134.617182
    160.091614
    359.680284
]
```

We can notice some interesting details:

- The `df.columns` method returns a `list`, as opposed to `Pandas` which returns a `pandas.core.indexes.base.Index` object.
- The `df.head()` method returns a `polars.internals.series.series.Series` object, similar to `Pandas`, which returns a `pandas.core.series.Series`.
- The `df.head()` method also returns the column data type, which in the case of `realSum` is `float64`.

We can also perform a statistical description:

Code

```
df.describe()
```

Output

```
+------------+------------+-------------+
| describe   | column_0   | realSum     |
| ---        | ---        | ---         |
| str        | f64        | f64         |
+=======================================+
| count      | 1200.0     | 1200.0      |
| null_count | 0.0        | 0.0         |
| mean       | 599.5      | 249.252516  |
| std        | 346.554469 | 240.584178  |
| min        | 0.0        | 64.971487   |
| max        | 1199.0     | 5856.081144 |
| median     | 599.5      | 192.460503  |
+------------+------------+-------------+
...
```

If we want to take a random entry sample, we can do so:

Code

```
df.sample(5)
```

Output

```
+-----+------------+-----------------+
|     | realSum    | room_type       |
| --- | ---        | ---             |
| i64 | f64        | str             |
+====================================+
| 636 | 139.29139  | Private room    |
| 13  | 577.498364 | Entire home/apt |
| 240 | 291.203141 | Entire home/apt |
| 15  | 127.605871 | Private room    |
| 707 | 755.819389 | Private room    |
+-----+------------+-----------------+
```

2. Indexing, selecting and filtering

`Polars` offers two main ways of indexing or filtering a DataFrame:

- Using square brackets `[]`.
- Using the `select` and `filter` methods.
  - The `select` method is used to select columns.
  - The `filter` method is used to select rows.

The square brackets `[]` method works similarly to `Pandas` but has limited usage in `Polars`; it only works in eager mode, and operations on multiple columns are not parallelized.

This method is recommended in the following cases:

- To extract a scalar value from a `DataFrame`.
- To convert a `DataFrame` column to a `Series`.
- For exploratory data analysis and to inspect some rows and/or columns.

2.1 Select

We can select the `realSum` column:

Code

```
df.select(pl.col("realSum"))
```

Output

```
┌────────────┐
│ realSum    │
│ ---        │
│ f64        │
╞════════════╡
│ 185.799757 │
│ 387.49182  │
│ 194.914462 │
│ 171.777134 │
│ …          │
│ 134.617182 │
│ 134.617182 │
│ 160.091614 │
│ 359.680284 │
└────────────┘
```

We can see that the `pl.col()` method was used; this method accepts one main parameter, `name`, where we can directly specify the column name or include a regular expression. Regular expressions should start with `^` and end with `$`.

We can use a regular expression to select all the columns containing `room`:

Code

```
df.select(pl.col("^room.*$"))
```

Output

```
┌─────────────────┬─────────────┬──────────────┐
│ room_type       ┆ room_shared ┆ room_private │
│ ---             ┆ ---         ┆ ---          │
│ str             ┆ bool        ┆ bool         │
╞═════════════════╪═════════════╪══════════════╡
│ Private room    ┆ false       ┆ true         │
│ Entire home/apt ┆ false       ┆ false        │
│ Private room    ┆ false       ┆ true         │
│ Private room    ┆ false       ┆ true         │
│ …               ┆ …           ┆ …            │
│ Private room    ┆ false       ┆ true         │
│ Private room    ┆ false       ┆ true         │
│ Entire home/apt ┆ false       ┆ false        │
│ Entire home/apt ┆ false       ┆ false        │
└─────────────────┴─────────────┴──────────────┘
```

Three columns were returned, which coincides with the expected columns from our `df.columns` output.

To select every column or exclude a column, we can use the following:

Code

```
# Selecting all
df.select(pl.col("*"))

# Selecting all except
df.select(pl.exclude("realSum"))
```

To select based on the `dtype` of the columns:

Code

```
df.select(pl.col(pl.Int64))
```

Output

```
shape: (1200, 5)
+------+-----------------+-------+-----+----------+
|      | person_capacity | multi | biz | bedrooms |
| ---  | ---             | ---   | --- | ---      |
| i64  | i64             | i64   | i64 | i64      |
+=================================================+
| 0    | 2               | 0     | 0   | 1        |
| 1    | 6               | 0     | 1   | 2        |
| 2    | 5               | 0     | 1   | 1        |
| 3    | 2               | 0     | 0   | 1        |
| ...  | ...             | ...   | ... | ...      |
| 1196 | 4               | 1     | 0   | 1        |
| 1197 | 4               | 1     | 0   | 1        |
| 1198 | 3               | 0     | 0   | 1        |
| 1199 | 4               | 1     | 0   | 1        |
+------+-----------------+-------+-----+----------+
```

2.2 Filter

We can also filter by `bedrooms` using a boolean comparison, select the `metro_dist` column, sort it ascendingly and get the first five entries:

Code

```
(df.filter(pl.col("bedrooms") >= 2).
 select(pl.col("metro_dist")).
 sort("metro_dist").
 head(5)
)
```

Output

```
┌────────────┐
│ metro_dist │
│ ---        │
│ f64        │
╞════════════╡
│ 0.088494   │
│ 0.110308   │
│ 0.12578    │
│ 0.131641   │
│ 0.135447   │
└────────────┘
```

Similar to `Pandas`, the execution order of a statement is from top to bottom, meaning it will filter the `bedrooms` column first and get the head of the resulting object last.

2.3 Filtering with multiple conditions

We want to look for a clean place hosting two people with a single bedroom. We want to sort descending by `cleanliness_rating` and be able to identify the site by its GPS coordinates.

Let us filter rooms with `person_capacity` = 2, `bedrooms` = 1, and sorting descending by `cleanliness_rating`:

Code

```
berlin_places = (df.filter((pl.col("person_capacity") == 2) &
                           (pl.col("bedrooms")) == 1).
                 groupby(['lat', 'lng'], maintain_order=True).
                 agg(pl.col("cleanliness_rating").mean()).
                 sort('cleanliness_rating', descending = True)
                 )
```

Output

```
┌──────────┬──────────┬────────────────────┐
│ lat      ┆ lng      ┆ cleanliness_rating │
│ ---      ┆ ---      ┆ ---                │
│ f64      ┆ f64      ┆ f64                │
╞══════════╪══════════╪════════════════════╡
│ 52.4915  ┆ 13.42344 ┆ 10.0               │
│ 52.47842 ┆ 13.5244  ┆ 10.0               │
│ 52.51229 ┆ 13.45862 ┆ 10.0               │
│ 52.49265 ┆ 13.43842 ┆ 10.0               │
│ ...      ┆ ...      ┆ ...                │
│ 52.49937 ┆ 13.35408 ┆ 6.0                │
│ 52.573   ┆ 13.42254 ┆ 6.0                │
│ 52.49168 ┆ 13.30429 ┆ 5.0                │
│ 52.51526 ┆ 13.46914 ┆ 4.0                │
└──────────┴──────────┴────────────────────┘
```

As we move further, we can see a pattern in `Polars` syntax; it's very similar to SQL's while simultaneously being related to `Pandas`. `Polars` almost writes as a declarative language, with each transformation step exposing clear steps. Clarity increases if we separate each statement in a newline continuation.

Since we don't have the actual addresses for the places we would like to study, we will use the geolocation libraries we installed earlier to visualize these coordinates in a `folium` map:

Code

```
# Multiple filtering
berlin_places = (df.filter((pl.col("person_capacity") == 2) &
                           (pl.col("bedrooms")) == 1).
                 sort('cleanliness_rating', descending = True).
                 head(10)
                 )

# Creating a folium map, initializing view with first item
berlin_map = folium.Map(
    location=[berlin_places[0]['lat'][0], berlin_places[0]['lng'][0]],
    tiles='cartodbpositron',
    zoom_start=12,
)

# Adding remaining coordinates
FastMarkerCluster(data=list(zip(berlin_places['lat'], berlin_places['lng']))).add_to(berlin_map)

# Export map to HTML file and visualize using any browser
berlin_map.save(os.path.join(wDir, 'berlin_places.html'))
```

Output

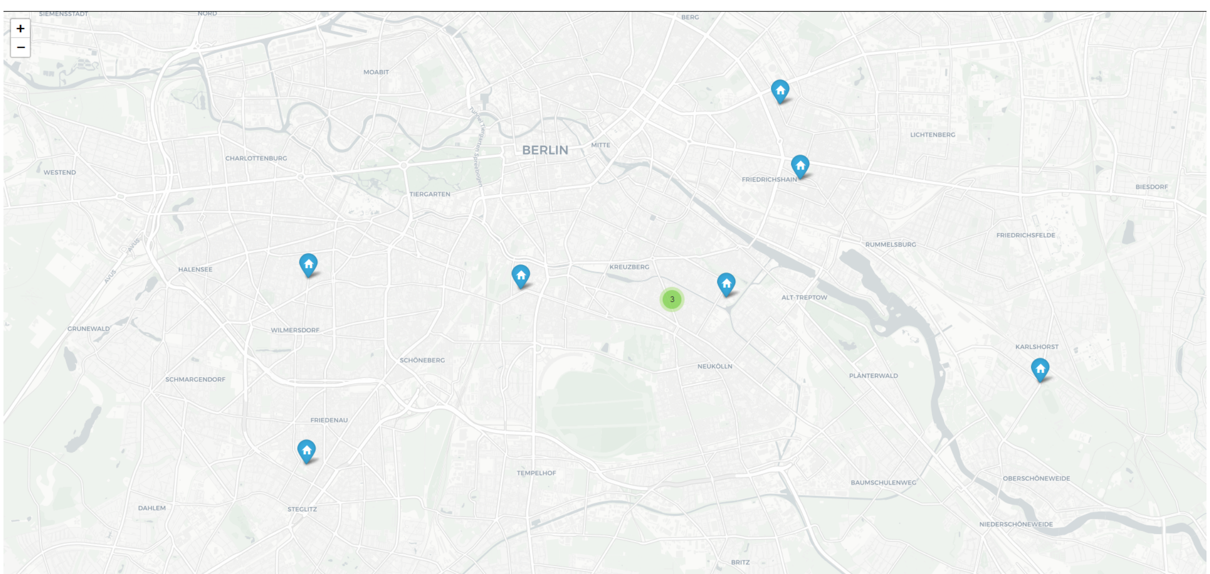

Figure 3: Berlin Place Coordinates In An HTML Folio Map

It seems like we should be looking for places near the Neukölln and Friedrichshain-Kreuzberg boroughs.

2.4. Filtering with advanced operators

We can make use of more advanced filtering operators to narrow our search:

Code

```
# Filter between range
(df.filter(pl.col("bedrooms").is_between(2, 4)).
 select(pl.col(['bedrooms', 'room_type'])).
 head(5)
)

# Filter null values
(df.filter(pl.col("bedrooms").is_null()).
 select(pl.col(['bedrooms', 'room_type'])).
 head(5)
)
```

Output

```
shape: (5, 2)
+----------+-----------------+
| bedrooms | room_type       |
| ---      | ---             |
| i64      | str             |
+===========================+
| 2        | Entire home/apt |
| 2        | Entire home/apt |
| 3        | Entire home/apt |
| 2        | Private room    |
| 2        | Entire home/apt |
+----------+-----------------+

shape: (0, 2)
+----------+-----------+
| bedrooms | room_type |
| ---      | ---       |
| i64      | str       |
+=====================+
+----------+-----------+
```

3. Aggregations

Similar to `Pandas`, we can use the `groupby` method to group different columns and perform aggregations using various functions.

Let us group by `room_shared` and calculate the average `cleanliness_rating` for each case:

Code

```
(df.groupby(['room_shared'], maintain_order=True).
    agg(pl.col("cleanliness_rating").mean())
    )
```

Output

```
┌─────────────┬────────────────────┐
│ room_shared ┆ cleanliness_rating │
│ ---         ┆ ---                │
│ bool        ┆ f64                │
╞═════════════╪════════════════════╡
│ false       ┆ 9.462995           │
│ true        ┆ 8.973684           │
└─────────────┴────────────────────┘
```

It appears that shared rooms are slightly behind in terms of cleanliness.

It's important to note that we're not using Python's aggregation methods; the methods are `Polars` implementations, meaning they're optimized for working with `Polars` DataFrame objects.

4. Joins

`Polars` supports several `join` strategies accessible by specifying the `strategy` argument.

The main strategies are:

- `inner` : Produces a `DataFrame` that contains only the rows where the join key exists in both `DataFrames` .
- `left` : Produces a `DataFrame` that contains all the rows from the left `DataFrame` and only the rows from the right `DataFrame` where the join key exists in the left `DataFrame` .
- `outer` : Produces a `DataFrame` that contains all the rows from both `DataFrames` .
- `cross` : Performs the cartesian product of the two `DataFrames` .

We can perform a `join` operation:

Code

```python
# Declare dataframes
df_writers = pl.DataFrame(
        {
         'name' : ['Jack', 'Charles', 'Clarice'],
        'surname' : ['Kerouac', 'Bukowski', 'Lispector'],
        'birth' : [datetime(1922, 3, 12),
                   datetime(1920, 8, 16),
                   datetime(1920, 12, 10)]
        }
)

df_books = pl.DataFrame(
        {
         'name' : ['Jack', 'Charles', 'Clarice'],
        'surname' : ['Kerouac', 'Bukowski', 'Lispector'],
        'book' : ['On The Road',
                  'Ham On Rye',
                  'The Passion According to G.H.']
        }
)

# Join
df_writers = df_writers.join(df_books, on=['name', 'surname'], how="left")

# Print result
df_writers
```

Output

```
shape: (3, 4)
+---------+-----------+---------------------+------------------------------+
| name    | surname   | birth               | book                         |
| ---     | ---       | ---                 | ---                          |
| str     | str       | datetime[μs]        | str                          |
+==========================================================================+
| Jack    | Kerouac   | 1922-03-12 00:00:00 | On The Road                  |
| Charles | Bukowski  | 1920-08-16 00:00:00 | Ham On Rye                   |
| Clarice | Lispector | 1920-12-10 00:00:00 | The Passion According to G.H. |
+---------+-----------+---------------------+------------------------------+
```

5. Concatenations

While a `join` operation is most often performed over the horizontal axis, a `concat` operation is performed over the vertical axis.

This can help us stack DataFrame objects, given they're of the same dimensions and data types:

Code

```python
# Read two different DataFrames
df_berlin = pl.read_csv(os.path.join(rDir, 'berlin_weekends.csv'))
df_vienna = pl.read_csv(os.path.join(rDir, 'vienna_weekends.csv'))

# Add city column to each one
df_berlin = (df_berlin.
             with_columns(pl.lit('Berlin').
             alias('city'))
             )

df_vienna = (df_vienna.
             with_columns(pl.lit('Vienna').
             alias('city'))
             )

# Concatenate them on vertical axis
df_berlin_vienna = pl.concat([df_berlin, df_vienna])
```

Output

```
SchemaError: cannot vstack: because column datatypes (dtypes) in the two DataFrames do not match
for left.name='person_capacity' with left.dtype=i64 != right.dtype=f64 with
right.name='person_capacity'
```

It seems like we got a `SchemaError`. The reason is that despite coming from the same source and having the same shape, our data sets have different data types in one of their columns, `person_capacity`. A `SchemaError` can represent the same as a `TypeError`; the only difference is that `Polars` uses schemas to define DataFrame objects.

In order to solve this conflict, we have two options:

- Cast `person_capacity` from `df_berlin` to `float64` data type.
- Cast `person_capacity` from `df_vienna` to `int64` data type.

Since there are no half-persons, we can proceed with the second option:

Code

```python
# Due to SchemaError, we need to cast data type from column person_capacity
df_vienna = df_vienna.with_columns(pl.col("person_capacity").cast(pl.Int64))

# Try concatenation again
df_berlin_vienna = pl.concat([df_berlin, df_vienna])
```

We can verify that our operation was performed successfully by getting the unique values for `city` from our resulting DataFrame:

Code

```
(df_berlin_vienna.groupby(['city'], maintain_order=True).
 agg(pl.col('lat').n_unique().
     alias('unique_latitudes')
     )
 )
```

Output

```
shape: (2, 2)
┌────────┬──────────────────┐
│ city   ┆ unique_latitudes │
│ ---    ┆ ---              │
│ str    ┆ u32              │
╞════════╪══════════════════╡
│ Berlin ┆ 1076             │
│ Vienna ┆ 1462             │
└────────┴──────────────────┘
```

6. Creating new columns

We already reviewed an example of creating new columns in `Polars` in the Writing section. The general syntax for this operation includes the following methods (the *`alias()` method is only required when we're trying to assign a new column which is the product of an aggregation operation*):

- `with_columns()`
- `alias()`

We can define a new column based on another object:

Code

```
# Define a numpy array of ones
new_col = np.random.random([len(df)])

# Assign new column to dataframe
df = df.with_columns(pl.Series(name="new_col", values=new_col))
```

Output

```
┌──────────┬──────────┬──────────┐
│ lng      ┆ lat      ┆ new_col  │
│ ---      ┆ ---      ┆ ---      │
│ f64      ┆ f64      ┆ f64      │
╞══════════╪══════════╪══════════╡
│ 13.42344 ┆ 52.4915  ┆ 0.414997 │
│ 13.503   ┆ 52.509   ┆ 0.397309 │
│ 13.468   ┆ 52.519   ┆ 0.277131 │
│ 13.47096 ┆ 52.51527 ┆ 0.429678 │
│ ...      ┆ ...      ┆ ...      │
│ 13.53187 ┆ 52.40874 ┆ 0.06139  │
│ 13.53301 ┆ 52.40712 ┆ 0.810651 │
│ 13.70702 ┆ 52.42405 ┆ 0.92665  │
│ 13.691   ┆ 52.37    ┆ 0.853674 │
└──────────┴──────────┴──────────┘
```

We can also define a new column name after some operation such as an aggregation:

Code

```
(df.groupby(['room_shared'], maintain_order=True).
 agg(pl.col('cleanliness_rating').mean().
     alias('average_cleanliness')
     )
 )
```

Output

```
shape: (2, 2)
┌─────────────┬─────────────────────┐
│ room_shared ┆ average_cleanliness │
│ ---         ┆ ---                 │
│ bool        ┆ f64                 │
╞═════════════╪═════════════════════╡
│ false       ┆ 9.462995            │
│ true        ┆ 8.973684            │
└─────────────┴─────────────────────┘
```

It's important to note that `alias()` is a method belonging to the `pl.col()` method and not to the DataFrame object. This makes sense since `alias()` aims at renaming or giving a name to a given column.

§

Multi-threaded execution

`Polars` uses an approach called *split-apply-combine* to process data. Multi-threaded execution happens on both the *split* and *apply* phases.

We can describe this process applied to a `groupby()` operation as follows:

- Data is loaded and contained in a `Polars` DataFrame object.
- Upon calling a `groupby()` operation, this DataFrame is split into n partitions.
- The aggregating operation is applied to each partition separately and in parallel.
- All partitions are then combined to build the final return object.

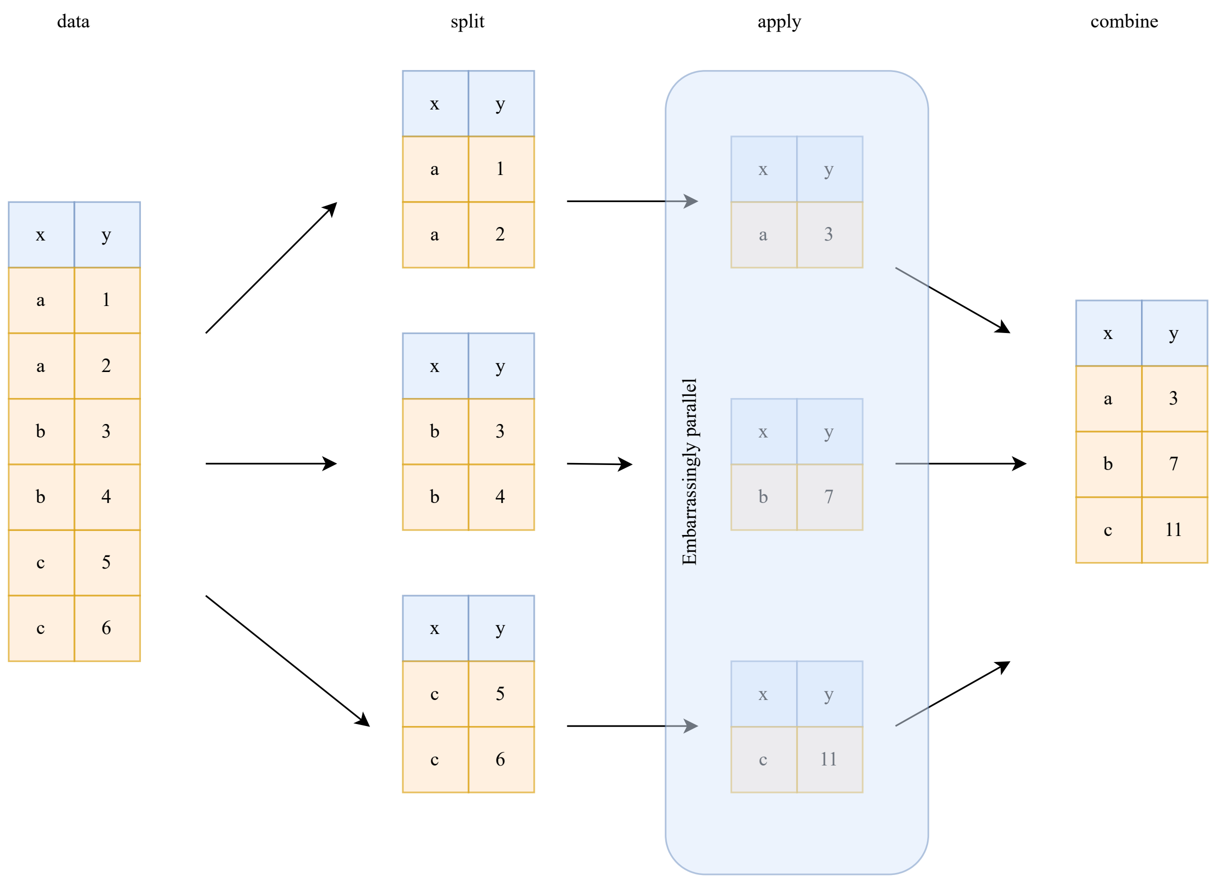

Figure 4: `Polars` Multithreaded Approach

For the hashing operations performed during the *split* phase, `Polars` uses a multithreaded lock-free approach that is illustrated in the following schema:

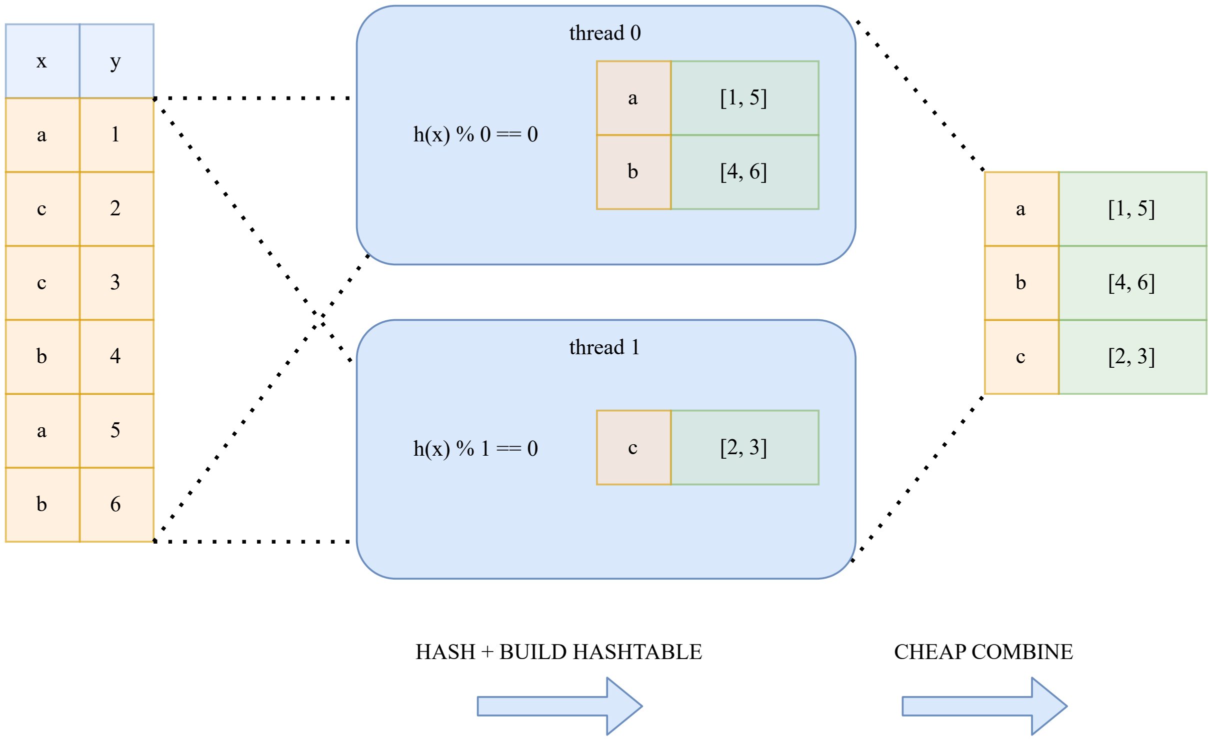

Figure 5: `Polars` Hashing Operations

A multi-threaded approach makes execution faster since multiple tasks are being processed simultaneously. That is not to say that we can use whichever method or function we wish and still be parallelized; if we were to use a `lambda` or a custom `Python` function to apply during a parallelized phase, `Polars` speed would be capped running `Python` code preventing any multiple threads from executing the function.

This is important to remember; if we're looking to maximize efficiency, the idea is to use native `Polars` functions and methods whenever possible.

§

Schemas and data types

As mentioned earlier, `Polars` works with schemas; the term *schema* is originally defined in a relational database context, representing how the data may relate to other tables or data models. In APIs such as `PySpark` or `Polars`, a schema is the data set type definition.

When working with Python, we often do not have to pay attention to the data types since Python is a dynamically typed language, meaning data type definitions are unnecessary. This applies, of course, when we have the data types we need; otherwise, we cast the data to their required data types.

Dynamic typing does not mean data types are ignored or not required, but they are inferred upon execution. This is a resource-intensive task, especially with large data sets. Also, not having a predefined schema can cause data type errors such as the one we encountered earlier; when we loaded our data sets, `Polars` inferred the data types based on the data set values.

`Polars` supports a wide variety of data types:

Class	Type	Description
Numeric	`Float32`	32-bit floating point type.
Numeric	`Float64`	64-bit floating point type.
Numeric	`Int16`	16-bit signed integer type.
Numeric	`Int32`	32-bit signed integer type.
Numeric	`Int64`	64-bit signed integer type.
Numeric	`Int8`	8-bit signed integer type.
Numeric	`UInt16`	16-bit unsigned integer type.
Numeric	`UInt32`	32-bit unsigned integer type.
Numeric	`UInt64`	64-bit unsigned integer type.
Numeric	`UInt8`	8-bit unsigned integer type.
Date / Time	`Date`	Calendar date type.
Date / Time	`Datetime`	Calendar date and time type.
Date / Time	`Duration`	Time duration/delta type.
Date / Time	`Time`	Time of day type.
Nested	`List(*args, **kwargs)`	List.
Nested	`Struct(*args, **kwargs)`	Struct.
Other	`Boolean`	Boolean type.
Other	`Binary`	Binary type.
Other	`Categorical`	A categorical encoding of a set of strings.
Other	`Null`	Type representing Null / None values.
Other	`Object`	Type for wrapping arbitrary Python objects.
Other	`Utf8`	UTF-8 encoded string type.
Other	`Unknown`	Type representing Datatype values that could not be determined statically.

Table 3: `Polars` Data Types

To avoid these errors and make processing more efficient, we can use a predefined schema:

Code

```
# Define schema
schema = {'' : pl.Int64,
          'realSum' : pl.Float64,
          'room_type' : pl.Utf8,
          'room_shared' : pl.Boolean,
          'room_private' : pl.Boolean,
          'person_capacity' : pl.Int64,
          'host_is_superhost' : pl.Boolean,
          'multi' : pl.Int64,
          'biz' : pl.Int64,
          'cleanliness_rating' : pl.Float64,
          'guest_satisfaction_overall' : pl.Float64,
          'bedrooms' : pl.Int64,
          'dist' : pl.Float64,
          'metro_dist' : pl.Float64,
          'attr_index' : pl.Float64,
          'attr_index_norm' : pl.Float64,
          'rest_index' : pl.Float64,
          'rest_index_norm' : pl.Float64,
          'lng' : pl.Float64,
          'lat' : pl.Float64,
          'strict' : pl.Boolean,
 }

# Read dataframe
df = pl.read_csv(os.path.join(rDir, 'vienna_weekends.csv'), dtypes = schema)
```

Output

```
ComputeError: Could not parse `4.0` as dtype Int64 at column 'person_capacity' (column number 6).
The current offset in the file is 270 bytes.

You might want to try:
- increasing `infer_schema_length` (e.g. `infer_schema_length=10000`),
- specifying the correct dtype with the `dtypes` argument
- setting `ignore_errors` to `True`,
- adding `4.0` to the `null_values` list.
```

The problem with predefining a schema upon data set reading is that if a given value does not match the predefined data type, it will return an error; the `dtype=schema` argument will not try to cast the elements. It will only try to set them.

What we can do to solve this issue is load our data set without inferring its schema and then cast all columns using our `schema` dictionary:

Code

```
# Load Vienna data set without infering schema
df = pl.read_csv(os.path.join(rDir, 'vienna_weekends.csv'), infer_schema_length=0)

# Iteratively cast data types
for i, x in schema.items():
    df = df.with_columns(pl.col(i).cast(x), strict = False)
```

If we look closely, we included a new parameter, `infer_schema_length=0`, which tells `Polars` that we don't want an inferred schema. This will set the data type to `Utf8` for all columns.

Output

```
ArrowErrorException: NotYetImplemented("Casting from LargeUtf8 to Boolean not supported")
```

The problem with this approach is that `pl.Boolean` type casting accepts a capitalized string. Since we have some columns with their boolean value in lowercase, this method also returns an error.

We can fix this by adding an exception handling specifically for these types of errors:

Code

```
for i, x in schema.items():
    try:
        df = df.with_columns(pl.col(i).cast(x), strict = False)

        # If we encounter a boolean lowercased column, we need to treat it specially
    except:
        df = df.with_columns(pl.col(i) == 'true')
        df = df.with_columns(pl.col(i).cast(x), strict = False)
```

Whenever an exception is raised, we regenerate the entire column by making a logical comparison against `"true"`. This fills our target column with actual boolean values, so we don't have to cast it afterwards.

This, of course, is somewhat problematic in cases when we don't fully know the nature of our data since any exception will be caught and treated as if it were a `pl.Boolean` type casting error.

We can do further manipulations and perfect our exception handling, but that is out of the scope of this segment.

When designing a data-loading pipeline, we must account for all these details; otherwise, our program will underperform and even break.

§

Conclusions

In this segment, we've gone from zero to `Polars`; it's a lot to digest, but the important thing is that we covered the most relevant functionalities and can extend from here by consulting external resources.

For those already familiar with `Pandas`, this remarkable cheatsheet covers `Polars` translations of the most relevant `Pandas` operations.

One disadvantage of `Polars` is the lack of community discussion; `Pandas` is everywhere, all the time, and there is a vast amount of resources out there. Hopefully, more people will adopt `Polars` in the future.

Finally, it's important to keep in mind that, as we reviewed, `Polars` accepts `Pandas`-like syntax, but that does not mean we should use it if we want to maintain the high performance `Polars` was designed to output; according to the Polars User Guide, *"if your `Polars` code looks like it could be `Pandas` code, it might run, but it likely runs slower than it should."*

§

References

- `Polars` Official Page, Home
- `Polars` User Guide
- Cheatsheet for Pandas to `Polars`
- `Polars` Official Page, GroupBy
- `Polars` Official Page, Data Types

§

Copyright