Polars User Guide

None

None

None

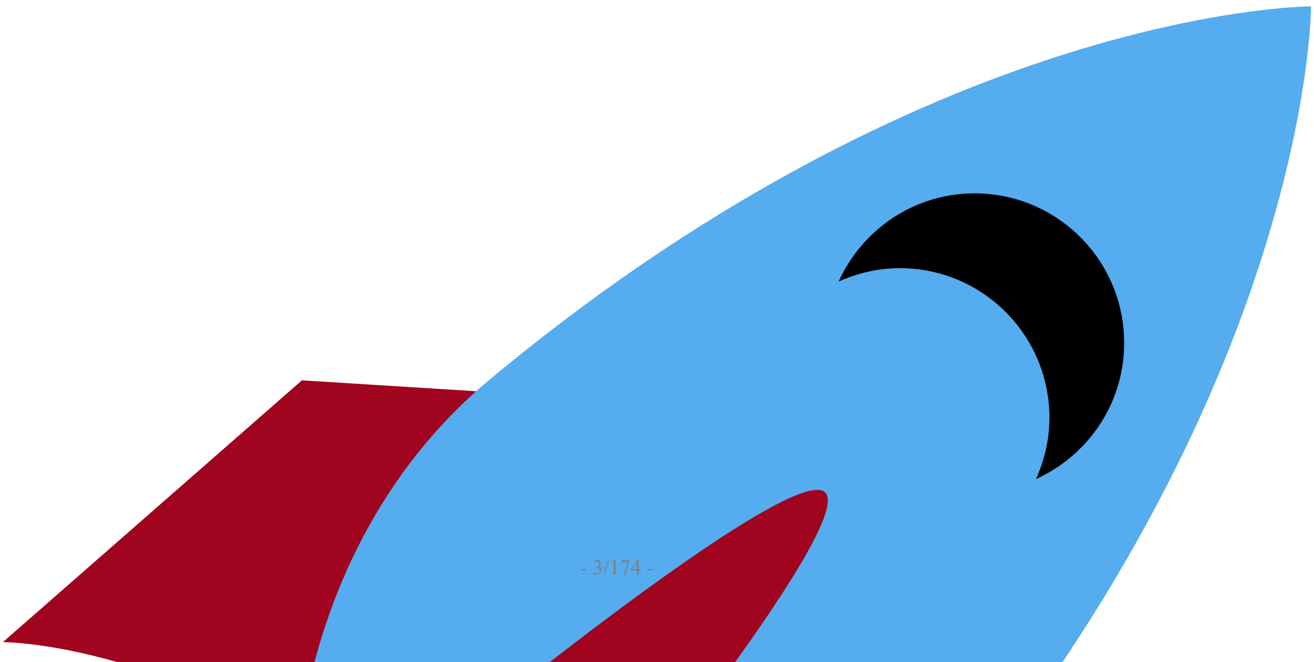
Table of contents

1. Polars 5

2. Blazingly Fast DataFrame Library

   2.1 About this guide 5

2.2 Performance



2.3 Example 6

2.4 Sponsors 7

2.5 Community 7

2.6 Contribute 7

2.7 License 7

3. Getting Started 8

3.1 Introduction 8

3.2 Installation 9

3.3 Series & DataFrames 10

3.4 Reading & Writing 16

3.5 Expressions 19

3.6 Combining DataFrames 27

4. User Guide 29

4.1 Introduction 29

4.2 Installation 30

4.3 Concepts 35

4.4 Expressions 50

4.5 Transformations 107

4.6 Lazy API 132

4.7 IO 142

4.8 SQL 154

4.9 Migrating 163

4.10 Misc 169

1. Polars

2. Blazingly Fast DataFrame Library

docs passing | Build and test failing | crates.io v0.31.1 | pypi v0.18.7 | npm v0.8.0

DOI 10.5281/zenodo.7697217

Polars is a highly performant DataFrame library for manipulating structured data. The core is written in Rust, but the library is available in Python, Rust & NodeJS. Its key features are:

- **Fast**: Polars is written from the ground up, designed close to the machine and without external dependencies.
- **I/O**: First class support for all common data storage layers: local, cloud storage & databases.
- **Easy to use**: Write your queries the way they were intended. Polars, internally, will determine the most efficient way to execute using its query optimizer.
- **Out of Core**: Polars supports out of core data transformation with its streaming API. Allowing you to process your results without requiring all your data to be in memory at the same time
- **Parallel**: Polars fully utilises the power of your machine by dividing the workload among the available CPU cores without any additional configuration.
- **Vectorized Query Engine**: Polars uses Apache Arrow, a columnar data format, to process your queries in a vectorized manner. It uses SIMD to optimize CPU usage.

2.1 About this guide

The `Polars` user guide is intended to live alongside the API documentation. Its purpose is to explain (new) users how to use `Polars` and to provide meaningful examples. The guide is split into two parts:

- Getting Started: A 10 minute helicopter view of the library and its primary function.
- User Guide: A detailed explanation of how the library is setup and how to use it most effectively.

If you are looking for details on a specific level / object, it is probably best to go the API documentation: Python | NodeJS | Rust.

2.2 Performance 🚀🚀

`Polars` is very fast, and in fact is one of the best performing solutions available. See the results in h2oai's db-benchmark, revived by the DuckDB project.

`Polars` TPCH Benchmark results are now available on the official website.

2.3 Example

Python Rust NodeJS

API `scan_csv` · API `filter` · API `groupby` · API `collect`

```python
import polars as pl

q = (
    pl.scan_csv("docs/src/data/iris.csv")
    .filter(pl.col("sepal_length") > 5)
    .groupby("species")
    .agg(pl.all().sum())
)

df = q.collect()
```

API `LazyCsvReader` · API `filter` · API `groupby` · API `collect` · Available on feature streaming · Available on feature csv

```rust
use polars::prelude::*;

let q = LazyCsvReader::new("docs/src/data/iris.csv")
    .has_header(true)
    .finish()?
    .filter(col("sepal_length").gt(lit(5)))
    .groupby(vec![col("species")])
    .agg([col("*").sum()]);

let df = q.collect();
```

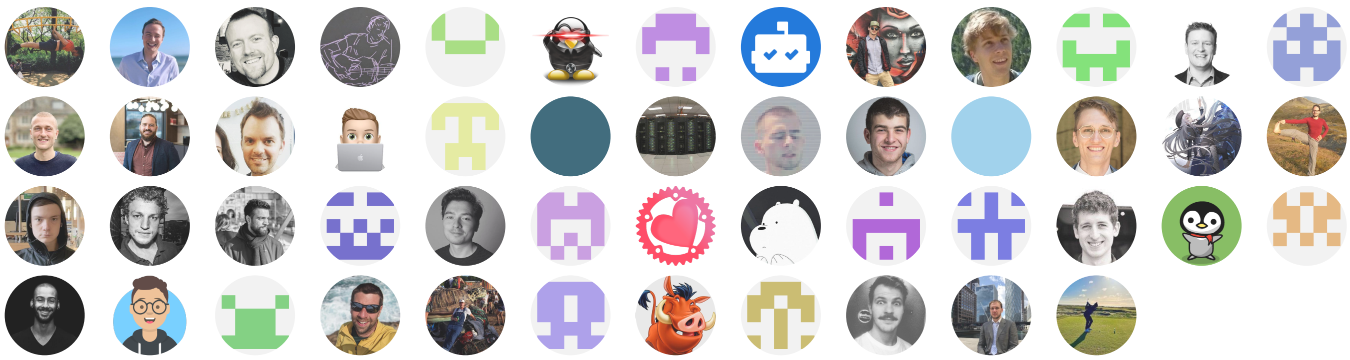
API `scanCSV` · API `filter` · API `groupBy` · API `collect`

```javascript
const pl = require("nodejs-polars");

q = pl
  .scanCSV("docs/src/data/iris.csv")
  .filter(pl.col("sepal_length").gt(5))
  .groupBy("species")
  .agg(pl.all().sum());

df = q.collect();
```

2.4 Sponsors

2.5 Community

`Polars` has a very active community with frequent releases (approximately weekly). Below are some of the top contributors to the project:



2.6 Contribute

Thanks for taking the time to contribute! We appreciate all contributions, from reporting bugs to implementing new features. If you're unclear on how to proceed read our contribution guide or contact us on discord.

2.7 License

This project is licensed under the terms of the MIT license.

3. Getting Started

3.1 Introduction

This getting started guide is written for new users of Polars. The goal is to provide a quick overview of the most common functionality. For a more detailed explanation, please go to the User Guide

Rust Users Only

Due to historical reasons the eager API in Rust is outdated. In the future we would like to redesign it as a small wrapper around the lazy API (as is the design in Python / NodeJS). In the examples we will use the lazy API instead with `.lazy()` and `.collect()`. For now you can ignore these two functions. If you want to know more about the lazy and eager API go here.

To enable the Lazy API ensure you have the feature flag `lazy` configured when installing Polars

```
# Cargo.toml
[dependencies]
polars = { version = "x", features = ["lazy", ...]}
```

Because of the ownership ruling in Rust we can not reuse the same `DataFrame` multiple times in the examples. For simplicity reasons we call `clone()` to overcome this issue. Note that this does not duplicate the data but just increments a pointer (`Arc`).

3.2 Installation

Polars is a library and installation is as simple as invoking the package manager of the corresponding programming language.

Python Rust NodeJS

```
pip install polars
```

```
cargo add polars
```

```
yarn add nodejs-polars
```

3.2.1 Importing

To use the library import it into your project

Python Rust NodeJS

```python
import polars as pl
```

```rust
use polars::prelude::*;
```

```javascript
// esm
import pl from 'nodejs-polars';

// require
const pl = require('nodejs-polars');
```

3.3 Series & DataFrames

The core base data structures provided by Polars are `Series` and `DataFrames`.

3.3.1 Series

Series are a 1-dimensional data structure. Within a series all elements have the same data type (e.g. int, string). The snippet below shows how to create a simple named `Series` object. In a later section of this getting started guide we will learn how to read data from external sources (e.g. files, database), for now lets keep it simple.

Python Rust NodeJS

API `Series`

```
import polars as pl

s = pl.Series("a", [1, 2, 3, 4, 5])
print(s)
```

API `Series`

```
use chrono::prelude::*;

let s = Series::new("a", [1, 2, 3, 4, 5]);
println!("{}",s);
```

API `Series`

```
const pl = require("nodejs-polars");

var s = pl.Series("a", [1, 2, 3, 4, 5]);
console.log(s);
```

```
shape: (5,)
Series: 'a' [i64]
[
	1
	2
	3
	4
	5
]
```

Methods

Although it is more common to work directly on a `DataFrame` object, `Series` implement a number of base methods which make it easy to perform transformations. Below are some examples of common operations you might want to perform. Note that these are for illustration purposes and only show a small subset of what is available.

Aggregations

`Series` out of the box supports all basic aggregations (e.g. min, max, mean, mode, ...).

Python Rust NodeJS

API `min` · API `max`

```
s = pl.Series("a", [1, 2, 3, 4, 5])
print(s.min())
print(s.max())
```

API `min` · API `max`

```
let s = Series::new("a", [1, 2, 3, 4, 5]);
// The use of generics is necessary for the type system
println!("{}",s.min::<u64>().unwrap());
println!("{}",s.max::<u64>().unwrap());
```

API `min` · API `max`

```
var s = pl.Series("a", [1, 2, 3, 4, 5]);
console.log(s.min());
console.log(s.max());
```

```
1
5
```

String

There are a number of methods related to string operations in the `StringNamespace`. These only work on `Series` with the Datatype `Utf8`.

Python Rust NodeJS

API `replace`

```
s = pl.Series("a", ["polar", "bear", "arctic", "polar fox", "polar bear"])
s2 = s.str.replace("polar", "pola")
print(s2)
```

```
// This operation is not directly available on the Series object yet, only on the DataFrame
```

API `replace`

```
var s = pl.Series("a", ["polar", "bear", "arctic", "polar fox", "polar bear"]);
var s2 = s.str.replace("polar", "pola");
console.log(s2);
```

```
shape: (5,)
Series: 'a' [str]
[
    "pola"
    "bear"
    "arctic"
    "pola fox"
    "pola bear"
]
```

Datetime

Similar to strings, there is a seperate namespace for datetime related operations in the `DateLikeNameSpace`. These only work on `Series` with DataTypes related to dates.

Python Rust NodeJS

API `day`

```
from datetime import datetime

start = datetime(2001, 1, 1)
stop = datetime(2001, 1, 9)
s = pl.date_range(start, stop, interval="2d", eager=True)
s.dt.day()
print(s)
```

```
// This operation is not directly available on the Series object yet, only on the DataFrame
```

API `day`

```
var s = pl.Series("a", [
  new Date(2001, 1, 1),
  new Date(2001, 1, 3),
  new Date(2001, 1, 5),
  new Date(2001, 1, 7),
  new Date(2001, 1, 9),
]);
var s2 = s.date.day();
console.log(s2);
```

```
shape: (5,)
Series: 'date' [datetime[μs]]
[
    2001-01-01 00:00:00
    2001-01-03 00:00:00
    2001-01-05 00:00:00
    2001-01-07 00:00:00
    2001-01-09 00:00:00
]
```

3.3.2 DataFrame

A `DataFrame` is a 2-dimensional data structure that is backed by a `Series`, and it could be seen as an abstraction of on collection (e.g. list) of `Series`. Operations that can be executed on `DataFrame` are very similar to what is done in a `SQL` like query. You can `GROUP BY`, `JOIN`, `PIVOT`, but also define custom functions. In the next pages we will cover how to perform these transformations.

Python Rust NodeJS

API `DataFrame`

```
from datetime import datetime

df = pl.DataFrame(
    {
        "integer": [1, 2, 3, 4, 5],
        "date": [
            datetime(2022, 1, 1),
            datetime(2022, 1, 2),
            datetime(2022, 1, 3),
            datetime(2022, 1, 4),
            datetime(2022, 1, 5),
        ],
        "float": [4.0, 5.0, 6.0, 7.0, 8.0],
    }
)

print(df)
```

API `DataFrame`

```
let df: DataFrame = df!("integer" => &[1, 2, 3, 4, 5],
                        "date" => &[
                            NaiveDate::from_ymd_opt(2022, 1, 1).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                            NaiveDate::from_ymd_opt(2022, 1, 2).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                            NaiveDate::from_ymd_opt(2022, 1, 3).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                            NaiveDate::from_ymd_opt(2022, 1, 4).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                            NaiveDate::from_ymd_opt(2022, 1, 5).unwrap().and_hms_opt(0, 0, 0).unwrap()
                        ],
                        "float" => &[4.0, 5.0, 6.0, 7.0, 8.0]
                        ).expect("should not fail");
println!("{}",df);
```

API `DataFrame`

```
let df = pl.DataFrame({
  integer: [1, 2, 3, 4, 5],
  date: [
    new Date(2022, 1, 1, 0, 0),
    new Date(2022, 1, 2, 0, 0),
    new Date(2022, 1, 3, 0, 0),
    new Date(2022, 1, 4, 0, 0),
    new Date(2022, 1, 5, 0, 0),
  ],
  float: [4.0, 5.0, 6.0, 7.0, 8.0],
});
console.log(df);
```

```
shape: (5, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 1       ┆ 2022-01-01 00:00:00 ┆ 4.0   │
│ 2       ┆ 2022-01-02 00:00:00 ┆ 5.0   │
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
│ 4       ┆ 2022-01-04 00:00:00 ┆ 7.0   │
│ 5       ┆ 2022-01-05 00:00:00 ┆ 8.0   │
└─────────┴─────────────────────┴───────┘
```

Viewing data

This part focuses on viewing data in a `DataFrame`. We will use the `DataFrame` from the previous example as a starting point.

HEAD

The `head` function shows by default the first 5 rows of a `DataFrame`. You can specify the number of rows you want to see (e.g. `df.head(10)`).

Python Rust NodeJS

API `head`

```
print(df.head(3))
```

API `head`

```
println!("{}",df.head(Some(3)));
```

API `head`

```
console.log(df.head(3));
```

```
shape: (3, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 1       ┆ 2022-01-01 00:00:00 ┆ 4.0   │
│ 2       ┆ 2022-01-02 00:00:00 ┆ 5.0   │
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
└─────────┴─────────────────────┴───────┘
```

TAIL

The `tail` function shows the last 5 rows of a `DataFrame`. You can also specify the number of rows you want to see, similar to `head`.

Python Rust NodeJS

API `tail`

```
print(df.tail(3))
```

API `tail`

```
println!("{}",df.tail(Some(3)));
```

API `tail`

```
console.log(df.tail(3));
```

```
shape: (3, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
│ 4       ┆ 2022-01-04 00:00:00 ┆ 7.0   │
│ 5       ┆ 2022-01-05 00:00:00 ┆ 8.0   │
└─────────┴─────────────────────┴───────┘
```

SAMPLE

If you want to get an impression of the data of your `DataFrame`, you can also use `sample`. With `sample` you get an *n* number of random rows from the `DataFrame`.

Python Rust NodeJS

API `sample`

```
print(df.sample(2))
```

API `sample_n`

```
println!("{}",df.sample_n(2, false, true, None)?);
```

API `sample`

```
console.log(df.sample(2));
```

```
shape: (2, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 1       ┆ 2022-01-01 00:00:00 ┆ 4.0   │
│ 2       ┆ 2022-01-02 00:00:00 ┆ 5.0   │
└─────────┴─────────────────────┴───────┘
```

DESCRIBE

`Describe` returns summary statistics of your `DataFrame`. It will provide several quick statistics if possible.

Python Rust NodeJS

API `describe`

```
print(df.describe())
```

API `describe` · Available on feature describe

```
println!("{}",df.describe(None));
```

API `describe`

```
console.log(df.describe());
```

```
shape: (9, 4)
┌────────────┬──────────┬─────────────────────┬──────────┐
│ describe   ┆ integer  ┆ date                ┆ float    │
│ ---        ┆ ---      ┆ ---                 ┆ ---      │
│ str        ┆ f64      ┆ str                 ┆ f64      │
╞════════════╪══════════╪═════════════════════╪══════════╡
│ count      ┆ 5.0      ┆ 5                   ┆ 5.0      │
│ null_count ┆ 0.0      ┆ 0                   ┆ 0.0      │
│ mean       ┆ 3.0      ┆ null                ┆ 6.0      │
│ std        ┆ 1.581139 ┆ null                ┆ 1.581139 │
│ min        ┆ 1.0      ┆ 2022-01-01 00:00:00 ┆ 4.0      │
│ max        ┆ 5.0      ┆ 2022-01-05 00:00:00 ┆ 8.0      │
│ median     ┆ 3.0      ┆ null                ┆ 6.0      │
│ 25%        ┆ 2.0      ┆ null                ┆ 5.0      │
│ 75%        ┆ 4.0      ┆ null                ┆ 7.0      │
└────────────┴──────────┴─────────────────────┴──────────┘
```

3.4 Reading & Writing

Polars supports reading & writing to all common files (e.g. csv, json, parquet), cloud storage (S3, Azure Blob, BigQuery) and databases (e.g. postgres, mysql). In the following examples we will show how to operate on most common file formats. For the following dataframe

Python Rust NodeJS

API DataFrame

```python
import polars as pl
from datetime import datetime

df = pl.DataFrame(
    {
        "integer": [1, 2, 3],
        "date": [
            datetime(2022, 1, 1),
            datetime(2022, 1, 2),
            datetime(2022, 1, 3),
        ],
        "float": [4.0, 5.0, 6.0],
    }
)

print(df)
```

API DataFrame

```rust
use std::fs::File;
use chrono::prelude::*;

let mut df: DataFrame = df!("integer" => &[1, 2, 3],
                        "date" => &[
                                    NaiveDate::from_ymd_opt(2022, 1, 1).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                                    NaiveDate::from_ymd_opt(2022, 1, 2).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                                    NaiveDate::from_ymd_opt(2022, 1, 3).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                        ],
                        "float" => &[4.0, 5.0, 6.0]
                        ).expect("should not fail");
println!("{}",df);
```

API DataFrame

```javascript
let df = pl.DataFrame({
  integer: [1, 2, 3],
  date: [
    new Date(2022, 1, 1, 0, 0),
    new Date(2022, 1, 2, 0, 0),
    new Date(2022, 1, 3, 0, 0),
  ],
  float: [4.0, 5.0, 6.0],
});
console.log(df);
```

```
shape: (3, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 1       ┆ 2022-01-01 00:00:00 ┆ 4.0   │
│ 2       ┆ 2022-01-02 00:00:00 ┆ 5.0   │
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
└─────────┴─────────────────────┴───────┘
```

CSV

Polars has its own fast implementation for csv reading with many flexible configuration options.

Python Rust NodeJS

API `read_csv` · API `write_csv`

```
df.write_csv("output.csv")
df_csv = pl.read_csv("output.csv")
print(df_csv)
```

API `CsvReader` · API `CsvWriter` · Available on feature csv

```
let mut file = File::create("output.csv").expect("could not create file");
CsvWriter::new(&mut file).has_header(true).with_delimiter(b',').finish(&mut df);
let df_csv = CsvReader::from_path("output.csv")?.infer_schema(None).has_header(true).finish()?;
println!("{}",df_csv);
```

API `readCSV` · API `writeCSV`

```
df.writeCSV("output.csv");
var df_csv = pl.readCSV("output.csv");
console.log(df_csv);
```

```
shape: (3, 3)
┌─────────┬────────────────────────────┬───────┐
│ integer ┆ date                       ┆ float │
│ ---     ┆ ---                        ┆ ---   │
│ i64     ┆ str                        ┆ f64   │
╞═════════╪════════════════════════════╪═══════╡
│ 1       ┆ 2022-01-01T00:00:00.000000 ┆ 4.0   │
│ 2       ┆ 2022-01-02T00:00:00.000000 ┆ 5.0   │
│ 3       ┆ 2022-01-03T00:00:00.000000 ┆ 6.0   │
└─────────┴────────────────────────────┴───────┘
```

As we can see above, Polars made the datetimes a `string`. We can tell Polars to parse dates, when reading the csv, to ensure the date becomes a datetime. The example can be found below:

Python Rust NodeJS

API `read_csv`

```
df_csv = pl.read_csv("output.csv", try_parse_dates=True)
print(df_csv)
```

API `CsvReader` · Available on feature csv

```
let mut file = File::create("output.csv").expect("could not create file");
CsvWriter::new(&mut file).has_header(true).with_delimiter(b',').finish(&mut df);
let df_csv = CsvReader::from_path("output.csv")?.infer_schema(None).has_header(true).with_parse_dates(true).finish()?;
println!("{}",df_csv);
```

API `readCSV`

```
var df_csv = pl.readCSV("output.csv", { parseDates: true });
console.log(df_csv);
```

```
shape: (3, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 1       ┆ 2022-01-01 00:00:00 ┆ 4.0   │
│ 2       ┆ 2022-01-02 00:00:00 ┆ 5.0   │
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
└─────────┴─────────────────────┴───────┘
```

JSON

Python Rust NodeJS

API `read_json` · API `write_json`

```python
df.write_json("output.json")
df_json = pl.read_json("output.json")
print(df_json)
```

API `JsonReader` · API `JsonWriter` · Available on feature json

```rust
let mut file = File::create("output.json").expect("could not create file");
JsonWriter::new(&mut file).finish(&mut df);
let mut f = File::open("output.json")?;
let df_json = JsonReader::new(f).with_json_format(JsonFormat::JsonLines).finish()?;
println!("{}",df_json);
```

API `readJSON` · API `writeJSON`

```javascript
df.writeJSON("output.json", { format: "json" });
let df_json = pl.readJSON("output.json");
console.log(df_json);
```

```
shape: (3, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 1       ┆ 2022-01-01 00:00:00 ┆ 4.0   │
│ 2       ┆ 2022-01-02 00:00:00 ┆ 5.0   │
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
└─────────┴─────────────────────┴───────┘
```

PARQUET

Python Rust NodeJS

API `read_parquet` · API `write_parquet`

```python
df.write_parquet("output.parquet")
df_parquet = pl.read_parquet("output.parquet")
print(df_parquet)
```

API `ParquetReader` · API `ParquetWriter` · Available on feature parquet

```rust
let mut file = File::create("output.parquet").expect("could not create file");
ParquetWriter::new(&mut file).finish(&mut df);
let mut f = File::open("output.parquet")?;
let df_parquet = ParquetReader::new(f).finish()?;
println!("{}",df_parquet);
```

API `readParquet` · API `writeParquet`

```javascript
df.writeParquet("output.parquet");
let df_parquet = pl.readParquet("output.parquet");
console.log(df_parquet);
```

```
shape: (3, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 1       ┆ 2022-01-01 00:00:00 ┆ 4.0   │
│ 2       ┆ 2022-01-02 00:00:00 ┆ 5.0   │
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
└─────────┴─────────────────────┴───────┘
```

To see more examples and other data formats go to the User Guide, section IO.

3.5 Expressions

`Expressions` are the core strength of `Polars`. The `expressions` offer a versatile structure that both solves easy queries and is easily extended to complex ones. Below we will cover the basic components that serve as building block (or in `Polars` terminology contexts) for all your queries:

- `select`
- `filter`
- `with_columns`
- `groupby`

To learn more about expressions and the context in which they operate, see the User Guide sections: Contexts and Expressions.

Select statement

To select a column we need to do two things. Define the `DataFrame` we want the data from. And second, select the data that we need. In the example below you see that we select `col('*')`. The asterisk stands for all columns.

Python Rust NodeJS

API `select`

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

API `select`

```
let out = df.clone().lazy().select([col("*")]).collect()?;
println!("{}",out);
```

API `select`

```
df.select(pl.col("*"));
```

```
shape: (8, 4)
┌─────┬──────────┬─────────────────────┬───────┐
│ a   ┆ b        ┆ c                   ┆ d     │
│ --- ┆ ---      ┆ ---                 ┆ ---   │
│ i64 ┆ f64      ┆ datetime[μs]        ┆ f64   │
╞═════╪══════════╪═════════════════════╪═══════╡
│ 0   ┆ 0.577013 ┆ 2022-12-01 00:00:00 ┆ 1.0   │
│ 1   ┆ 0.114686 ┆ 2022-12-02 00:00:00 ┆ 2.0   │
│ 2   ┆ 0.612896 ┆ 2022-12-03 00:00:00 ┆ NaN   │
│ 3   ┆ 0.342322 ┆ 2022-12-04 00:00:00 ┆ NaN   │
│ 4   ┆ 0.185987 ┆ 2022-12-05 00:00:00 ┆ 0.0   │
│ 5   ┆ 0.376874 ┆ 2022-12-06 00:00:00 ┆ -5.0  │
│ 6   ┆ 0.286651 ┆ 2022-12-07 00:00:00 ┆ -42.0 │
│ 7   ┆ 0.646312 ┆ 2022-12-08 00:00:00 ┆ null  │
└─────┴──────────┴─────────────────────┴───────┘
```

You can also specify the specific columns that you want to return. There are two ways to do this. The first option is to create a `list` of column names, as seen below.

Python Rust NodeJS

API `select`

```
df.select(pl.col(["a", "b"]))
```

API `select`

```
let out = df.clone().lazy().select([col("a"), col("b")]).collect()?;
println!("{}",out);
```

API `select`

```
df.select(pl.col(["a", "b"]));
```

```
shape: (8, 2)
┌─────┬──────────┐
│ a   ┆ b        │
│ --- ┆ ---      │
│ i64 ┆ f64      │
╞═════╪══════════╡
│ 0   ┆ 0.577013 │
│ 1   ┆ 0.114686 │
│ 2   ┆ 0.612896 │
│ 3   ┆ 0.342322 │
│ 4   ┆ 0.185987 │
│ 5   ┆ 0.376874 │
│ 6   ┆ 0.286651 │
│ 7   ┆ 0.646312 │
└─────┴──────────┘
```

The second option is to specify each column within a `list` in the `select` statement. This option is shown below.

Python Rust NodeJS

API `select`

```
df.select([pl.col("a"), pl.col("b")]).limit(3)
```

API `select`

```
let out = df.clone().lazy().select([col("a"), col("b")]).limit(3).collect()?;
println!("{}",out);
```

API `select`

```
df.select([pl.col("a"), pl.col("b")]).limit(3);
```

```
shape: (3, 2)
┌─────┬──────────┐
│ a   ┆ b        │
│ --- ┆ ---      │
│ i64 ┆ f64      │
╞═════╪══════════╡
│ 0   ┆ 0.577013 │
│ 1   ┆ 0.114686 │
│ 2   ┆ 0.612896 │
└─────┴──────────┘
```

If you want to exclude an entire column from your view, you can simply use `exclude` in your `select` statement.

Python Rust NodeJS

API `select`

```
df.select([pl.exclude("a")])
```

API `select`

```
let out = df.clone().lazy().select([col("*").exclude(["a"])]).collect()?;
println!("{}",out);
```

API `select`

```
df.select([pl.exclude("a")]);
```

```
shape: (8, 3)
┌──────────┬─────────────────────┬───────┐
│ b        ┆ c                   ┆ d     │
│ ---      ┆ ---                 ┆ ---   │
│ f64      ┆ datetime[μs]        ┆ f64   │
╞══════════╪═════════════════════╪═══════╡
│ 0.577013 ┆ 2022-12-01 00:00:00 ┆ 1.0   │
│ 0.114686 ┆ 2022-12-02 00:00:00 ┆ 2.0   │
│ 0.612896 ┆ 2022-12-03 00:00:00 ┆ NaN   │
│ 0.342322 ┆ 2022-12-04 00:00:00 ┆ NaN   │
│ 0.185987 ┆ 2022-12-05 00:00:00 ┆ 0.0   │
│ 0.376874 ┆ 2022-12-06 00:00:00 ┆ -5.0  │
│ 0.286651 ┆ 2022-12-07 00:00:00 ┆ -42.0 │
│ 0.646312 ┆ 2022-12-08 00:00:00 ┆ null  │
└──────────┴─────────────────────┴───────┘
```

Filter

The `filter` option allows us to create a subset of the `DataFrame`. We use the same `DataFrame` as earlier and we filter between two specified dates.

Python Rust NodeJS

API `filter`

```
df.filter(
    pl.col("c").is_between(datetime(2022, 12, 2), datetime(2022, 12, 8)),
)
```

API `filter`

```
// TODO
```

API `filter`

```
df.filter(pl.col("c").gt(new Date(2022, 12, 2)).lt(new Date(2022, 12, 8)));
```

```
shape: (7, 4)
┌─────┬──────────┬─────────────────────┬───────┐
│ a   ┆ b        ┆ c                   ┆ d     │
│ --- ┆ ---      ┆ ---                 ┆ ---   │
│ i64 ┆ f64      ┆ datetime[μs]        ┆ f64   │
╞═════╪══════════╪═════════════════════╪═══════╡
│ 1   ┆ 0.114686 ┆ 2022-12-02 00:00:00 ┆ 2.0   │
│ 2   ┆ 0.612896 ┆ 2022-12-03 00:00:00 ┆ NaN   │
│ 3   ┆ 0.342322 ┆ 2022-12-04 00:00:00 ┆ NaN   │
│ 4   ┆ 0.185987 ┆ 2022-12-05 00:00:00 ┆ 0.0   │
│ 5   ┆ 0.376874 ┆ 2022-12-06 00:00:00 ┆ -5.0  │
│ 6   ┆ 0.286651 ┆ 2022-12-07 00:00:00 ┆ -42.0 │
│ 7   ┆ 0.646312 ┆ 2022-12-08 00:00:00 ┆ null  │
└─────┴──────────┴─────────────────────┴───────┘
```

With `filter` you can also create more complex filters that include multiple columns.

Python Rust NodeJS

API `filter`

```
df.filter((pl.col("a") <= 3) & (pl.col("d").is_not_nan()))
```

API `filter`

```
let out = df.clone().lazy().filter(col("a").lt_eq(3).and(col("d").is_not_null())).collect()?;
println!("{}",out);
```

API `filter`

```
df.filter(pl.col("a").ltEq(3).and(pl.col("d").isNotNull()));
```

```
shape: (2, 4)
┌─────┬──────────┬─────────────────────┬─────┐
│ a   ┆ b        ┆ c                   ┆ d   │
│ --- ┆ ---      ┆ ---                 ┆ --- │
│ i64 ┆ f64      ┆ datetime[μs]        ┆ f64 │
╞═════╪══════════╪═════════════════════╪═════╡
│ 0   ┆ 0.577013 ┆ 2022-12-01 00:00:00 ┆ 1.0 │
│ 1   ┆ 0.114686 ┆ 2022-12-02 00:00:00 ┆ 2.0 │
└─────┴──────────┴─────────────────────┴─────┘
```

With_columns

`with_columns` allows you to create new columns for you analyses. We create two new columns `e` and `b+42`. First we sum all values from column `b` and store the results in column `e`. After that we add `42` to the values of `b`. Creating a new column `b+42` to store these results.

Python Rust NodeJS

API `with_columns`

```
df.with_columns([pl.col("b").sum().alias("e"), (pl.col("b") + 42).alias("b+42")])
```

API `with_columns`

```
let out = df.clone().lazy().with_columns(
    [
        col("b").sum().alias("e"),
        (col("b") + lit(42)).alias("b+42")
    ]
).collect()?;
println!("{}",out);
```

API `withColumns`

```
df.withColumns([
  pl.col("b").sum().alias("e"),
  pl.col("b").plus(42).alias("b+42"),
]);
```

```
shape: (8, 6)
┌─────┬──────────┬─────────────────────┬──────┬──────────┬───────────┐
│ a   ┆ b        ┆ c                   ┆ d    ┆ e        ┆ b+42      │
│ --- ┆ ---      ┆ ---                 ┆ ---  ┆ ---      ┆ ---       │
│ i64 ┆ f64      ┆ datetime[μs]        ┆ f64  ┆ f64      ┆ f64       │
╞═════╪══════════╪═════════════════════╪══════╪══════════╪═══════════╡
│ 0   ┆ 0.577013 ┆ 2022-12-01 00:00:00 ┆ 1.0  ┆ 3.142741 ┆ 42.577013 │
│ 1   ┆ 0.114686 ┆ 2022-12-02 00:00:00 ┆ 2.0  ┆ 3.142741 ┆ 42.114686 │
│ 2   ┆ 0.612896 ┆ 2022-12-03 00:00:00 ┆ NaN  ┆ 3.142741 ┆ 42.612896 │
│ 3   ┆ 0.342322 ┆ 2022-12-04 00:00:00 ┆ NaN  ┆ 3.142741 ┆ 42.342322 │
│ 4   ┆ 0.185987 ┆ 2022-12-05 00:00:00 ┆ 0.0  ┆ 3.142741 ┆ 42.185987 │
│ 5   ┆ 0.376874 ┆ 2022-12-06 00:00:00 ┆ -5.0 ┆ 3.142741 ┆ 42.376874 │
```

```
│ 6   ┆ 0.286651 ┆ 2022-12-07 00:00:00 ┆ -42.0 ┆ 3.142741 ┆ 42.286651 │
│ 7   ┆ 0.646312 ┆ 2022-12-08 00:00:00 ┆ null  ┆ 3.142741 ┆ 42.646312 │
└─────┴──────────┴─────────────────────┴───────┴──────────┴───────────┘
```

Groupby

We will create a new `DataFrame` for the Groupby functionality. This new `DataFrame` will include several 'groups' that we want to groupby.

Python Rust NodeJS

API `DataFrame`

```python
df2 = pl.DataFrame(
    {
        "x": np.arange(0, 8),
        "y": ["A", "A", "A", "B", "B", "C", "X", "X"],
    }
)
```

API `DataFrame`

```rust
let df2: DataFrame = df!("x" => 0..8,
                        "y"=> &["A", "A", "A", "B", "B", "C", "X", "X"],
                    ).expect("should not fail");
println!("{}",df2);
```

API `DataFrame`

```javascript
df2 = pl.DataFrame({
  x: [...Array(8).keys()],
  y: ["A", "A", "A", "B", "B", "C", "X", "X"],
});
```

```
shape: (8, 2)
┌─────┬─────┐
│ x   ┆ y   │
│ --- ┆ --- │
│ i64 ┆ str │
╞═════╪═════╡
│ 0   ┆ A   │
│ 1   ┆ A   │
│ 2   ┆ A   │
│ 3   ┆ B   │
│ 4   ┆ B   │
│ 5   ┆ C   │
│ 6   ┆ X   │
│ 7   ┆ X   │
└─────┴─────┘
```

Python Rust NodeJS

API `groupby`

```python
df2.groupby("y", maintain_order=True).count()
```

API `groupby`

```rust
let out = df2.clone().lazy().groupby(["y"]).agg([count()]).collect()?;
println!("{}",out);
```

API `groupBy`

```javascript
df2.groupBy("y").count();
console.log(df2);
```

```
shape: (4, 2)
┌─────┬───────┐
│ y   ┆ count │
│ --- ┆ ---   │
│ str ┆ u32   │
╞═════╪═══════╡
│ A   ┆ 3     │
│ B   ┆ 2     │
│ C   ┆ 1     │
│ X   ┆ 2     │
└─────┴───────┘
```

Python Rust NodeJS

API `groupby`

```python
df2.groupby("y", maintain_order=True).agg(
    [
        pl.col("*").count().alias("count"),
        pl.col("*").sum().alias("sum"),
    ]
)
```

API `groupby`

```rust
let out = df2.clone().lazy().groupby(["y"]).agg([
    col("*").count().alias("count"),
    col("*").sum().alias("sum"),
]).collect()?;
println!("{}",out);
```

API `groupBy`

```javascript
df2
  .groupBy("y")
  .agg(pl.col("*").count().alias("count"), pl.col("*").sum().alias("sum"));
```

```
shape: (4, 3)
┌─────┬───────┬─────┐
│ y   ┆ count ┆ sum │
│ --- ┆ ---   ┆ --- │
│ str ┆ u32   ┆ i64 │
╞═════╪═══════╪═════╡
│ A   ┆ 3     ┆ 3   │
│ B   ┆ 2     ┆ 7   │
│ C   ┆ 1     ┆ 5   │
│ X   ┆ 2     ┆ 13  │
└─────┴───────┴─────┘
```

Combining operations

Below are some examples on how to combine operations to create the `DataFrame` you require.

Python Rust NodeJS

API `select` · API `with_columns`

```
df_x = df.with_columns((pl.col("a") * pl.col("b")).alias("a * b")).select(
    [pl.all().exclude(["c", "d"])]
)

print(df_x)
```

API `select` · API `with_columns`

```
let out = df.clone().lazy().with_columns([
    (col("a") * col("b")).alias("a * b")
]).select([
    col("*").exclude(["c","d"])
]).collect()?;
println!("{}",out);
```

API `select` · API `withColumns`

```
df_x = df
  .withColumns(pl.col("a").mul(pl.col("b")).alias("a * b"))
  .select([pl.all().exclude(["c", "d"])]);

console.log(df_x);
```

```
shape: (8, 3)
┌─────┬──────────┬──────────┐
│ a   ┆ b        ┆ a * b    │
│ --- ┆ ---      ┆ ---      │
│ i64 ┆ f64      ┆ f64      │
╞═════╪══════════╪══════════╡
│ 0   ┆ 0.577013 ┆ 0.0      │
│ 1   ┆ 0.114686 ┆ 0.114686 │
│ 2   ┆ 0.612896 ┆ 1.225792 │
│ 3   ┆ 0.342322 ┆ 1.026967 │
│ 4   ┆ 0.185987 ┆ 0.743947 │
│ 5   ┆ 0.376874 ┆ 1.884371 │
│ 6   ┆ 0.286651 ┆ 1.719908 │
```

```
│ 7   ┆ 0.646312 ┆ 4.524185 │
└─────┴──────────┴──────────┘
```

Python Rust NodeJS

API `select` · API `with_columns`

```python
df_y = df.with_columns([(pl.col("a") * pl.col("b")).alias("a * b")]).select(
    [pl.all().exclude("d")]
)

print(df_y)
```

API `select` · API `with_columns`

```rust
let out = df.clone().lazy().with_columns([
    (col("a") * col("b")).alias("a * b")
]).select([
    col("*").exclude(["d"])
]).collect()?;
println!("{}",out);
```

API `select` · API `withColumns`

```javascript
df_y = df
  .withColumns([pl.col("a").mul(pl.col("b")).alias("a * b")])
  .select([pl.all().exclude("d")]);
console.log(df_y);
```

```
shape: (8, 4)
┌─────┬──────────┬─────────────────────┬──────────┐
│ a   ┆ b        ┆ c                   ┆ a * b    │
│ --- ┆ ---      ┆ ---                 ┆ ---      │
│ i64 ┆ f64      ┆ datetime[μs]        ┆ f64      │
╞═════╪══════════╪═════════════════════╪══════════╡
│ 0   ┆ 0.577013 ┆ 2022-12-01 00:00:00 ┆ 0.0      │
│ 1   ┆ 0.114686 ┆ 2022-12-02 00:00:00 ┆ 0.114686 │
│ 2   ┆ 0.612896 ┆ 2022-12-03 00:00:00 ┆ 1.225792 │
│ 3   ┆ 0.342322 ┆ 2022-12-04 00:00:00 ┆ 1.026967 │
│ 4   ┆ 0.185987 ┆ 2022-12-05 00:00:00 ┆ 0.743947 │
│ 5   ┆ 0.376874 ┆ 2022-12-06 00:00:00 ┆ 1.884371 │
│ 6   ┆ 0.286651 ┆ 2022-12-07 00:00:00 ┆ 1.719908 │
│ 7   ┆ 0.646312 ┆ 2022-12-08 00:00:00 ┆ 4.524185 │
└─────┴──────────┴─────────────────────┴──────────┘
```

3.6 Combining DataFrames

There are two ways `DataFrame` s can be combined depending on the use case: join and concat.

3.6.1 Join

Polars supports all types of join (e.g. left, right, inner, outer). Let's have a closer look on how to `join` two `DataFrames` into a single `DataFrame`. Our two `DataFrames` both have an 'id'-like column: `a` and `x`. We can use those columns to `join` the `DataFrames` in this example.

Python Rust NodeJS

API `join`

```
df = pl.DataFrame(
    {
        "a": np.arange(0, 8),
        "b": np.random.rand(8),
        "d": [1, 2.0, np.NaN, np.NaN, 0, -5, -42, None],
    }
)

df2 = pl.DataFrame(
    {
        "x": np.arange(0, 8),
        "y": ["A", "A", "A", "B", "B", "C", "X", "X"],
    }
)
joined = df.join(df2, left_on="a", right_on="x")
print(joined)
```

API `join`

```
use rand::Rng;
let mut rng = rand::thread_rng();

let df: DataFrame = df!("a" => 0..8,
                        "b"=> (0..8).map(|_| rng.gen::<f64>()).collect::<Vec<f64>>(),
                        "d"=> [Some(1.0), Some(2.0), None, None, Some(0.0), Some(-5.0), Some(-42.), None]
                    ).expect("should not fail");
let df2: DataFrame = df!("x" => 0..8,
                        "y"=> &["A", "A", "A", "B", "B", "C", "X", "X"],
                    ).expect("should not fail");
let joined = df.join(&df2,["a"],["x"],JoinType::Left,None)?;
println!("{}",joined);
```

API `join`

```
df = pl.DataFrame({
  a: [...Array(8).keys()],
  b: Array.from({ length: 8 }, () => Math.random()),
  d: [1, 2.0, null, null, 0, -5, -42, null],
});

df2 = pl.DataFrame({
  x: [...Array(8).keys()],
  y: ["A", "A", "A", "B", "B", "C", "X", "X"],
});
joined = df.join(df2, { leftOn: "a", rightOn: "x" });
console.log(joined);
```

```
shape: (8, 4)
┌─────┬──────────┬───────┬─────┐
│ a   ┆ b        ┆ d     ┆ y   │
│ --- ┆ ---      ┆ ---   ┆ --- │
│ i64 ┆ f64      ┆ f64   ┆ str │
╞═════╪══════════╪═══════╪═════╡
│ 0   ┆ 0.253982 ┆ 1.0   ┆ A   │
│ 1   ┆ 0.396669 ┆ 2.0   ┆ A   │
│ 2   ┆ 0.131986 ┆ NaN   ┆ A   │
│ 3   ┆ 0.143672 ┆ NaN   ┆ B   │
│ 4   ┆ 0.356455 ┆ 0.0   ┆ B   │
│ 5   ┆ 0.111859 ┆ -5.0  ┆ C   │
│ 6   ┆ 0.399095 ┆ -42.0 ┆ X   │
│ 7   ┆ 0.150452 ┆ null  ┆ X   │
└─────┴──────────┴───────┴─────┘
```

To see more examples with other types of joins, go the User Guide.

3.6.2 Concat

We can also `concatenate` two `DataFrames`. Vertical concatenation will make the `DataFrame` longer. Horizontal concatenation will make the `DataFrame` wider. Below you can see the result of an horizontal concatenation of our two `DataFrames`.

Python Rust NodeJS

API `hstack`

```
stacked = df.hstack(df2)
print(stacked)
```

API `hstack`

```
let stacked = df.hstack(df2.get_columns())?;
println!("{}",stacked);
```

API `hstack`

```
stacked = df.hstack(df2);
console.log(stacked);
```

```
shape: (8, 5)
┌─────┬──────────┬───────┬─────┬─────┐
│ a   ┆ b        ┆ d     ┆ x   ┆ y   │
│ --- ┆ ---      ┆ ---   ┆ --- ┆ --- │
│ i64 ┆ f64      ┆ f64   ┆ i64 ┆ str │
╞═════╪══════════╪═══════╪═════╪═════╡
│ 0   ┆ 0.253982 ┆ 1.0   ┆ 0   ┆ A   │
│ 1   ┆ 0.396669 ┆ 2.0   ┆ 1   ┆ A   │
│ 2   ┆ 0.131986 ┆ NaN   ┆ 2   ┆ A   │
│ 3   ┆ 0.143672 ┆ NaN   ┆ 3   ┆ B   │
│ 4   ┆ 0.356455 ┆ 0.0   ┆ 4   ┆ B   │
│ 5   ┆ 0.111859 ┆ -5.0  ┆ 5   ┆ C   │
│ 6   ┆ 0.399095 ┆ -42.0 ┆ 6   ┆ X   │
│ 7   ┆ 0.150452 ┆ null  ┆ 7   ┆ X   │
└─────┴──────────┴───────┴─────┴─────┘
```

4. User Guide

4.1 Introduction

This User Guide is an introduction to the `Polars` DataFrame library. Its goal is to introduce you to `Polars` by going through examples and comparing it to other solutions. Some design choices are introduced here. The guide will also introduce you to optimal usage of `Polars`.

Even though `Polars` is completely written in `Rust` (no runtime overhead!) and uses `Arrow` -- the native arrow2 `Rust` implementation -- as its foundation, the examples presented in this guide will be mostly using its higher-level language bindings. Higher-level bindings only serve as a thin wrapper for functionality implemented in the core library.

For `Pandas` users, our Python package will offer the easiest way to get started with `Polars`.

Philosophy

The goal of `Polars` is to provide a lightning fast `DataFrame` library that:

- Utilizes all available cores on your machine.
- Optimizes queries to reduce unneeded work/memory allocations.
- Handles datasets much larger than your available RAM.
- Has an API that is consistent and predictable.
- Has a strict schema (data-types should be known before running the query).

Polars is written in Rust which gives it C/C++ performance and allows it to fully control performance critical parts in a query engine.

As such `Polars` goes to great lengths to:

- Reduce redundant copies.
- Traverse memory cache efficiently.
- Minimize contention in parallelism.
- Process data in chunks.
- Reuse memory allocations.

4.2 Installation

Polars is a library and installation is as simple as invoking the package manager of the corresponding programming language.

Python Rust NodeJS

```
pip install polars

cargo add polars -F lazy

# Or Cargo.toml
[dependencies]
polars = { version = "x", features = ["lazy", ...]}

yarn add nodejs-polars
```

4.2.1 Importing

To use the library import it into your project

Python Rust NodeJS

```
import polars as pl

use polars::prelude::*;

// esm
import pl from 'nodejs-polars';

// require
const pl = require('nodejs-polars');
```

4.2.2 Feature Flags

By using the above command you install the core of `Polars` onto your system. However depending on your use case you might want to install the optional dependencies as well. These are made optional to minimize the footprint. The flags are different depending on the programming language. Throughout the user guide we will mention when a functionality is used that requires an additional dependency.

Python

```
# For example
pip install polars[numpy, fsspec]
```

Tag	Description
all	Install all optional dependencies (all of the following)
pandas	Install with Pandas for converting data to and from Pandas Dataframes/Series
numpy	Install with numpy for converting data to and from numpy arrays
pyarrow	Reading data formats using PyArrow
fsspec	Support for reading from remote file systems
connectorx	Support for reading from SQL databases
xlsx2csv	Support for reading from Excel files
deltalake	Support for reading from Delta Lake Tables
timezone	Timezone support, only needed if 1. you are on Python < 3.9 and/or 2. you are on Windows, otherwise no dependencies will be installed

Rust

```
# Cargo.toml
[dependencies]
polars = { version = "0.26.1", features = ["lazy","temporal","describe","json","parquet","dtype-datetime"]}
```

The opt-in features are:

- Additional data types:
- `dtype-date`
- `dtype-datetime`
- `dtype-time`
- `dtype-duration`
- `dtype-i8`
- `dtype-i16`
- `dtype-u8`
- `dtype-u16`
- `dtype-categorical`
- `dtype-struct`
- `performant` - Longer compile times more fast paths.
- `lazy` - Lazy API
- `lazy_regex` - Use regexes in column selection
- `dot_diagram` - Create dot diagrams from lazy logical plans.
- `sql` - Pass SQL queries to polars.
- `streaming` - Be able to process datasets that are larger than RAM.
- `random` - Generate arrays with randomly sampled values
- `ndarray`- Convert from `DataFrame` to `ndarray`
- `temporal` - Conversions between Chrono and Polars for temporal data types
- `timezones` - Activate timezone support.
- `strings` - Extra string utilities for `Utf8Chunked`
- `string_justify` - `zfill`, `ljust`, `rjust`
- `string_from_radix` - `parse_int`
- `object` - Support for generic ChunkedArrays called `ObjectChunked<T>` (generic over `T`). These are downcastable from Series through the Any trait.
- Performance related:
- `nightly` - Several nightly only features such as SIMD and specialization.
- `performant` - more fast paths, slower compile times.
- `bigidx` - Activate this feature if you expect >> 2^32 rows. This has not been needed by anyone. This allows polars to scale up way beyond that by using `u64` as an index. Polars will be a bit slower with this feature activated as many data structures are less cache efficient.
- `cse` - Activate common subplan elimination optimization
- IO related:
- `serde` - Support for serde serialization and deserialization. Can be used for JSON and more serde supported serialization formats.
- `serde-lazy` - Support for serde serialization and deserialization. Can be used for JSON and more serde supported serialization formats.
- `parquet` - Read Apache Parquet format
- `json` - JSON serialization
- `ipc` - Arrow's IPC format serialization
- `decompress` - Automatically infer compression of csvs and decompress them. Supported compressions: - zip - gzip
- `DataFrame` operations:
- `dynamic_groupby` - Groupby based on a time window instead of predefined keys. Also activates rolling window group by operations.
- `sort_multiple` - Allow sorting a `DataFrame` on multiple columns
- `rows` - Create `DataFrame` from rows and extract rows from `DataFrames`. And activates `pivot` and `transpose` operations
- `join_asof` - Join ASOF, to join on nearest keys instead of exact equality match.
- `cross_join` - Create the cartesian product of two DataFrames.
- `semi_anti_join` - SEMI and ANTI joins.
- `groupby_list` - Allow groupby operation on keys of type List.
- `row_hash` - Utility to hash DataFrame rows to UInt64Chunked
- `diagonal_concat` - Concat diagonally thereby combining different schemas.
- `horizontal_concat` - Concat horizontally and extend with null values if lengths don't match
- `dataframe_arithmetic` - Arithmetic on (Dataframe and DataFrames) and (DataFrame on Series)
- `partition_by` - Split into multiple DataFrames partitioned by groups.

- `Series` / `Expression` operations:
- `is_in` - Check for membership in `Series`
- `zip_with` - Zip two Series/ ChunkedArrays
- `round_series` - round underlying float types of `Series`.
- `repeat_by` - [Repeat element in an Array N times, where N is given by another array.
- `is_first` - Check if element is first unique value.
- `is_last` - Check if element is last unique value.
- `checked_arithmetic` - checked arithmetic/ returning `None` on invalid operations.
- `dot_product` - Dot/inner product on Series and Expressions.
- `concat_str` - Concat string data in linear time.
- `reinterpret` - Utility to reinterpret bits to signed/unsigned
- `take_opt_iter` - Take from a Series with `Iterator<Item=Option<usize>>`
- `mode` - Return the most occurring value(s)
- `cum_agg` - cumsum, cummin, cummax aggregation.
- `rolling_window` - rolling window functions, like rolling_mean
- `interpolate` interpolate None values
- `extract_jsonpath` - Run jsonpath queries on Utf8Chunked
- `list` - List utils.
- `list_take` take sublist by multiple indices
- `rank` - Ranking algorithms.
- `moment` - kurtosis and skew statistics
- `ewma` - Exponential moving average windows
- `abs` - Get absolute values of Series
- `arange` - Range operation on Series
- `product` - Compute the product of a Series.
- `diff` - `diff` operation.
- `pct_change` - Compute change percentages.
- `unique_counts` - Count unique values in expressions.
- `log` - Logarithms for `Series`.
- `list_to_struct` - Convert `List` to `Struct` dtypes.
- `list_count` - Count elements in lists.
- `list_eval` - Apply expressions over list elements.
- `cumulative_eval` - Apply expressions over cumulatively increasing windows.
- `arg_where` - Get indices where condition holds.
- `search_sorted` - Find indices where elements should be inserted to maintain order.
- `date_offset` Add an offset to dates that take months and leap years into account.
- `trigonometry` Trigonometric functions.
- `sign` Compute the element-wise sign of a Series.
- `propagate_nans` NaN propagating min/max aggregations.
- `DataFrame` pretty printing
- `fmt` - Activate DataFrame formatting

4.3 Concepts

4.3.1 Data types

`Polars` is entirely based on `Arrow` data types and backed by `Arrow` memory arrays. This makes data processing cache-efficient and well-supported for Inter Process Communication. Most data types follow the exact implementation from `Arrow`, with the exception of `Utf8` (this is actually `LargeUtf8`), `Categorical`, and `Object` (support is limited). The data types are:

Group	Type	Details
Numeric	`Int8`	8-bit signed integer.
	`Int16`	16-bit signed integer.
	`Int32`	32-bit signed integer.
	`Int64`	64-bit signed integer.
	`UInt8`	8-bit unsigned integer.
	`UInt16`	16-bit unsigned integer.
	`UInt32`	32-bit unsigned integer.
	`UInt64`	64-bit unsigned integer.
	`Float32`	32-bit floating point.
	`Float64`	64-bit floating point.
Nested	`Struct`	A struct array is represented as a `Vec<Series>` and is useful to pack multiple/heterogenous values in a single column.
	`List`	A list array contains a child array containing the list values and an offset array. (this is actually `Arrow` `LargeList` internally).
Temporal	`Date`	Date representation, internally represented as days since UNIX epoch encoded by a 32-bit signed integer.
	`Datetime`	Datetime representation, internally represented as microseconds since UNIX epoch encoded by a 64-bit signed integer.
	`Duration`	A timedelta type, internally represented as microseconds. Created when subtracting `Date/Datetime`.
	`Time`	Time representation, internally represented as nanoseconds since midnight.
Other	`Boolean`	Boolean type effectively bit packed.
	`Utf8`	String data (this is actually `Arrow` `LargeUtf8` internally).
	`Binary`	Store data as bytes.
	`Object`	A limited supported data type that can be any value.
	`Categorical`	A categorical encoding of a set of strings.

To learn more about the internal representation of these data types, check the `Arrow` columnar format.

4.3.2 Data Structures

The core base data structures provided by Polars are `Series` and `DataFrames`.

Series

Series are a 1-dimensional data structure. Within a series all elements have the same Data Type . The snippet below shows how to create a simple named `Series` object.

Python Rust NodeJS

API `Series`

```python
import polars as pl

s = pl.Series("a", [1, 2, 3, 4, 5])
print(s)
```

API `Series`

```rust
use chrono::prelude::*;

let s = Series::new("a", [1, 2, 3, 4, 5]);
println!("{}",s);
```

API `Series`

```javascript
const pl = require("nodejs-polars");

var s = pl.Series("a", [1, 2, 3, 4, 5]);
console.log(s);
```

```
shape: (5,)
Series: 'a' [i64]
[
    1
    2
    3
    4
    5
]
```

DataFrame

A `DataFrame` is a 2-dimensional data structure that is backed by a `Series`, and it can be seen as an abstraction of a collection (e.g. list) of `Series`. Operations that can be executed on a `DataFrame` are very similar to what is done in a `SQL` like query. You can `GROUP BY`, `JOIN`, `PIVOT`, but also define custom functions.

Python Rust NodeJS

API `DataFrame`

```python
from datetime import datetime

df = pl.DataFrame(
    {
        "integer": [1, 2, 3, 4, 5],
        "date": [
            datetime(2022, 1, 1),
            datetime(2022, 1, 2),
            datetime(2022, 1, 3),
            datetime(2022, 1, 4),
            datetime(2022, 1, 5),
        ],
        "float": [4.0, 5.0, 6.0, 7.0, 8.0],
    }
)

print(df)
```

API `DataFrame`

```rust
let df: DataFrame = df!("integer" => &[1, 2, 3, 4, 5],
                        "date" => &[
                            NaiveDate::from_ymd_opt(2022, 1, 1).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                            NaiveDate::from_ymd_opt(2022, 1, 2).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                            NaiveDate::from_ymd_opt(2022, 1, 3).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                            NaiveDate::from_ymd_opt(2022, 1, 4).unwrap().and_hms_opt(0, 0, 0).unwrap(),
                            NaiveDate::from_ymd_opt(2022, 1, 5).unwrap().and_hms_opt(0, 0, 0).unwrap()
                        ],
                        "float" => &[4.0, 5.0, 6.0, 7.0, 8.0]
                        ).expect("should not fail");
println!("{}",df);
```

API `DataFrame`

```javascript
let df = pl.DataFrame({
  integer: [1, 2, 3, 4, 5],
  date: [
    new Date(2022, 1, 1, 0, 0),
    new Date(2022, 1, 2, 0, 0),
    new Date(2022, 1, 3, 0, 0),
    new Date(2022, 1, 4, 0, 0),
    new Date(2022, 1, 5, 0, 0),
  ],
  float: [4.0, 5.0, 6.0, 7.0, 8.0],
});
console.log(df);
```

```
shape: (5, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 1       ┆ 2022-01-01 00:00:00 ┆ 4.0   │
│ 2       ┆ 2022-01-02 00:00:00 ┆ 5.0   │
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
│ 4       ┆ 2022-01-04 00:00:00 ┆ 7.0   │
│ 5       ┆ 2022-01-05 00:00:00 ┆ 8.0   │
└─────────┴─────────────────────┴───────┘
```

VIEWING DATA

This part focuses on viewing data in a `DataFrame`. We will use the `DataFrame` from the previous example as a starting point.

Head

The `head` function shows by default the first 5 rows of a `DataFrame`. You can specify the number of rows you want to see (e.g. `df.head(10)`).

Python Rust NodeJS

API `head`

```
print(df.head(3))
```

API `head`

```
println!("{}",df.head(Some(3)));
```

API `head`

```
console.log(df.head(3));
```

```
shape: (3, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 1       ┆ 2022-01-01 00:00:00 ┆ 4.0   │
│ 2       ┆ 2022-01-02 00:00:00 ┆ 5.0   │
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
└─────────┴─────────────────────┴───────┘
```

Tail

The `tail` function shows the last 5 rows of a `DataFrame`. You can also specify the number of rows you want to see, similar to `head`.

Python Rust NodeJS

API `tail`

```
print(df.tail(3))
```

API `tail`

```
println!("{}",df.tail(Some(3)));
```

API `tail`

```
console.log(df.tail(3));
```

```
shape: (3, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 3       ┆ 2022-01-03 00:00:00 ┆ 6.0   │
│ 4       ┆ 2022-01-04 00:00:00 ┆ 7.0   │
│ 5       ┆ 2022-01-05 00:00:00 ┆ 8.0   │
└─────────┴─────────────────────┴───────┘
```

Sample

If you want to get an impression of the data of your `DataFrame`, you can also use `sample`. With `sample` you get an *n* number of random rows from the `DataFrame`.

Python Rust NodeJS

API `sample`

```
print(df.sample(2))
```

API `sample_n`

```
println!("{}",df.sample_n(2, false, true, None)?);
```

API `sample`

```
console.log(df.sample(2));
```

```
shape: (2, 3)
┌─────────┬─────────────────────┬───────┐
│ integer ┆ date                ┆ float │
│ ---     ┆ ---                 ┆ ---   │
│ i64     ┆ datetime[μs]        ┆ f64   │
╞═════════╪═════════════════════╪═══════╡
│ 4       ┆ 2022-01-04 00:00:00 ┆ 7.0   │
│ 5       ┆ 2022-01-05 00:00:00 ┆ 8.0   │
└─────────┴─────────────────────┴───────┘
```

Describe

`Describe` returns summary statistics of your `DataFrame`. It will provide several quick statistics if possible.

Python Rust NodeJS

API `describe`

```
print(df.describe())
```

API `describe` · Available on feature describe

```
println!("{}",df.describe(None));
```

API `describe`

```
console.log(df.describe());
```

```
shape: (9, 4)
┌────────────┬──────────┬─────────────────────┬──────────┐
│ describe   ┆ integer  ┆ date                ┆ float    │
│ ---        ┆ ---      ┆ ---                 ┆ ---      │
│ str        ┆ f64      ┆ str                 ┆ f64      │
╞════════════╪══════════╪═════════════════════╪══════════╡
│ count      ┆ 5.0      ┆ 5                   ┆ 5.0      │
│ null_count ┆ 0.0      ┆ 0                   ┆ 0.0      │
│ mean       ┆ 3.0      ┆ null                ┆ 6.0      │
│ std        ┆ 1.581139 ┆ null                ┆ 1.581139 │
│ min        ┆ 1.0      ┆ 2022-01-01 00:00:00 ┆ 4.0      │
│ max        ┆ 5.0      ┆ 2022-01-05 00:00:00 ┆ 8.0      │
│ median     ┆ 3.0      ┆ null                ┆ 6.0      │
│ 25%        ┆ 2.0      ┆ null                ┆ 5.0      │
│ 75%        ┆ 4.0      ┆ null                ┆ 7.0      │
└────────────┴──────────┴─────────────────────┴──────────┘
```

4.3.3 Contexts

Polars has developed its own Domain Specific Language (DSL) for transforming data. The language is very easy to use and allows for complex queries that remain human readable. The two core components of the language are Contexts and Expressions, the latter we will cover in the next section.

A context, as implied by the name, refers to the context in which an expression needs to be evaluated. There are three main contexts [1]:

1. Selection: `df.select([..])` , `df.with_columns([..])`
2. Filtering: `df.filter()`
3. Groupby / Aggregation: `df.groupby(..).agg([..])`

The examples below are performed on the following `DataFrame` :

Python Rust NodeJS

API `DataFrame`

```python
df = pl.DataFrame(
    {
        "nrs": [1, 2, 3, None, 5],
        "names": ["foo", "ham", "spam", "egg", None],
        "random": np.random.rand(5),
        "groups": ["A", "A", "B", "C", "B"],
    }
)
print(df)
```

API `DataFrame`

```rust
use rand::{thread_rng, Rng};

let mut arr = [0f64; 5];
thread_rng().fill(&mut arr);

let df = df! (
    "nrs" => &[Some(1), Some(2), Some(3), None, Some(5)],
    "names" => &[Some("foo"), Some("ham"), Some("spam"), Some("eggs"), None],
    "random" => &arr,
    "groups" => &["A", "A", "B", "C", "B"],
)?;

println!("{}", &df);
```

API `DataFrame`

```javascript
const arr = Array.from({ length: 5 }).map((_) =>
  chance.floating({ min: 0, max: 1 }),
);

let df = pl.DataFrame({
  nrs: [1, 2, 3, null, 5],
  names: ["foo", "ham", "spam", "egg", null],
  random: arr,
  groups: ["A", "A", "B", "C", "B"],
});
console.log(df);
```

```
shape: (5, 4)
┌──────┬───────┬──────────┬────────┐
│ nrs  ┆ names ┆ random   ┆ groups │
│ ---  ┆ ---   ┆ ---      ┆ ---    │
│ i64  ┆ str   ┆ f64      ┆ str    │
╞══════╪═══════╪══════════╪════════╡
│ 1    ┆ foo   ┆ 0.154163 ┆ A      │
│ 2    ┆ ham   ┆ 0.74005  ┆ A      │
│ 3    ┆ spam  ┆ 0.263315 ┆ B      │
│ null ┆ egg   ┆ 0.533739 ┆ C      │
│ 5    ┆ null  ┆ 0.014575 ┆ B      │
└──────┴───────┴──────────┴────────┘
```

Select

In the `select` context the selection applies expressions over columns. The expressions in this context must produce `Series` that are all the same length or have a length of 1.

A `Series` of a length of 1 will be broadcasted to match the height of the `DataFrame`. Note that a select may produce new columns that are aggregations, combinations of expressions, or literals.

Python Rust NodeJS

API `select`

```
out = df.select(
    pl.sum("nrs"),
    pl.col("names").sort(),
    pl.col("names").first().alias("first name"),
    (pl.mean("nrs") * 10).alias("10xnrs"),
)
print(out)
```

API `select`

```
let out = df
    .clone()
    .lazy()
    .select([
        sum("nrs"),
        col("names").sort(false),
        col("names").first().alias("first name"),
        (mean("nrs") * lit(10)).alias("10xnrs"),
    ])
    .collect()?;
println!("{}", out);
```

API `select`

```
let out = df.select(
  pl.col("nrs").sum(),
  pl.col("names").sort(),
  pl.col("names").first().alias("first name"),
  pl.mean("nrs").multiplyBy(10).alias("10xnrs"),
);
console.log(out);
```

```
shape: (5, 4)
┌─────┬───────┬────────────┬────────┐
│ nrs ┆ names ┆ first name ┆ 10xnrs │
│ --- ┆ ---   ┆ ---        ┆ ---    │
│ i64 ┆ str   ┆ str        ┆ f64    │
╞═════╪═══════╪════════════╪════════╡
│ 11  ┆ null  ┆ foo        ┆ 27.5   │
│ 11  ┆ egg   ┆ foo        ┆ 27.5   │
│ 11  ┆ foo   ┆ foo        ┆ 27.5   │
│ 11  ┆ ham   ┆ foo        ┆ 27.5   │
│ 11  ┆ spam  ┆ foo        ┆ 27.5   │
└─────┴───────┴────────────┴────────┘
```

As you can see from the query the `select` context is very powerful and allows you to perform arbitrary expressions independent (and in parallel) of each other.

Similarly to the `select` statement there is the `with_columns` statement which also is an entrance to the selection context. The main difference is that `with_columns` retains the original columns and adds new ones while `select` drops the original columns.

Python Rust NodeJS

API `with_columns`

```
df = df.with_columns(
    pl.sum("nrs").alias("nrs_sum"),
    pl.col("random").count().alias("count"),
)
print(df)
```

API `with_columns`

```
let out = df
    .clone()
    .lazy()
    .with_columns([
        sum("nrs").alias("nrs_sum"),
        col("random").count().alias("count"),
    ])
    .collect()?;
println!("{}", out);
```

API `withColumns`

```
df = df.withColumns(
  pl.col("nrs").sum().alias("nrs_sum"),
  pl.col("random").count().alias("count"),
);

console.log(df);
```

```
shape: (5, 6)
┌──────┬───────┬──────────┬────────┬─────────┬───────┐
│ nrs  ┆ names ┆ random   ┆ groups ┆ nrs_sum ┆ count │
│ ---  ┆ ---   ┆ ---      ┆ ---    ┆ ---     ┆ ---   │
│ i64  ┆ str   ┆ f64      ┆ str    ┆ i64     ┆ u32   │
╞══════╪═══════╪══════════╪════════╪═════════╪═══════╡
│ 1    ┆ foo   ┆ 0.154163 ┆ A      ┆ 11      ┆ 5     │
│ 2    ┆ ham   ┆ 0.74005  ┆ A      ┆ 11      ┆ 5     │
│ 3    ┆ spam  ┆ 0.263315 ┆ B      ┆ 11      ┆ 5     │
│ null ┆ egg   ┆ 0.533739 ┆ C      ┆ 11      ┆ 5     │
│ 5    ┆ null  ┆ 0.014575 ┆ B      ┆ 11      ┆ 5     │
└──────┴───────┴──────────┴────────┴─────────┴───────┘
```

Filter

In the `filter` context you filter the existing dataframe based on arbritary expression which evaluates to the `Boolean` data type.

Python Rust NodeJS

API `filter`

```
out = df.filter(pl.col("nrs") > 2)
print(out)
```

API `filter`

```
let out = df.clone().lazy().filter(col("nrs").gt(lit(2))).collect()?;
println!("{}", out);
```

API `filter`

```
out = df.filter(pl.col("nrs").gt(2));
console.log(out);
```

```
shape: (2, 6)
┌─────┬───────┬──────────┬────────┬─────────┬───────┐
```

```
| nrs | names | random   | groups | nrs_sum | count |
| --- | ---   | ---      | ---    | ---     | ---   |
| i64 | str   | f64      | str    | i64     | u32   |
╞═════╪═══════╪══════════╪════════╪═════════╪═══════╡
| 3   | spam  | 0.263315 | B      | 11      | 5     |
| 5   | null  | 0.014575 | B      | 11      | 5     |
└─────┴───────┴──────────┴────────┴─────────┴───────┘
```

Groupby / Aggregation

In the `groupby` context expressions work on groups and thus may yield results of any length (a group may have many members).

Python Rust NodeJS

API `groupby`

```python
out = df.groupby("groups").agg(
    pl.sum("nrs"),  # sum nrs by groups
    pl.col("random").count().alias("count"),  # count group members
    # sum random where name != null
    pl.col("random").filter(pl.col("names").is_not_null()).sum().suffix("_sum"),
    pl.col("names").reverse().alias(("reversed names")),
)
print(out)
```

API `groupby`

```rust
let out = df
    .lazy()
    .groupby([col("groups")])
    .agg([
        sum("nrs"),                           // sum nrs by groups
        col("random").count().alias("count"), // count group members
        // sum random where name != null
        col("random")
            .filter(col("names").is_not_null())
            .sum()
            .suffix("_sum"),
        col("names").reverse().alias("reversed names"),
    ])
    .collect()?;
println!("{}", out);
```

API `groupBy`

```javascript
out = df.groupBy("groups").agg(
  pl
    .col("nrs")
    .sum(), // sum nrs by groups
  pl
    .col("random")
    .count()
    .alias("count"), // count group members
  // sum random where name != null
  pl
    .col("random")
    .filter(pl.col("names").isNotNull())
    .sum()
    .suffix("_sum"),
  pl.col("names").reverse().alias("reversed names"),
);
console.log(out);
```

```
shape: (3, 5)
┌────────┬──────┬───────┬────────────┬────────────────┐
| groups | nrs  | count | random_sum | reversed names |
| ---    | ---  | ---   | ---        | ---            |
| str    | i64  | u32   | f64        | list[str]      |
╞════════╪══════╪═══════╪════════════╪════════════════╡
| A      | 3    | 2     | 0.894213   | ["ham", "foo"] |
| B      | 8    | 2     | 0.263315   | [null, "spam"] |
| C      | null | 1     | 0.533739   | ["egg"]        |
└────────┴──────┴───────┴────────────┴────────────────┘
```

As you can see from the result all expressions are applied to the group defined by the `groupby` context. Besides the standard `groupby`, `groupby_dynamic`, and `groupby_rolling` are also entrances to the groupby context.

1. There are additional List and SQL contexts which are covered later in this guide. But for simplicity, we leave them out of scope for now. ↩

4.3.4 Expressions

`Polars` has a powerful concept called expressions that is central to its very fast performance.

Expressions are at the core of many data science operations:

- taking a sample of rows from a column
- multiplying values in a column
- extracting a column of years from dates
- convert a column of strings to lowercase
- and so on!

However, expressions are also used within other operations:

- taking the mean of a group in a `groupby` operation
- calculating the size of groups in a `groupby` operation
- taking the sum horizontally across columns

`Polars` performs these core data transformations very quickly by:

- automatic query optimization on each expression
- automatic parallelization of expressions on many columns

Polars expressions are a mapping from a series to a series (or mathematically `Fn(Series) -> Series`). As expressions have a `Series` as an input and a `Series` as an output then it is straightforward to do a sequence of expressions (similar to method chaining in `Pandas`).

Examples

The following is an expression:

Python Rust NodeJS

API `col` · API `sort` · API `head`

```
pl.col("foo").sort().head(2)
```

API `col` · API `sort` · API `head`

```
df.column("foo")?.sort(false).head(Some(2));
```

API `head`

```
pl.col("foo").sort().head(2);
```

The snippet above says:

1. Select column "foo"
2. Then sort the column (not in reversed order)
3. Then take the first two values of the sorted output

The power of expressions is that every expression produces a new expression, and that they can be *piped* together. You can run an expression by passing them to one of `Polars` execution contexts.

Here we run two expressions by running `df.select` :

Python Rust NodeJS

API `select`

```
df.select(pl.col("foo").sort().head(2), pl.col("bar").filter(pl.col("foo") == 1).sum())
```

API `select`

```
df.clone().lazy().select([
    col("foo").sort(Default::default()).head(Some(2)),
    col("bar").filter(col("foo").eq(lit(1))).sum(),
 ]).collect()?;
```

API `select`

```
df.select(
  pl.col("foo").sort().head(2),
  pl.col("bar").filter(pl.col("foo").eq(1)).sum(),
);
```

All expressions are run in parallel, meaning that separate `Polars` expressions are **embarrassingly parallel**. Note that within an expression there may be more parallelization going on.

Conclusion

This is the tip of the iceberg in terms of possible expressions. There are a ton more, and they can be combined in a variety of ways. This page is intended to get you familiar with the concept of expressions, in the section on expressions we will dive deeper.

4.3.5 Lazy / Eager API

`Polars` supports two modes of operation: lazy and eager. In the eager API the query is executed immediately while in the lazy API the query is only evaluated once it is 'needed'. Deferring the execution to the last minute can have significant performance advantages that is why the Lazy API is preferred in most cases. Let us demonstrate this with an example:

Python Rust NodeJS

API `read_csv`

```python
df = pl.read_csv("docs/src/data/iris.csv")
df_small = df.filter(pl.col("sepal_length") > 5)
df_agg = df_small.groupby("species").agg(pl.col("sepal_width").mean())
print(df_agg)
```

API `CsvReader` · Available on feature csv

```rust
let df = CsvReader::from_path("docs/src/data/iris.csv").unwrap().finish().unwrap();
let mask = df.column("sepal_width")?.f64()?.gt(5.0);
let df_small = df.filter(&mask)?;
let df_agg = df_small.groupby(["species"])?.select(["sepal_width"]).mean()?;
println!("{}", df_agg);
```

API `readCSV`

```javascript
df = pl.readCSV("docs/src/data/iris.csv");
df_small = df.filter(pl.col("sepal_length").gt(5));
df_agg = df_small.groupBy("species").agg(pl.col("sepal_width").mean());
console.log(df_agg);
```

In this example we use the eager API to:

1. Read the iris dataset.
2. Filter the dataset based on sepal length
3. Calculate the mean of the sepal width per species

Every step is executed immediately returning the intermediate results. This can be very wasteful as we might do work or load extra data that is not being used. If we instead used the lazy API and waited on execution until all the steps are defined then the query planner could perform various optimizations. In this case:

- Predicate pushdown: Apply filters as early as possible while reading the dataset, thus only reading rows with sepal length greater than 5.
- Projection pushdown: Select only the columns that are needed while reading the dataset, thus removing the need to load additional columns (e.g. petal length & petal width)

Python Rust NodeJS

API `scan_csv`

```
q = (
    pl.scan_csv("docs/src/data/iris.csv")
    .filter(pl.col("sepal_length") > 5)
    .groupby("species")
    .agg(pl.col("sepal_width").mean())
)

df = q.collect()
```

API `LazyCsvReader` · Available on feature csv

```
let q = LazyCsvReader::new("docs/src/data/iris.csv")
    .has_header(true)
    .finish()?
    .filter(col("sepal_length").gt(lit(5)))
    .groupby(vec![col("species")])
    .agg([col("sepal_width").mean()]);
let df = q.collect()?;
println!("{}", df);
```

API `scanCSV`

```
q = pl
  .scanCSV("docs/src/data/iris.csv")
  .filter(pl.col("sepal_length").gt(5))
  .groupBy("species")
  .agg(pl.col("sepal_width").mean());

df = q.collect();
```

These will significantly lower the load on memory & CPU thus allowing you to fit bigger datasets in memory and process faster. Once the query is defined you call `collect` to inform `Polars` that you want to execute it. In the section on Lazy API we will go into more details on its implementation.

> **Eager API**
>
> In many cases the eager API is actually calling the lazy API under the hood and immediately collecting the result. This has the benefit that within the query itself optimization(s) made by the query planner can still take place.

WHEN TO USE WHICH

In general the lazy API should be preferred unless you are either interested in the intermediate results or are doing exploratory work and don't know yet what your query is going to look like.

4.3.6 Streaming API

One additional benefit of the lazy API is that it allows queries to be executed in a streaming manner. Instead of processing the data all-at-once `Polars` can execute the query in batches allowing you to process datasets that are larger-than-memory.

To tell Polars we want to execute a query in streaming mode we pass the `streaming=True` argument to `collect`

Python Rust

API `collect`

```
q = (
    pl.scan_csv("docs/src/data/iris.csv")
    .filter(pl.col("sepal_length") > 5)
    .groupby("species")
    .agg(pl.col("sepal_width").mean())
)

df = q.collect(streaming=True)
```

API `collect` · Available on feature streaming

```
let q = LazyCsvReader::new("docs/src/data/iris.csv")
    .has_header(true)
    .finish()?
    .filter(col("sepal_length").gt(lit(5)))
    .groupby(vec![col("species")])
    .agg([col("sepal_width").mean()]);

let df = q.with_streaming(true).collect()?;
println!("{}", df);
```

When is streaming available?

Streaming is still in development. We can ask Polars to execute any lazy query in streaming mode. However, not all lazy operations support streaming. If there is an operation for which streaming is not supported Polars will run the query in non-streaming mode.

Streaming is supported for many operations including:

- `filter`, `slice`, `head`, `tail`
- `with_columns`, `select`
- `groupby`
- `join`
- `sort`
- `explode`, `melt`
- `scan_csv`, `scan_parquet`, `scan_ipc`

4.4 Expressions

4.4.1 Basic Operators

This section describes how to use basic operators (e.g. addition, substraction) in conjunction with Expressions. We will provide various examples using different themes in the context of the following dataframe.

> **Note**
>
> In Rust and Python it is possible to use the operators directly (as in `+ - * / < >`) as the language allows operator overloading. For instance, the operator `+` translates to the `.add()` method. In NodeJS this is not possible and you must use the methods themselves, in python and rust you can choose which one you prefer.

Python

API `DataFrame`

```
df = pl.DataFrame(
    {
        "nrs": [1, 2, 3, None, 5],
        "names": ["foo", "ham", "spam", "egg", None],
        "random": np.random.rand(5),
        "groups": ["A", "A", "B", "C", "B"],
    }
)
print(df)
```

```
shape: (5, 4)
┌──────┬───────┬──────────┬────────┐
│ nrs  ┆ names ┆ random   ┆ groups │
│ ---  ┆ ---   ┆ ---      ┆ ---    │
│ i64  ┆ str   ┆ f64      ┆ str    │
╞══════╪═══════╪══════════╪════════╡
│ 1    ┆ foo   ┆ 0.154163 ┆ A      │
│ 2    ┆ ham   ┆ 0.74005  ┆ A      │
│ 3    ┆ spam  ┆ 0.263315 ┆ B      │
│ null ┆ egg   ┆ 0.533739 ┆ C      │
│ 5    ┆ null  ┆ 0.014575 ┆ B      │
└──────┴───────┴──────────┴────────┘
```

NUMERICAL

Python

API `operators`

```
df_numerical = df.select(
    (pl.col("nrs") + 5).alias("nrs + 5"),
    (pl.col("nrs") - 5).alias("nrs - 5"),
    (pl.col("nrs") * pl.col("random")).alias("nrs * random"),
    (pl.col("nrs") / pl.col("random")).alias("nrs / random"),
)
print(df_numerical)
```

```
shape: (5, 4)
┌─────────┬─────────┬──────────────┬──────────────┐
│ nrs + 5 ┆ nrs - 5 ┆ nrs * random ┆ nrs / random │
│ ---     ┆ ---     ┆ ---          ┆ ---          │
│ i64     ┆ i64     ┆ f64          ┆ f64          │
╞═════════╪═════════╪══════════════╪══════════════╡
│ 6       ┆ -4      ┆ 0.154163     ┆ 6.486647     │
│ 7       ┆ -3      ┆ 1.480099     ┆ 2.702521     │
│ 8       ┆ -2      ┆ 0.789945     ┆ 11.393198    │
│ null    ┆ null    ┆ null         ┆ null         │
│ 10      ┆ 0       ┆ 0.072875     ┆ 343.054056   │
└─────────┴─────────┴──────────────┴──────────────┘
```

LOGICAL

Python

API `operators`

```python
df_logical = df.select(
    (pl.col("nrs") > 1).alias("nrs > 1"),
    (pl.col("random") <= 0.5).alias("random < .5"),
    (pl.col("nrs") != 1).alias("nrs != 1"),
    (pl.col("nrs") == 1).alias("nrs == 1"),
    ((pl.col("random") <= 0.5) & (pl.col("nrs") > 1)).alias("and_expr"),  # and
    ((pl.col("random") <= 0.5) | (pl.col("nrs") > 1)).alias("or_expr"),  # or
)
print(df_logical)
```

```
shape: (5, 6)
┌─────────┬─────────────┬──────────┬──────────┬──────────┬─────────┐
│ nrs > 1 ┆ random < .5 ┆ nrs != 1 ┆ nrs == 1 ┆ and_expr ┆ or_expr │
│ ---     ┆ ---         ┆ ---      ┆ ---      ┆ ---      ┆ ---     │
│ bool    ┆ bool        ┆ bool     ┆ bool     ┆ bool     ┆ bool    │
╞═════════╪═════════════╪══════════╪══════════╪══════════╪═════════╡
│ false   ┆ true        ┆ false    ┆ true     ┆ false    ┆ true    │
│ true    ┆ false       ┆ true     ┆ false    ┆ false    ┆ true    │
│ true    ┆ true        ┆ true     ┆ false    ┆ true     ┆ true    │
│ null    ┆ false       ┆ null     ┆ null     ┆ false    ┆ null    │
│ true    ┆ true        ┆ true     ┆ false    ┆ true     ┆ true    │
└─────────┴─────────────┴──────────┴──────────┴──────────┴─────────┘
```

4.4.2 Column Selections

Let's create a dataset to use in this section:

Python

API `DataFrame`

```
from datetime import date, datetime

df = pl.DataFrame(
    {
        "id": [9, 4, 2],
        "place": ["Mars", "Earth", "Saturn"],
        "date": pl.date_range(date(2022, 1, 1), date(2022, 1, 3), "1d", eager=True),
        "sales": [33.4, 2142134.1, 44.7],
        "has_people": [False, True, False],
        "logged_at": pl.date_range(
            datetime(2022, 12, 1), datetime(2022, 12, 1, 0, 0, 2), "1s", eager=True
        ),
    }
).with_row_count("rn")
print(df)
```

```
shape: (3, 7)
┌─────┬─────┬────────┬────────────┬───────────┬────────────┬─────────────────────┐
│ rn  ┆ id  ┆ place  ┆ date       ┆ sales     ┆ has_people ┆ logged_at           │
│ --- ┆ --- ┆ ---    ┆ ---        ┆ ---       ┆ ---        ┆ ---                 │
│ u32 ┆ i64 ┆ str    ┆ date       ┆ f64       ┆ bool       ┆ datetime[μs]        │
╞═════╪═════╪════════╪════════════╪═══════════╪════════════╪═════════════════════╡
│ 0   ┆ 9   ┆ Mars   ┆ 2022-01-01 ┆ 33.4      ┆ false      ┆ 2022-12-01 00:00:00 │
│ 1   ┆ 4   ┆ Earth  ┆ 2022-01-02 ┆ 2142134.1 ┆ true       ┆ 2022-12-01 00:00:01 │
│ 2   ┆ 2   ┆ Saturn ┆ 2022-01-03 ┆ 44.7      ┆ false      ┆ 2022-12-01 00:00:02 │
└─────┴─────┴────────┴────────────┴───────────┴────────────┴─────────────────────┘
```

Expression Expansion

As we've seen in the previous section, we can select specific columns using the `pl.col` method. It can also select multiple columns - both as a means of convenience, and to *expand* the expression.

This kind of convenience feature isn't just decorative or syntactic sugar. It allows for a very powerful application of DRY principles in your code: a single expression that specifies multiple columns expands into a list of expressions (depending on the DataFrame schema), resulting in being able to select multiple columns + run computation on them!

SELECT ALL, OR ALL BUT SOME

We can select all columns in the `DataFrame` object by providing the argument `*`:

Python

API `all`

```
out = df.select(pl.col("*"))

# Is equivalent to
out = df.select(pl.all())
print(out)
```

```
shape: (3, 7)
┌─────┬─────┬────────┬────────────┬───────────┬────────────┬─────────────────────┐
│ rn  ┆ id  ┆ place  ┆ date       ┆ sales     ┆ has_people ┆ logged_at           │
│ --- ┆ --- ┆ ---    ┆ ---        ┆ ---       ┆ ---        ┆ ---                 │
│ u32 ┆ i64 ┆ str    ┆ date       ┆ f64       ┆ bool       ┆ datetime[μs]        │
╞═════╪═════╪════════╪════════════╪═══════════╪════════════╪═════════════════════╡
│ 0   ┆ 9   ┆ Mars   ┆ 2022-01-01 ┆ 33.4      ┆ false      ┆ 2022-12-01 00:00:00 │
│ 1   ┆ 4   ┆ Earth  ┆ 2022-01-02 ┆ 2142134.1 ┆ true       ┆ 2022-12-01 00:00:01 │
│ 2   ┆ 2   ┆ Saturn ┆ 2022-01-03 ┆ 44.7      ┆ false      ┆ 2022-12-01 00:00:02 │
└─────┴─────┴────────┴────────────┴───────────┴────────────┴─────────────────────┘
```

Often, we don't just want to include all columns, but include all *while* excluding a few. This can be done easily as well:

Python

API `exclude`

```
out = df.select(pl.col("*").exclude("logged_at", "rn"))
print(out)
```

```
shape: (3, 5)
┌─────┬────────┬────────────┬───────────┬────────────┐
│ id  ┆ place  ┆ date       ┆ sales     ┆ has_people │
│ --- ┆ ---    ┆ ---        ┆ ---       ┆ ---        │
│ i64 ┆ str    ┆ date       ┆ f64       ┆ bool       │
╞═════╪════════╪════════════╪═══════════╪════════════╡
│ 9   ┆ Mars   ┆ 2022-01-01 ┆ 33.4      ┆ false      │
│ 4   ┆ Earth  ┆ 2022-01-02 ┆ 2142134.1 ┆ true       │
│ 2   ┆ Saturn ┆ 2022-01-03 ┆ 44.7      ┆ false      │
└─────┴────────┴────────────┴───────────┴────────────┘
```

BY MULTIPLE STRINGS

Specifying multiple strings allows expressions to *expand* to all matching columns:

Python

API `dt.to_string`

```
out = df.select(pl.col("date", "logged_at").dt.to_string("%Y-%h-%d"))
print(out)
```

```
shape: (3, 2)
┌─────────────┬─────────────┐
│ date        ┆ logged_at   │
│ ---         ┆ ---         │
│ str         ┆ str         │
╞═════════════╪═════════════╡
│ 2022-Jan-01 ┆ 2022-Dec-01 │
│ 2022-Jan-02 ┆ 2022-Dec-01 │
│ 2022-Jan-03 ┆ 2022-Dec-01 │
└─────────────┴─────────────┘
```

BY REGULAR EXPRESSIONS

Multiple column selection is possible by regular expressions also, by making sure to wrap the regex by `^` and `$` to let `pl.col` know that a regex selection is expected:

Python

```
out = df.select(pl.col("^.*(as|sa).*$"))
print(out)
```

```
shape: (3, 2)
┌───────────┬────────────┐
│ sales     ┆ has_people │
│ ---       ┆ ---        │
│ f64       ┆ bool       │
╞═══════════╪════════════╡
│ 33.4      ┆ false      │
│ 2142134.1 ┆ true       │
│ 44.7      ┆ false      │
└───────────┴────────────┘
```

BY DATA TYPE

`pl.col` can select multiple columns using Polars data types:

Python

API `n_unique`

```
out = df.select(pl.col(pl.Int64, pl.UInt32, pl.Boolean).n_unique())
print(out)
```

```
shape: (1, 3)
┌─────┬─────┬────────────┐
│ rn  ┆ id  ┆ has_people │
│ --- ┆ --- ┆ ---        │
│ u32 ┆ u32 ┆ u32        │
╞═════╪═════╪════════════╡
│ 3   ┆ 3   ┆ 2          │
└─────┴─────┴────────────┘
```

Using `selectors`

Polars also allows for the use of intuitive selections for columns based on their name, `dtype` or other properties; and this is built on top of existing functionality outlined in `col` used above. It is recommended to use them by importing and aliasing `polars.selectors` as `cs`.

BY `DTYPE`

To select just the integer and string columns, we can do:

Python

API `selectors`

```
import polars.selectors as cs

out = df.select(cs.integer(), cs.string())
print(out)
```

```
shape: (3, 3)
┌─────┬─────┬────────┐
│ rn  ┆ id  ┆ place  │
│ --- ┆ --- ┆ ---    │
│ u32 ┆ i64 ┆ str    │
╞═════╪═════╪════════╡
│ 0   ┆ 9   ┆ Mars   │
│ 1   ┆ 4   ┆ Earth  │
│ 2   ┆ 2   ┆ Saturn │
└─────┴─────┴────────┘
```

APPLYING SET OPERATIONS

These *selectors* also allow for set based selection operations. For instance, to select the **numeric** columns **except** the **first** column that indicates row numbers:

Python

API `cs.first` · API `cs.numeric`

```
out = df.select(cs.numeric() - cs.first())
print(out)
```

```
shape: (3, 2)
┌─────┬───────────┐
│ id  ┆ sales     │
│ --- ┆ ---       │
│ i64 ┆ f64       │
╞═════╪═══════════╡
│ 9   ┆ 33.4      │
│ 4   ┆ 2142134.1 │
│ 2   ┆ 44.7      │
└─────┴───────────┘
```

We can also select the row number by name **and** any **non**-numeric columns:

Python

API `cs.by_name` · API `cs.numeric`

```
out = df.select(cs.by_name("rn") | ~cs.numeric())
print(out)
```

```
shape: (3, 4)
┌────────┬────────────┬────────────┬─────────────────────┐
│ place  ┆ date       ┆ has_people ┆ logged_at           │
│ ---    ┆ ---        ┆ ---        ┆ ---                 │
│ str    ┆ date       ┆ bool       ┆ datetime[μs]        │
╞════════╪════════════╪════════════╪═════════════════════╡
│ Mars   ┆ 2022-01-01 ┆ false      ┆ 2022-12-01 00:00:00 │
│ Earth  ┆ 2022-01-02 ┆ true       ┆ 2022-12-01 00:00:01 │
│ Saturn ┆ 2022-01-03 ┆ false      ┆ 2022-12-01 00:00:02 │
└────────┴────────────┴────────────┴─────────────────────┘
```

BY PATTERNS AND SUBSTRINGS

Selectors can also be matched by substring and regex patterns:

Python

API `cs.contains` · API `cs.matches`

```
out = df.select(cs.contains("rn"), cs.matches(".*_.*"))
print(out)
```

```
shape: (3, 3)
┌─────┬────────────┬─────────────────────┐
│ rn  ┆ has_people ┆ logged_at           │
│ --- ┆ ---        ┆ ---                 │
│ u32 ┆ bool       ┆ datetime[μs]        │
╞═════╪════════════╪═════════════════════╡
│ 0   ┆ false      ┆ 2022-12-01 00:00:00 │
│ 1   ┆ true       ┆ 2022-12-01 00:00:01 │
│ 2   ┆ false      ┆ 2022-12-01 00:00:02 │
└─────┴────────────┴─────────────────────┘
```

CONVERTING TO EXPRESSIONS

What if we want to apply a specific operation on the selected columns (i.e. get back to representing them as **expressions** to operate upon)? We can simply convert them using `as_expr` and then proceed as normal:

Python

API `cs.temporal`

```
out = df.select(cs.temporal().as_expr().dt.to_string("%Y-%h-%d"))
print(out)
```

```
shape: (3, 2)
┌─────────────┬─────────────┐
│ date        ┆ logged_at   │
│ ---         ┆ ---         │
│ str         ┆ str         │
╞═════════════╪═════════════╡
│ 2022-Jan-01 ┆ 2022-Dec-01 │
│ 2022-Jan-02 ┆ 2022-Dec-01 │
│ 2022-Jan-03 ┆ 2022-Dec-01 │
└─────────────┴─────────────┘
```

DEBUGGING `SELECTORS`

Polars also provides two helpful utility functions to aid with using selectors: `is_selector` and `selector_column_names`:

Python

API `is_selector`

```
from polars.selectors import is_selector

out = cs.temporal()
print(is_selector(out))
```

```
True
```

To predetermine the column names that are selected, which is especially useful for a LazyFrame object:

Python

API `selector_column_names`

```
from polars.selectors import selector_column_names

out = cs.temporal().as_expr().dt.to_string("%Y-%h-%d")
print(selector_column_names(df, out))
```

```
('date', 'logged_at')
```

4.4.3 Functions

`Polars` expressions have a large number of built in functions. These allow you to create complex queries without the need for user defined functions. There are too many to go through here, but we will cover some of the more popular use cases. If you want to view all the functions go to the API Reference for your programming language.

In the examples below we will use the following `DataFrame`:

Python

API `DataFrame`

```
df = pl.DataFrame(
    {
        "nrs": [1, 2, 3, None, 5],
        "names": ["foo", "ham", "spam", "egg", "spam"],
        "random": np.random.rand(5),
        "groups": ["A", "A", "B", "C", "B"],
    }
)
print(df)
```

```
shape: (5, 4)
┌──────┬───────┬──────────┬────────┐
│ nrs  ┆ names ┆ random   ┆ groups │
│ ---  ┆ ---   ┆ ---      ┆ ---    │
│ i64  ┆ str   ┆ f64      ┆ str    │
╞══════╪═══════╪══════════╪════════╡
│ 1    ┆ foo   ┆ 0.154163 ┆ A      │
│ 2    ┆ ham   ┆ 0.74005  ┆ A      │
│ 3    ┆ spam  ┆ 0.263315 ┆ B      │
│ null ┆ egg   ┆ 0.533739 ┆ C      │
│ 5    ┆ spam  ┆ 0.014575 ┆ B      │
└──────┴───────┴──────────┴────────┘
```

Column Naming

By default if you perform an expression it will keep the same name as the original column. In the example below we perform an expression on the `nrs` column. Note that the output `DataFrame` still has the same name.

Python Python

```
df_samename = df.select(pl.col("nrs") + 5)
print(df_samename)

df_samename = df.select(pl.col("nrs") + 5)
print(df_samename)
```

```
shape: (5, 1)
┌──────┐
│ nrs  │
│ ---  │
│ i64  │
╞══════╡
│ 6    │
│ 7    │
│ 8    │
│ null │
│ 10   │
└──────┘
```

This might get problematic in the case you use the same column multiple times in your expression as the output columns will get duplicated. For example, the following query will fail.

Python

```
try:
    df_samename2 = df.select(pl.col("nrs") + 5, pl.col("nrs") - 5)
    print(df_samename2)
except Exception as e:
    print(e)
```

```
column with name 'nrs' has more than one occurrences
```

You can change the output name of an expression by using the `alias` function

Python

API `alias`

```
df_alias = df.select(
    (pl.col("nrs") + 5).alias("nrs + 5"),
    (pl.col("nrs") - 5).alias("nrs - 5"),
)
print(df_alias)
```

```
shape: (5, 2)
┌─────────┬─────────┐
│ nrs + 5 ┆ nrs - 5 │
│ ---     ┆ ---     │
│ i64     ┆ i64     │
╞═════════╪═════════╡
│ 6       ┆ -4      │
│ 7       ┆ -3      │
│ 8       ┆ -2      │
│ null    ┆ null    │
│ 10      ┆ 0       │
└─────────┴─────────┘
```

In case of multiple columns for example when using `all()` or `col(*)` you can apply a mapping function `map_alias` to change the original column name into something else. In case you want to add a suffix (`suffix()`) or prefix (`prefix()`) these are also built in.

Python

API `prefix` API `suffix` API `map_alias`

Count Unique Values

There are two ways to count unique values in `Polars` : an exact methodology and an approximation. The approximation uses the HyperLogLog++ algorithm to approximate the cardinality and is especially useful for very large datasets where an approximation is good enough.

Python

API `n_unique` · API `approx_unique`

```
df_alias = df.select(
    pl.col("names").n_unique().alias("unique"),
    pl.approx_unique("names").alias("unique_approx"),
)
print(df_alias)
```

```
shape: (1, 2)
┌────────┬───────────────┐
│ unique ┆ unique_approx │
│ ---    ┆ ---           │
│ u32    ┆ u32           │
╞════════╪═══════════════╡
│ 4      ┆ 4             │
└────────┴───────────────┘
```

Conditionals

`Polars` supports if-else like conditions in expressions with the `when`, `then`, `otherwise` syntax. The predicate is placed in the `when` clause and when this evaluates to `true` the `then` expression is applied otherwise the `otherwise` expression is applied (row-wise).

Python

API `when`

```python
df_conditional = df.select(
    pl.col("nrs"),
    pl.when(pl.col("nrs") > 2)
    .then(pl.lit(True))
    .otherwise(pl.lit(False))
    .alias("conditional"),
)
print(df_conditional)
```

```
shape: (5, 2)
┌──────┬─────────────┐
│ nrs  ┆ conditional │
│ ---  ┆ ---         │
│ i64  ┆ bool        │
╞══════╪═════════════╡
│ 1    ┆ false       │
│ 2    ┆ false       │
│ 3    ┆ true        │
│ null ┆ false       │
│ 5    ┆ true        │
└──────┴─────────────┘
```

4.4.4 Casting

Casting converts the underlying `DataType` of a column to a new one. Polars uses Arrow to manage the data in memory and relies on the compute kernels in the rust implementation to do the conversion. Casting is available with the `cast()` method.

The `cast` method includes a `strict` parameter that determines how Polars behaves when it encounters a value that can't be converted from the source `DataType` to the target `DataType`. By default, `strict=True`, which means that Polars will throw an error to notify the user of the failed conversion and provide details on the values that couldn't be cast. On the other hand, if `strict=False`, any values that can't be converted to the target `DataType` will be quietly converted to `null`.

Numerics

Let's take a look at the following `DataFrame` which contains both integers and floating point numbers.

Python

API `DataFrame`

```python
df = pl.DataFrame(
    {
        "integers": [1, 2, 3, 4, 5],
        "big_integers": [1, 10000002, 3, 10000004, 10000005],
        "floats": [4.0, 5.0, 6.0, 7.0, 8.0],
        "floats_with_decimal": [4.532, 5.5, 6.5, 7.5, 8.5],
    }
)

print(df)
```

```
shape: (5, 4)
┌──────────┬──────────────┬────────┬─────────────────────┐
│ integers ┆ big_integers ┆ floats ┆ floats_with_decimal │
│ ---      ┆ ---          ┆ ---    ┆ ---                 │
│ i64      ┆ i64          ┆ f64    ┆ f64                 │
╞══════════╪══════════════╪════════╪═════════════════════╡
│ 1        ┆ 1            ┆ 4.0    ┆ 4.532               │
│ 2        ┆ 10000002     ┆ 5.0    ┆ 5.5                 │
│ 3        ┆ 3            ┆ 6.0    ┆ 6.5                 │
│ 4        ┆ 10000004     ┆ 7.0    ┆ 7.5                 │
│ 5        ┆ 10000005     ┆ 8.0    ┆ 8.5                 │
└──────────┴──────────────┴────────┴─────────────────────┘
```

To perform casting operations between floats and integers, or vice versa, we can invoke the `cast()` function.

Python

API `cast`

```python
out = df.select(
    pl.col("integers").cast(pl.Float32).alias("integers_as_floats"),
    pl.col("floats").cast(pl.Int32).alias("floats_as_integers"),
    pl.col("floats_with_decimal")
    .cast(pl.Int32)
    .alias("floats_with_decimal_as_integers"),
)
print(out)
```

```
shape: (5, 3)
┌────────────────────┬────────────────────┬─────────────────────────────────┐
│ integers_as_floats ┆ floats_as_integers ┆ floats_with_decimal_as_integers │
│ ---                ┆ ---                ┆ ---                             │
│ f32                ┆ i32                ┆ i32                             │
╞════════════════════╪════════════════════╪═════════════════════════════════╡
│ 1.0                ┆ 4                  ┆ 4                               │
│ 2.0                ┆ 5                  ┆ 5                               │
│ 3.0                ┆ 6                  ┆ 6                               │
│ 4.0                ┆ 7                  ┆ 7                               │
│ 5.0                ┆ 8                  ┆ 8                               │
└────────────────────┴────────────────────┴─────────────────────────────────┘
```

Note that in the case of decimal values these are rounded downwards when casting to an integer.

Downcast

Reducing the memory footprint is also achievable by modifying the number of bits allocated to an element. As an illustration, the code below demonstrates how casting from `Int64` to `Int16` and from `Float64` to `Float32` can be used to lower memory usage.

Python

API `cast`

```
out = df.select(
    pl.col("integers").cast(pl.Int16).alias("integers_smallfootprint"),
    pl.col("floats").cast(pl.Float32).alias("floats_smallfootprint"),
)
print(out)
```

```
shape: (5, 2)
┌─────────────────────────┬───────────────────────┐
│ integers_smallfootprint ┆ floats_smallfootprint │
│ ---                     ┆ ---                   │
│ i16                     ┆ f32                   │
╞═════════════════════════╪═══════════════════════╡
│ 1                       ┆ 4.0                   │
│ 2                       ┆ 5.0                   │
│ 3                       ┆ 6.0                   │
│ 4                       ┆ 7.0                   │
│ 5                       ┆ 8.0                   │
└─────────────────────────┴───────────────────────┘
```

Overflow

When performing downcasting, it is crucial to ensure that the chosen number of bits (such as 64, 32, or 16) is sufficient to accommodate the largest and smallest numbers in the column. For example, using a 32-bit signed integer (`Int32`) allows handling integers within the range of -2147483648 to +2147483647, while using `Int8` covers integers between -128 to 127. Attempting to cast to a `DataType` that is too small will result in a `ComputeError` thrown by Polars, as the operation is not supported.

Python

API `cast`

```
try:
    out = df.select(pl.col("big_integers").cast(pl.Int8))
    print(out)
except Exception as e:
    print(e)
```

```
strict conversion from `i64` to `i8` failed for value(s) [10000002, 10000004, 10000005]; if you were trying to cast Utf8 to temporal dtypes, consider using
`strptime`
```

You can set the `strict` parameter to `False` , this converts values that are overflowing to null values.

Python

API `cast`

```
out = df.select(pl.col("big_integers").cast(pl.Int8, strict=False))
print(out)
```

```
shape: (5, 1)
┌──────────────┐
│ big_integers │
│ ---          │
│ i8           │
╞══════════════╡
│ 1            │
│ null         │
│ 3            │
│ null         │
│ null         │
└──────────────┘
```

Strings

Strings can be casted to numerical data types and vice versa:

Python

API `cast`

```
df = pl.DataFrame(
    {
        "integers": [1, 2, 3, 4, 5],
        "float": [4.0, 5.03, 6.0, 7.0, 8.0],
        "floats_as_string": ["4.0", "5.0", "6.0", "7.0", "8.0"],
    }
)

out = df.select(
    pl.col("integers").cast(pl.Utf8),
    pl.col("float").cast(pl.Utf8),
    pl.col("floats_as_string").cast(pl.Float64),
)
print(out)
```

```
shape: (5, 3)
┌──────────┬───────┬──────────────────┐
│ integers ┆ float ┆ floats_as_string │
│ ---      ┆ ---   ┆ ---              │
│ str      ┆ str   ┆ f64              │
╞══════════╪═══════╪══════════════════╡
│ 1        ┆ 4.0   ┆ 4.0              │
│ 2        ┆ 5.03  ┆ 5.0              │
│ 3        ┆ 6.0   ┆ 6.0              │
│ 4        ┆ 7.0   ┆ 7.0              │
│ 5        ┆ 8.0   ┆ 8.0              │
└──────────┴───────┴──────────────────┘
```

In case the column contains a non-numerical value, Polars will throw a `ComputeError` detailing the conversion error. Setting `strict=False` will convert the non float value to `null`.

Python

API `cast`

```
df = pl.DataFrame({"strings_not_float": ["4.0", "not_a_number", "6.0", "7.0", "8.0"]})
try:
    out = df.select(pl.col("strings_not_float").cast(pl.Float64))
    print(out)
except Exception as e:
    print(e)
```

```
strict conversion from `str` to `f64` failed for value(s) ["not_a_number"]; if you were trying to cast Utf8 to temporal dtypes, consider using `strptime`
```

Booleans

Booleans can be expressed as either 1 (`True`) or 0 (`False`). It's possible to perform casting operations between a numerical `DataType` and a boolean, and vice versa. However, keep in mind that casting from a string (`Utf8`) to a boolean is not permitted.

Python

API `cast`

```
df = pl.DataFrame(
    {
        "integers": [-1, 0, 2, 3, 4],
        "floats": [0.0, 1.0, 2.0, 3.0, 4.0],
        "bools": [True, False, True, False, True],
    }
)

out = df.select(pl.col("integers").cast(pl.Boolean), pl.col("floats").cast(pl.Boolean))
print(out)
```

```
shape: (5, 2)
┌──────────┬────────┐
│ integers ┆ floats │
│ ---      ┆ ---    │
│ bool     ┆ bool   │
╞══════════╪════════╡
│ true     ┆ false  │
│ false    ┆ true   │
│ true     ┆ true   │
│ true     ┆ true   │
│ true     ┆ true   │
└──────────┴────────┘
```

Dates

Temporal data types such as `Date` or `Datetime` are represented as the number of days (`Date`) and microseconds (`Datetime`) since epoch. Therefore, casting between the numerical types and the temporal data types is allowed.

Python

API `cast`

```python
from datetime import date, datetime

df = pl.DataFrame(
    {
        "date": pl.date_range(date(2022, 1, 1), date(2022, 1, 5), eager=True),
        "datetime": pl.date_range(
            datetime(2022, 1, 1), datetime(2022, 1, 5), eager=True
        ),
    }
)

out = df.select(pl.col("date").cast(pl.Int64), pl.col("datetime").cast(pl.Int64))
print(out)
```

```
shape: (5, 2)
┌───────┬──────────────────┐
│ date  ┆ datetime         │
│ ---   ┆ ---              │
│ i64   ┆ i64              │
╞═══════╪══════════════════╡
│ 18993 ┆ 1640995200000000 │
│ 18994 ┆ 1641081600000000 │
│ 18995 ┆ 1641168000000000 │
│ 18996 ┆ 1641254400000000 │
│ 18997 ┆ 1641340800000000 │
└───────┴──────────────────┘
```

To perform casting operations between strings and `Dates` / `Datetimes` , `strftime` and `strptime` are utilized. Polars adopts the chrono format syntax for when formatting. It's worth noting that `strptime` features additional options that support timezone functionality. Refer to the API documentation for further information.

Python

API `strftime` · API `strptime`

```python
df = pl.DataFrame(
    {
        "date": pl.date_range(date(2022, 1, 1), date(2022, 1, 5), eager=True),
        "string": [
            "2022-01-01",
            "2022-01-02",
            "2022-01-03",
            "2022-01-04",
            "2022-01-05",
        ],
    }
)

out = df.select(
    pl.col("date").dt.strftime("%Y-%m-%d"),
    pl.col("string").str.strptime(pl.Datetime, "%Y-%m-%d"),
)
print(out)
```

```
shape: (5, 2)
┌────────────┬─────────────────────┐
│ date       ┆ string              │
```

```
│ ---        ┆ ---                 │
│ str        ┆ datetime[μs]        │
╞════════════╪═════════════════════╡
│ 2022-01-01 ┆ 2022-01-01 00:00:00 │
│ 2022-01-02 ┆ 2022-01-02 00:00:00 │
│ 2022-01-03 ┆ 2022-01-03 00:00:00 │
│ 2022-01-04 ┆ 2022-01-04 00:00:00 │
│ 2022-01-05 ┆ 2022-01-05 00:00:00 │
└────────────┴─────────────────────┘
```

4.4.5 Strings

The following section discusses operations performed on `Utf8` strings, which are a frequently used `DataType` when working with `DataFrames`. However, processing strings can often be inefficient due to their unpredictable memory size, causing the CPU to access many random memory locations. To address this issue, Polars utilizes `Arrow` as its backend, which stores all strings in a contiguous block of memory. As a result, string traversal is cache-optimal and predictable for the CPU.

String processing functions are available in the `str` namespace.

Accessing the string namespace

The `str` namespace can be accessed through the `.str` attribute of a column with `Utf8` data type. In the following example, we create a column named `animal` and compute the length of each element in the column in terms of the number of bytes and the number of characters. If you are working with ASCII text, then the results of these two computations will be the same, and using `lengths` is recommended since it is faster.

Python

API `lengths` · **API** `n_chars`

```python
df = pl.DataFrame({"animal": ["Crab", "cat and dog", "rab$bit", None]})

out = df.select(
    pl.col("animal").str.lengths().alias("byte_count"),
    pl.col("animal").str.n_chars().alias("letter_count"),
)
print(out)
```

```
shape: (4, 2)
┌────────────┬──────────────┐
│ byte_count ┆ letter_count │
│ ---        ┆ ---          │
│ u32        ┆ u32          │
╞════════════╪══════════════╡
│ 4          ┆ 4            │
│ 11         ┆ 11           │
│ 7          ┆ 7            │
│ null       ┆ null         │
└────────────┴──────────────┘
```

String Parsing

`Polars` offers multiple methods for checking and parsing elements of a string. Firstly, we can use the `contains` method to check whether a given pattern exists within a substring. Subsequently, we can extract these patterns and replace them using other methods, which will be demonstrated in upcoming examples.

Check for existence of a pattern

To check for the presence of a pattern within a string, we can use the contains method. The `contains` method accepts either a regular substring or a regex pattern, depending on the value of the `literal` parameter. If the pattern we're searching for is a simple substring located either at the beginning or end of the string, we can alternatively use the `starts_with` and `ends_with` functions.

Python

API `str.contains` · **API** `starts_with` · **API** `ends_with`

```python
out = df.select(
    pl.col("animal"),
    pl.col("animal").str.contains("cat|bit").alias("regex"),
    pl.col("animal").str.contains("rab$", literal=True).alias("literal"),
    pl.col("animal").str.starts_with("rab").alias("starts_with"),
    pl.col("animal").str.ends_with("dog").alias("ends_with"),
)
print(out)
```

```
shape: (4, 5)
┌─────────────┬───────┬─────────┬─────────────┬───────────┐
│ animal      ┆ regex ┆ literal ┆ starts_with ┆ ends_with │
│ ---         ┆ ---   ┆ ---     ┆ ---         ┆ ---       │
│ str         ┆ bool  ┆ bool    ┆ bool        ┆ bool      │
╞═════════════╪═══════╪═════════╪═════════════╪═══════════╡
│ Crab        ┆ false ┆ false   ┆ false       ┆ false     │
```

```
│ cat and dog │ true  │ false    │ false       │ true      │
│ rab$bit     │ true  │ true     │ true        │ false     │
│ null        │ null  │ null     │ null        │ null      │
└─────────────┴───────┴──────────┴─────────────┴───────────┘
```

Extract a pattern

The `extract` method allows us to extract a pattern from a specified string. This method takes a regex pattern containing one or more capture groups, which are defined by parentheses `()` in the pattern. The group index indicates which capture group to output.

Python

API `extract`

```python
df = pl.DataFrame(
    {
        "a": [
            "http://vote.com/ballon_dor?candidate=messi&ref=polars",
            "http://vote.com/ballon_dor?candidat=jorginho&ref=polars",
            "http://vote.com/ballon_dor?candidate=ronaldo&ref=polars",
        ]
    }
)
out = df.select(
    pl.col("a").str.extract(r"candidate=(\w+)", group_index=1),
)
print(out)
```

```
shape: (3, 1)
┌─────────┐
│ a       │
│ ---     │
│ str     │
╞═════════╡
│ messi   │
│ null    │
│ ronaldo │
└─────────┘
```

To extract all occurrences of a pattern within a string, we can use the `extract_all` method. In the example below, we extract all numbers from a string using the regex pattern `(\d+)`, which matches one or more digits. The resulting output of the `extract_all` method is a list containing all instances of the matched pattern within the string.

Python

API `extract_all`

```python
df = pl.DataFrame({"foo": ["123 bla 45 asd", "xyz 678 910t"]})
out = df.select(
    pl.col("foo").str.extract_all(r"(\d+)").alias("extracted_nrs"),
)
print(out)
```

```
shape: (2, 1)
┌────────────────┐
│ extracted_nrs  │
│ ---            │
│ list[str]      │
╞════════════════╡
│ ["123", "45"]  │
│ ["678", "910"] │
└────────────────┘
```

Replace a pattern

We have discussed two methods for pattern matching and extraction thus far, and now we will explore how to replace a pattern within a string. Similar to `extract` and `extract_all`, Polars provides the `replace` and `replace_all` methods for this purpose. In the example below we replace one match of `abc` at the end of a word (`\b`) by `ABC` and we replace all occurrence of `a` with `-`.

Python

API `replace` · API `replace_all`

```
df = pl.DataFrame({"id": [1, 2], "text": ["123abc", "abc456"]})
out = df.with_columns(
    pl.col("text").str.replace(r"abc\b", "ABC"),
    pl.col("text").str.replace_all("a", "-", literal=True).alias("text_replace_all"),
)
print(out)
```

```
shape: (2, 3)
┌─────┬────────┬──────────────────┐
│ id  ┆ text   ┆ text_replace_all │
│ --- ┆ ---    ┆ ---              │
│ i64 ┆ str    ┆ str              │
╞═════╪════════╪══════════════════╡
│ 1   ┆ 123ABC ┆ 123-bc           │
│ 2   ┆ abc456 ┆ -bc456           │
└─────┴────────┴──────────────────┘
```

API Documentation

In addition to the examples covered above, Polars offers various other string manipulation methods for tasks such as formatting, stripping, splitting, and more. To explore these additional methods, you can go to the API documentation of your chosen programming language for Polars.

4.4.6 Aggregation

`Polars` implements a powerful syntax defined not only in its lazy API, but also in its eager API. Let's take a look at what that means.

We can start with the simple US congress `dataset`.

Python Rust

API `DataFrame` · API `Categorical`

```
url = "https://theunitedstates.io/congress-legislators/legislators-historical.csv"

dtypes = {
    "first_name": pl.Categorical,
    "gender": pl.Categorical,
    "type": pl.Categorical,
    "state": pl.Categorical,
    "party": pl.Categorical,
}

dataset = pl.read_csv(url, dtypes=dtypes).with_columns(
    pl.col("birthday").str.strptime(pl.Date, strict=False)
)
```

API `DataFrame` · API `Categorical` · Available on feature dtype-categorical

```
use std::io::Cursor;
use reqwest::blocking::Client;

let url = "https://theunitedstates.io/congress-legislators/legislators-historical.csv";

let mut schema = Schema::new();
schema.with_column("first_name".to_string(), DataType::Categorical(None));
schema.with_column("gender".to_string(), DataType::Categorical(None));
schema.with_column("type".to_string(), DataType::Categorical(None));
schema.with_column("state".to_string(), DataType::Categorical(None));
schema.with_column("party".to_string(), DataType::Categorical(None));
schema.with_column("birthday".to_string(), DataType::Date);

let data: Vec<u8> = Client::new().get(url).send()?.text()?.bytes().collect();

let dataset = CsvReader::new(Cursor::new(data))
    .has_header(true)
    .with_dtypes(Some(&schema))
    .with_parse_dates(true)
    .finish()?;

println!("{}", &dataset);
```

Basic aggregations

You can easily combine different aggregations by adding multiple expressions in a `list`. There is no upper bound on the number of aggregations you can do, and you can make any combination you want. In the snippet below we do the following aggregations:

Per GROUP `"first_name"` we

- count the number of rows in the group:
- short form: `pl.count("party")`
- full form: `pl.col("party").count()`
- aggregate the gender values groups:
- full form: `pl.col("gender")`
- get the first value of column `"last_name"` in the group:
- short form: `pl.first("last_name")` (not available in Rust)
- full form: `pl.col("last_name").first()`

Besides the aggregation, we immediately sort the result and limit to the top `5` so that we have a nice summary overview.

Python Rust

API `groupby`

```
q = (
    dataset.lazy()
    .groupby("first_name")
    .agg(
        pl.count(),
        pl.col("gender"),
        pl.first("last_name"),
    )
    .sort("count", descending=True)
    .limit(5)
)

df = q.collect()
print(df)
```

API `groupby`

```
let df = dataset
    .clone()
    .lazy()
    .groupby(["first_name"])
    .agg([count(), col("gender").list(), col("last_name").first()])
    .sort(
        "count",
        SortOptions {
            descending: true,
            nulls_last: true,
        },
    )
    .limit(5)
    .collect()?;

println!("{}", df);
```

```
shape: (5, 4)
┌────────────┬───────┬───────────────────┬───────────┐
│ first_name ┆ count ┆ gender            ┆ last_name │
│ ---        ┆ ---   ┆ ---               ┆ ---       │
│ cat        ┆ u32   ┆ list[cat]         ┆ str       │
╞════════════╪═══════╪═══════════════════╪═══════════╡
│ John       ┆ 1256  ┆ ["M", "M", … "M"] ┆ Walker    │
│ William    ┆ 1022  ┆ ["M", "M", … "M"] ┆ Few       │
│ James      ┆ 714   ┆ ["M", "M", … "M"] ┆ Armstrong │
│ Thomas     ┆ 454   ┆ ["M", "M", … "M"] ┆ Tucker    │
│ Charles    ┆ 439   ┆ ["M", "M", … "M"] ┆ Carroll   │
└────────────┴───────┴───────────────────┴───────────┘
```

Conditionals

It's that easy! Let's turn it up a notch. Let's say we want to know how many delegates of a "state" are "Pro" or "Anti" administration. We could directly query that in the aggregation without the need of a `lambda` or grooming the `DataFrame`.

Python Rust

API `groupby`

```python
q = (
    dataset.lazy()
    .groupby("state")
    .agg(
        (pl.col("party") == "Anti-Administration").sum().alias("anti"),
        (pl.col("party") == "Pro-Administration").sum().alias("pro"),
    )
    .sort("pro", descending=True)
    .limit(5)
)

df = q.collect()
print(df)
```

API `groupby`

```rust
let df = dataset
    .clone()
    .lazy()
    .groupby(["state"])
    .agg([
        (col("party").eq(lit("Anti-Administration")))
            .sum()
            .alias("anti"),
        (col("party").eq(lit("Pro-Administration")))
            .sum()
            .alias("pro"),
    ])
    .sort(
        "pro",
        SortOptions {
            descending: true,
            nulls_last: false,
        },
    )
    .limit(5)
    .collect()?;

println!("{}", df);
```

```
shape: (5, 3)
┌───────┬──────┬──────┐
│ state ┆ anti ┆ pro  │
│ ---   ┆ ---  ┆ ---  │
│ cat   ┆ u32  ┆ u32  │
╞═══════╪══════╪══════╡
│ PI    ┆ null ┆ null │
│ OL    ┆ null ┆ null │
│ NJ    ┆ 0    ┆ 3    │
│ CT    ┆ 0    ┆ 3    │
│ NC    ┆ 1    ┆ 2    │
└───────┴──────┴──────┘
```

Similarly, this could also be done with a nested GROUPBY, but that doesn't help show off some of these nice features. 😉

Python Rust

API `groupby`

```
q = (
    dataset.lazy()
    .groupby("state", "party")
    .agg(pl.count("party").alias("count"))
    .filter(
        (pl.col("party") == "Anti-Administration")
        | (pl.col("party") == "Pro-Administration")
    )
    .sort("count", descending=True)
    .limit(5)
)

df = q.collect()
print(df)
```

API `groupby`

```
let df = dataset
    .clone()
    .lazy()
    .groupby(["state", "party"])
    .agg([col("party").count().alias("count")])
    .filter(
        col("party")
            .eq(lit("Anti-Administration"))
            .or(col("party").eq(lit("Pro-Administration"))),
    )
    .sort(
        "count",
        SortOptions {
            descending: true,
            nulls_last: true,
        },
    )
    .limit(5)
    .collect()?;

println!("{}", df);
```

```
shape: (5, 3)
┌───────┬─────────────────────┬───────┐
│ state ┆ party               ┆ count │
│ ---   ┆ ---                 ┆ ---   │
│ cat   ┆ cat                 ┆ u32   │
╞═══════╪═════════════════════╪═══════╡
│ VA    ┆ Anti-Administration ┆ 3     │
│ NJ    ┆ Pro-Administration  ┆ 3     │
│ CT    ┆ Pro-Administration  ┆ 3     │
│ NC    ┆ Pro-Administration  ┆ 2     │
│ SC    ┆ Pro-Administration  ┆ 1     │
└───────┴─────────────────────┴───────┘
```

Filtering

We can also filter the groups. Let's say we want to compute a mean per group, but we don't want to include all values from that group, and we also don't want to filter the rows from the `DataFrame` (because we need those rows for another aggregation).

In the example below we show how this can be done.

Note

Note that we can make `Python` functions for clarity. These functions don't cost us anything. That is because we only create `Polars` expressions, we don't apply a custom function over a `Series` during runtime of the query. Of course, you can make functions that return expressions in Rust, too.

Python Rust

API `groupby`

```python
def compute_age() -> pl.Expr:
    return date(2021, 1, 1).year - pl.col("birthday").dt.year()


def avg_birthday(gender: str) -> pl.Expr:
    return (
        compute_age()
        .filter(pl.col("gender") == gender)
        .mean()
        .alias(f"avg {gender} birthday")
    )


q = (
    dataset.lazy()
    .groupby("state")
    .agg(
        avg_birthday("M"),
        avg_birthday("F"),
        (pl.col("gender") == "M").sum().alias("# male"),
        (pl.col("gender") == "F").sum().alias("# female"),
    )
    .limit(5)
)

df = q.collect()
print(df)
```

API `groupby`

```rust
fn compute_age() -> Expr {
    lit(2022) - col("birthday").dt().year()
}

fn avg_birthday(gender: &str) -> Expr {
    compute_age()
        .filter(col("gender").eq(lit(gender)))
        .mean()
        .alias(&format!("avg {} birthday", gender))
}

let df = dataset
    .clone()
    .lazy()
    .groupby(["state"])
    .agg([
        avg_birthday("M"),
        avg_birthday("F"),
        (col("gender").eq(lit("M"))).sum().alias("# male"),
        (col("gender").eq(lit("F"))).sum().alias("# female"),
    ])
    .limit(5)
    .collect()?;

println!("{}", df);
```

```
shape: (5, 5)
┌───────┬────────────────┬────────────────┬────────┬──────────┐
│ state ┆ avg M birthday ┆ avg F birthday ┆ # male ┆ # female │
│ ---   ┆ ---            ┆ ---            ┆ ---    ┆ ---      │
│ cat   ┆ f64            ┆ f64            ┆ u32    ┆ u32      │
╞═══════╪════════════════╪════════════════╪════════╪══════════╡
│ DE    ┆ 181.593407     ┆ null           ┆ 97     ┆ 0        │
│ ND    ┆ 136.833333     ┆ 82.5           ┆ 42     ┆ 2        │
│ TN    ┆ 175.949091     ┆ 109.6          ┆ 297    ┆ 5        │
│ NM    ┆ 139.0          ┆ 67.666667      ┆ 52     ┆ 6        │
│ KS    ┆ 148.397059     ┆ 85.714286      ┆ 136    ┆ 7        │
└───────┴────────────────┴────────────────┴────────┴──────────┘
```

Sorting

It's common to see a `DataFrame` being sorted for the sole purpose of managing the ordering during a GROUPBY operation. Let's say that we want to get the names of the oldest and youngest politicians per state. We could SORT and GROUPBY.

Python Rust

API `groupby`

```python
def get_person() -> pl.Expr:
    return pl.col("first_name") + pl.lit(" ") + pl.col("last_name")


q = (
    dataset.lazy()
    .sort("birthday", descending=True)
    .groupby("state")
    .agg(
        get_person().first().alias("youngest"),
        get_person().last().alias("oldest"),
    )
    .limit(5)
)

df = q.collect()
print(df)
```

API `groupby`

```rust
fn get_person() -> Expr {
    col("first_name") + lit(" ") + col("last_name")
}

let df = dataset
    .clone()
    .lazy()
    .sort(
        "birthday",
        SortOptions {
            descending: true,
            nulls_last: true,
        },
    )
    .groupby(["state"])
    .agg([
        get_person().first().alias("youngest"),
        get_person().last().alias("oldest"),
    ])
    .limit(5)
    .collect()?;

println!("{}", df);
```

```
shape: (5, 3)
┌───────┬─────────────────────┬────────────────┐
│ state ┆ youngest            ┆ oldest         │
│ ---   ┆ ---                 ┆ ---            │
│ cat   ┆ str                 ┆ str            │
╞═══════╪═════════════════════╪════════════════╡
│ KS    ┆ Steven Watkins      ┆ James Lane     │
│ NM    ┆ Xochitl Torres Small┆ José Gallegos  │
│ MO    ┆ Vicky Hartzler      ┆ Spencer Pettis │
│ DE    ┆ John Carney         ┆ Samuel White   │
│ TN    ┆ Stephen Fincher     ┆ William Cocke  │
└───────┴─────────────────────┴────────────────┘
```

However, **if** we also want to sort the names alphabetically, this breaks. Luckily we can sort in a `groupby` context separate from the `DataFrame`.

Python Rust

API `groupby`

```python
def get_person() -> pl.Expr:
    return pl.col("first_name") + pl.lit(" ") + pl.col("last_name")


q = (
    dataset.lazy()
    .sort("birthday", descending=True)
    .groupby("state")
    .agg(
        get_person().first().alias("youngest"),
        get_person().last().alias("oldest"),
        get_person().sort().first().alias("alphabetical_first"),
    )
    .limit(5)
)

df = q.collect()
print(df)
```

API `groupby`

```rust
let df = dataset
    .clone()
    .lazy()
    .sort(
        "birthday",
        SortOptions {
            descending: true,
            nulls_last: true,
        },
    )
    .groupby(["state"])
    .agg([
        get_person().first().alias("youngest"),
        get_person().last().alias("oldest"),
        get_person().sort(false).first().alias("alphabetical_first"),
    ])
    .limit(5)
    .collect()?;

println!("{}", df);
```

```
shape: (5, 4)
┌───────┬─────────────────────┬────────────────┬────────────────────┐
│ state ┆ youngest            ┆ oldest         ┆ alphabetical_first │
│ ---   ┆ ---                 ┆ ---            ┆ ---                │
│ cat   ┆ str                 ┆ str            ┆ str                │
╞═══════╪═════════════════════╪════════════════╪════════════════════╡
│ ND    ┆ Rick Berg           ┆ Lyman Casey    ┆ Arthur Link        │
│ NM    ┆ Xochitl Torres Small┆ José Gallegos  ┆ Albert Fall        │
│ KS    ┆ Steven Watkins      ┆ James Lane     ┆ Abel Wilder        │
│ TN    ┆ Stephen Fincher     ┆ William Cocke  ┆ Aaron Brown        │
│ DE    ┆ John Carney         ┆ Samuel White   ┆ Albert Polk        │
└───────┴─────────────────────┴────────────────┴────────────────────┘
```

We can even sort by another column in the `groupby` context. If we want to know if the alphabetically sorted name is male or female we could add: `pl.col("gender").sort_by("first_name").first().alias("gender")`

Python Rust

API `groupby`

```
def get_person() -> pl.Expr:
    return pl.col("first_name") + pl.lit(" ") + pl.col("last_name")


q = (
    dataset.lazy()
    .sort("birthday", descending=True)
    .groupby("state")
    .agg(
        get_person().first().alias("youngest"),
        get_person().last().alias("oldest"),
        get_person().sort().first().alias("alphabetical_first"),
        pl.col("gender").sort_by("first_name").first().alias("gender"),
    )
    .sort("state")
    .limit(5)
)

df = q.collect()
print(df)
```

API `groupby`

```
let df = dataset
    .clone()
    .lazy()
    .sort(
        "birthday",
        SortOptions {
            descending: true,
            nulls_last: true,
        },
    )
    .groupby(["state"])
    .agg([
        get_person().first().alias("youngest"),
        get_person().last().alias("oldest"),
        get_person().sort(false).first().alias("alphabetical_first"),
        col("gender")
            .sort_by(["first_name"], [false])
            .first()
            .alias("gender"),
    ])
    .sort("state", SortOptions::default())
    .limit(5)
    .collect()?;

println!("{}", df);
```

```
shape: (5, 5)
┌───────┬──────────────────┬────────────────────┬────────────────────┬────────┐
│ state ┆ youngest         ┆ oldest             ┆ alphabetical_first ┆ gender │
│ ---   ┆ ---              ┆ ---                ┆ ---                ┆ ---    │
│ cat   ┆ str              ┆ str                ┆ str                ┆ cat    │
╞═══════╪══════════════════╪════════════════════╪════════════════════╪════════╡
│ PA    ┆ Conor Lamb       ┆ Thomas Fitzsimons  ┆ Aaron Kreider      ┆ M      │
│ OH    ┆ Anthony Gonzalez ┆ John Smith         ┆ Aaron Harlan       ┆ M      │
│ VA    ┆ Scott Taylor     ┆ William Grayson    ┆ A. McEachin        ┆ M      │
│ MA    ┆ Joseph Kennedy   ┆ William Widgery    ┆ Aaron Hobart       ┆ M      │
│ FL    ┆ Patrick Murphy   ┆ Charles Downing    ┆ Abijah Gilbert     ┆ M      │
└───────┴──────────────────┴────────────────────┴────────────────────┴────────┘
```

DO NOT KILL PARALLELIZATION

Python Users Only

The following section is specific to `Python`, and doesn't apply to `Rust`. Within `Rust`, blocks and closures (lambdas) can, and will, be executed concurrently.

We have all heard that `Python` is slow, and does "not scale." Besides the overhead of running "slow" bytecode, `Python` has to remain within the constraints of the Global Interpreter Lock (GIL). This means that if you were to use a `lambda` or a custom `Python` function to apply during a parallelized phase, `Polars` speed is capped running `Python` code preventing any multiple threads from executing the function.

This all feels terribly limiting, especially because we often need those `lambda` functions in a `.groupby()` step, for example. This approach is still supported by `Polars`, but keeping in mind bytecode **and** the GIL costs have to be paid. It is recommended to try to solve your queries using the expression syntax before moving to `lambdas`. If you want to learn more about using `lambdas`, go to the user defined functions section.

CONCLUSION

In the examples above we've seen that we can do a lot by combining expressions. By doing so we delay the use of custom `Python` functions that slow down the queries (by the slow nature of Python AND the GIL).

If we are missing a type expression let us know by opening a feature request!

4.4.7 Missing data

This page sets out how missing data is represented in `Polars` and how missing data can be filled.

`null` and `NaN` values

Each column in a `DataFrame` (or equivalently a `Series`) is an Arrow array or a collection of Arrow arrays based on the Apache Arrow format. Missing data is represented in Arrow and `Polars` with a `null` value. This `null` missing value applies for all data types including numerical values.

`Polars` also allows `NotaNumber` or `NaN` values for float columns. These `NaN` values are considered to be a type of floating point data rather than missing data. We discuss `NaN` values separately below.

You can manually define a missing value with the python `None` value:

Python

API `DataFrame`

```
df = pl.DataFrame(
    {
        "value": [1, None],
    },
)
print(df)
```

```
shape: (2, 1)
┌───────┐
│ value │
│ ---   │
│ i64   │
╞═══════╡
│ 1     │
│ null  │
└───────┘
```

> **Info**
>
> In `Pandas` the value for missing data depends on the dtype of the column. In `Polars` missing data is always represented as a `null` value.

Missing data metadata

Each Arrow array used by `Polars` stores two kinds of metadata related to missing data. This metadata allows `Polars` to quickly show how many missing values there are and which values are missing.

The first piece of metadata is the `null_count` - this is the number of rows with `null` values in the column:

Python

API `null_count`

```
null_count_df = df.null_count()
print(null_count_df)
```

```
shape: (1, 1)
┌───────┐
│ value │
│ ---   │
│ u32   │
╞═══════╡
│ 1     │
└───────┘
```

The `null_count` method can be called on a `DataFrame`, a column from a `DataFrame` or a `Series`. The `null_count` method is a cheap operation as `null_count` is already calculated for the underlying Arrow array.

The second piece of metadata is an array called a *validity bitmap* that indicates whether each data value is valid or missing. The validity bitmap is memory efficient as it is bit encoded - each value is either a 0 or a 1. This bit encoding means the memory overhead per array is only (array length / 8) bytes. The validity bitmap is used by the `is_null` method in `Polars`.

You can return a `Series` based on the validity bitmap for a column in a `DataFrame` or a `Series` with the `is_null` method:

Python

API `is_null`

```
is_null_series = df.select(
    pl.col("value").is_null(),
)
print(is_null_series)
```

```
shape: (2, 1)
┌───────┐
│ value │
│ ---   │
│ bool  │
╞═══════╡
│ false │
│ true  │
└───────┘
```

The `is_null` method is a cheap operation that does not require scanning the full column for `null` values. This is because the validity bitmap already exists and can be returned as a Boolean array.

Filling missing data

Missing data in a `Series` can be filled with the `fill_null` method. You have to specify how you want the `fill_null` method to fill the missing data. The main ways to do this are filling with:

- a literal such as 0 or "0"
- a strategy such as filling forwards
- an expression such as replacing with values from another column
- interpolation

We illustrate each way to fill nulls by defining a simple `DataFrame` with a missing value in `col2`:

Python

API `DataFrame`

```
df = pl.DataFrame(
    {
        "col1": [1, 2, 3],
        "col2": [1, None, 3],
    },
)
print(df)
```

```
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 1    ┆ 1    │
│ 2    ┆ null │
│ 3    ┆ 3    │
└──────┴──────┘
```

FILL WITH SPECIFIED LITERAL VALUE

We can fill the missing data with a specified literal value with `pl.lit`:

Python

API `fill_null`

```
fill_literal_df = (
    df.with_columns(
        pl.col("col2").fill_null(
            pl.lit(2),
        ),
    ),
)
print(fill_literal_df)
```

```
(shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 1    ┆ 1    │
│ 2    ┆ 2    │
│ 3    ┆ 3    │
└──────┴──────┘,)
```

FILL WITH A STRATEGY

We can fill the missing data with a strategy such as filling forward:

Python

API `fill_null`

```
fill_forward_df = df.with_columns(
    pl.col("col2").fill_null(strategy="forward"),
)
print(fill_forward_df)
```

```
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 1    ┆ 1    │
│ 2    ┆ 1    │
│ 3    ┆ 3    │
└──────┴──────┘
```

You can find other fill strategies in the API docs.

FILL WITH AN EXPRESSION

For more flexibility we can fill the missing data with an expression. For example, to fill nulls with the median value from that column:

Python

API `fill_null`

```
fill_median_df = df.with_columns(
    pl.col("col2").fill_null(pl.median("col2")),
)
print(fill_median_df)
```

```
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ f64  │
╞══════╪══════╡
│ 1    ┆ 1.0  │
```

```
│ 2    ┆ 2.0 │
│ 3    ┆ 3.0 │
└──────┴─────┘
```

In this case the column is cast from integer to float because the median is a float statistic.

FILL WITH INTERPOLATION

In addition, we can fill nulls with interpolation (without using the `fill_null` function):

Python

API `interpolate`

```python
fill_interpolation_df = df.with_columns(
    pl.col("col2").interpolate(),
)
print(fill_interpolation_df)
```

```
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 1    ┆ 1    │
│ 2    ┆ 2    │
│ 3    ┆ 3    │
└──────┴──────┘
```

`NotaNumber` or `NaN` values

Missing data in a `Series` has a `null` value. However, you can use `NotaNumber` or `NaN` values in columns with float datatypes. These `NaN` values can be created from Numpy's `np.nan` or the native python `float('nan')` :

Python

API `DataFrame`

```python
nan_df = pl.DataFrame(
    {
        "value": [1.0, np.NaN, float("nan"), 3.0],
    },
)
print(nan_df)
```

```
shape: (4, 1)
┌───────┐
│ value │
│ ---   │
│ f64   │
╞═══════╡
│ 1.0   │
│ NaN   │
│ NaN   │
│ 3.0   │
└───────┘
```

Info

In `Pandas` by default a `NaN` value in an integer column causes the column to be cast to float. This does not happen in `Polars` - instead an exception is raised.

`NaN` values are considered to be a type of floating point data and are **not considered to be missing data** in `Polars` . This means:

- `NaN` values are **not** counted with the `null_count` method
- `NaN` values are filled when you use `fill_nan` method but are **not** filled with the `fill_null` method

`Polars` has `is_nan` and `fill_nan` methods which work in a similar way to the `is_null` and `fill_null` methods. The underlying Arrow arrays do not have a pre-computed validity bitmask for `NaN` values so this has to be computed for the `is_nan` method.

One further difference between `null` and `NaN` values is that taking the `mean` of a column with `null` values excludes the `null` values from the calculation but with `NaN` values taking the mean results in a `NaN`. This behaviour can be avoided by replacing the `NaN` values with `null` values;

Python

API `fill_nan`

```
mean_nan_df = nan_df.with_columns(
    pl.col("value").fill_nan(None).alias("value"),
).mean()
print(mean_nan_df)
```

```
shape: (1, 1)
┌───────┐
│ value │
│ ---   │
│ f64   │
╞═══════╡
│ 2.0   │
└───────┘
```

4.4.8 Window functions

Window functions are expressions with superpowers. They allow you to perform aggregations on groups in the `select` context. Let's get a feel for what that means. First we create a dataset. The dataset loaded in the snippet below contains information about pokemon:

Python Rust

API `read_csv`

```
import polars as pl

# then let's load some csv data with information about pokemon
df = pl.read_csv(
    "https://gist.githubusercontent.com/ritchie46/cac6b337ea52281aa23c049250a4ff03/raw/89a957ff3919d90e6ef2d34235e6bf22304f3366/pokemon.csv"
)
print(df.head())
```

API `CsvReader` · Available on feature csv

```
use polars::prelude::*;
use reqwest::blocking::Client;

let data: Vec<u8> = Client::new()
    .get("https://gist.githubusercontent.com/ritchie46/cac6b337ea52281aa23c049250a4ff03/raw/89a957ff3919d90e6ef2d34235e6bf22304f3366/pokemon.csv")
    .send()?
    .text()?
    .bytes()
    .collect();

let df = CsvReader::new(std::io::Cursor::new(data))
    .has_header(true)
    .finish()?;

println!("{}", df);
```

```
shape: (5, 13)
┌─────┬───────────────────────┬────────┬────────┬───┬─────────┬───────┬────────────┬───────────┐
│ #   ┆ Name                  ┆ Type 1 ┆ Type 2 ┆ … ┆ Sp. Def ┆ Speed ┆ Generation ┆ Legendary │
│ --- ┆ ---                   ┆ ---    ┆ ---    ┆   ┆ ---     ┆ ---   ┆ ---        ┆ ---       │
│ i64 ┆ str                   ┆ str    ┆ str    ┆   ┆ i64     ┆ i64   ┆ i64        ┆ bool      │
╞═════╪═══════════════════════╪════════╪════════╪═══╪═════════╪═══════╪════════════╪═══════════╡
│ 1   ┆ Bulbasaur             ┆ Grass  ┆ Poison ┆ … ┆ 65      ┆ 45    ┆ 1          ┆ false     │
│ 2   ┆ Ivysaur               ┆ Grass  ┆ Poison ┆ … ┆ 80      ┆ 60    ┆ 1          ┆ false     │
│ 3   ┆ Venusaur              ┆ Grass  ┆ Poison ┆ … ┆ 100     ┆ 80    ┆ 1          ┆ false     │
│ 3   ┆ VenusaurMega Venusaur ┆ Grass  ┆ Poison ┆ … ┆ 120     ┆ 80    ┆ 1          ┆ false     │
│ 4   ┆ Charmander            ┆ Fire   ┆ null   ┆ … ┆ 50      ┆ 65    ┆ 1          ┆ false     │
└─────┴───────────────────────┴────────┴────────┴───┴─────────┴───────┴────────────┴───────────┘
```

Groupby Aggregations in selection

Below we show how to use window functions to group over different columns and perform an aggregation on them. Doing so allows us to use multiple groupby operations in parallel, using a single query. The results of the aggregation are projected back to the original rows. Therefore, a window function will almost always lead to a `DataFrame` with the same size as the original.

We will discuss later the cases where a window function can change the numbers of rows in a `DataFrame`.

Note how we call `.over("Type 1")` and `.over(["Type 1", "Type 2"])`. Using window functions we can aggregate over different groups in a single `select` call! Note that, in Rust, the type of the argument to `over()` must be a collection, so even when you're only using one column, you must provided it in an array.

The best part is, this won't cost you anything. The computed groups are cached and shared between different `window` expressions.

Python Rust

API `over`

```python
out = df.select(
    "Type 1",
    "Type 2",
    pl.col("Attack").mean().over("Type 1").alias("avg_attack_by_type"),
    pl.col("Defense")
    .mean()
    .over(["Type 1", "Type 2"])
    .alias("avg_defense_by_type_combination"),
    pl.col("Attack").mean().alias("avg_attack"),
)
print(out)
```

API `over`

```rust
let out = df
    .clone()
    .lazy()
    .select([
        col("Type 1"),
        col("Type 2"),
        col("Attack")
            .mean()
            .over(["Type 1"])
            .alias("avg_attack_by_type"),
        col("Defense")
            .mean()
            .over(["Type 1", "Type 2"])
            .alias("avg_defense_by_type_combination"),
        col("Attack").mean().alias("avg_attack"),
    ])
    .collect()?;

println!("{}", out);
```

```
shape: (163, 5)
┌─────────┬────────┬────────────────────┬─────────────────────────────────┬────────────┐
│ Type 1  ┆ Type 2 ┆ avg_attack_by_type ┆ avg_defense_by_type_combination ┆ avg_attack │
│ ---     ┆ ---    ┆ ---                ┆ ---                             ┆ ---        │
│ str     ┆ str    ┆ f64                ┆ f64                             ┆ f64        │
╞═════════╪════════╪════════════════════╪═════════════════════════════════╪════════════╡
│ Grass   ┆ Poison ┆ 72.923077          ┆ 67.8                            ┆ 75.349693  │
│ Grass   ┆ Poison ┆ 72.923077          ┆ 67.8                            ┆ 75.349693  │
│ Grass   ┆ Poison ┆ 72.923077          ┆ 67.8                            ┆ 75.349693  │
│ Grass   ┆ Poison ┆ 72.923077          ┆ 67.8                            ┆ 75.349693  │
│ …       ┆ …      ┆ …                  ┆ …                               ┆ …          │
│ Dragon  ┆ null   ┆ 94.0               ┆ 55.0                            ┆ 75.349693  │
│ Dragon  ┆ null   ┆ 94.0               ┆ 55.0                            ┆ 75.349693  │
│ Dragon  ┆ Flying ┆ 94.0               ┆ 95.0                            ┆ 75.349693  │
│ Psychic ┆ null   ┆ 53.875             ┆ 51.428571                       ┆ 75.349693  │
└─────────┴────────┴────────────────────┴─────────────────────────────────┴────────────┘
```

Operations per group

Window functions can do more than aggregation. They can also be viewed as an operation within a group. If, for instance, you want to `sort` the values within a `group`, you can write `col("value").sort().over("group")` and voilà! We sorted by group!

Let's filter out some rows to make this more clear.

Python Rust

API `filter`

```
filtered = df.filter(pl.col("Type 2") == "Psychic").select(
    "Name",
    "Type 1",
    "Speed",
)
print(filtered)
```

API `filter`

```
let filtered = df
    .clone()
    .lazy()
    .filter(col("Type 2").eq(lit("Psychic")))
    .select([col("Name"), col("Type 1"), col("Speed")])
    .collect()?;

println!("{}", filtered);
```

```
shape: (7, 3)
┌─────────────────────┬────────┬───────┐
│ Name                ┆ Type 1 ┆ Speed │
│ ---                 ┆ ---    ┆ ---   │
│ str                 ┆ str    ┆ i64   │
╞═════════════════════╪════════╪═══════╡
│ Slowpoke            ┆ Water  ┆ 15    │
│ Slowbro             ┆ Water  ┆ 30    │
│ SlowbroMega Slowbro ┆ Water  ┆ 30    │
│ Exeggcute           ┆ Grass  ┆ 40    │
│ Exeggutor           ┆ Grass  ┆ 55    │
│ Starmie             ┆ Water  ┆ 115   │
│ Jynx                ┆ Ice    ┆ 95    │
└─────────────────────┴────────┴───────┘
```

Observe that the group `Water` of column `Type 1` is not contiguous. There are two rows of `Grass` in between. Also note that each pokemon within a group are sorted by `Speed` in `ascending` order. Unfortunately, for this example we want them sorted in `descending` speed order. Luckily with window functions this is easy to accomplish.

Python Rust

API `over`

```
out = filtered.with_columns(
    pl.col(["Name", "Speed"]).sort_by("Speed", descending=True).over("Type 1"),
)
print(out)
```

API `over`

```
let out = filtered
    .lazy()
    .with_columns([cols(["Name", "Speed"]).sort_by(["Speed"],[true]).over(["Type 1"])])
    .collect()?;
println!("{}", out);
```

```
shape: (7, 3)
┌─────────────────────┬────────┬───────┐
│ Name                ┆ Type 1 ┆ Speed │
│ ---                 ┆ ---    ┆ ---   │
│ str                 ┆ str    ┆ i64   │
╞═════════════════════╪════════╪═══════╡
│ Starmie             ┆ Water  ┆ 115   │
│ Slowbro             ┆ Water  ┆ 30    │
│ SlowbroMega Slowbro ┆ Water  ┆ 30    │
│ Exeggutor           ┆ Grass  ┆ 55    │
│ Exeggcute           ┆ Grass  ┆ 40    │
│ Slowpoke            ┆ Water  ┆ 15    │
│ Jynx                ┆ Ice    ┆ 95    │
└─────────────────────┴────────┴───────┘
```

`Polars` keeps track of each group's location and maps the expressions to the proper row locations. This will also work over different groups in a single `select`.

The power of window expressions is that you often don't need a `groupby -> explode` combination, but you can put the logic in a single expression. It also makes the API cleaner. If properly used a:

- `groupby` -> marks that groups are aggregated and we expect a `DataFrame` of size `n_groups`
- `over` -> marks that we want to compute something within a group, and doesn't modify the original size of the `DataFrame` except in specific cases

Map the expression result to the DataFrame rows

In cases where the expression results in multiple values per group, the Window function has 3 strategies for linking the values back to the `DataFrame` rows:

- `mapping_strategy = 'group_to_rows'` -> each value is assigned back to one row. The number of values returned should match the number of rows.
- `mapping_strategy = 'join'` -> the values are imploded in a list, and the list is repeated on all rows. This can be memory intensive.
- `mapping_strategy = 'explode'` -> the values are exploded to new rows. This operation changes the number of rows.

Window expression rules

The evaluations of window expressions are as follows (assuming we apply it to a `pl.Int32` column):

Python Rust

API `over`

```python
# aggregate and broadcast within a group
# output type: -> Int32
pl.sum("foo").over("groups")

# sum within a group and multiply with group elements
# output type: -> Int32
(pl.col("x").sum() * pl.col("y")).over("groups")

# sum within a group and multiply with group elements
# and aggregate the group to a list
# output type: -> List(Int32)
(pl.col("x").sum() * pl.col("y")).over("groups", mapping_strategy="join")

# sum within a group and multiply with group elements
# and aggregate the group to a list
# then explode the list to multiple rows

# This is the fastest method to do things over groups when the groups are sorted
(pl.col("x").sum() * pl.col("y")).over("groups", mapping_strategy="explode")
```

API `over`

```rust
// aggregate and broadcast within a group
// output type: -> i32
sum("foo").over([col("groups")])
// sum within a group and multiply with group elements
// output type: -> i32
(col("x").sum() * col("y"))
    .over([col("groups")])
    .alias("x1")
// sum within a group and multiply with group elements
// and aggregate the group to a list
// output type: -> ChunkedArray<i32>
(col("x").sum() * col("y"))
    .list()
    .over([col("groups")])
    .alias("x2")
// note that it will require an explicit `list()` call
// sum within a group and multiply with group elements
// and aggregate the group to a list
// the flatten call explodes that list

// This is the fastest method to do things over groups when the groups are sorted
(col("x").sum() * col("y"))
    .list()
    .over([col("groups")])
    .flatten()
    .alias("x3");
```

More examples

For more exercise, below are some window functions for us to compute:

- sort all pokemon by type
- select the first `3` pokemon per type as `"Type 1"`
- sort the pokemon within a type by speed in descending order and select the first `3` as `"fastest/group"`
- sort the pokemon within a type by attack in descending order and select the first `3` as `"strongest/group"`
- sort the pokemon within a type by name and select the first `3` as `"sorted_by_alphabet"`

Python Rust

API `over` · API `implode`

```python
out = df.sort("Type 1").select(
    pl.col("Type 1").head(3).over("Type 1", mapping_strategy="explode"),
    pl.col("Name")
    .sort_by(pl.col("Speed"), descending=True)
    .head(3)
    .over("Type 1", mapping_strategy="explode")
    .alias("fastest/group"),
    pl.col("Name")
    .sort_by(pl.col("Attack"), descending=True)
    .head(3)
    .over("Type 1", mapping_strategy="explode")
    .alias("strongest/group"),
    pl.col("Name")
    .sort()
    .head(3)
    .over("Type 1", mapping_strategy="explode")
    .alias("sorted_by_alphabet"),
)
print(out)
```

API `over` · API `implode`

```rust
let out = df
    .clone()
    .lazy()
    .select([
        col("Type 1")
            .head(Some(3))
            .list()
            .over(["Type 1"])
            .flatten(),
        col("Name")
            .sort_by(["Speed"], [true])
            .head(Some(3))
            .list()
            .over(["Type 1"])
            .flatten()
            .alias("fastest/group"),
        col("Name")
            .sort_by(["Attack"], [true])
            .head(Some(3))
            .list()
            .over(["Type 1"])
            .flatten()
            .alias("strongest/group"),
        col("Name")
            .sort(false)
            .head(Some(3))
            .list()
            .over(["Type 1"])
            .flatten()
            .alias("sorted_by_alphabet"),
    ])
    .collect()?;
println!("{:?}", out);
```

```
shape: (43, 4)
┌────────┬─────────────────────┬─────────────────────┬─────────────────────┐
│ Type 1 ┆ fastest/group       ┆ strongest/group     ┆ sorted_by_alphabet  │
│ ---    ┆ ---                 ┆ ---                 ┆ ---                 │
│ str    ┆ str                 ┆ str                 ┆ str                 │
╞════════╪═════════════════════╪═════════════════════╪═════════════════════╡
│ Bug    ┆ BeedrillMega Beedrill ┆ PinsirMega Pinsir   ┆ Beedrill            │
│ Bug    ┆ Scyther             ┆ BeedrillMega Beedrill ┆ BeedrillMega Beedrill │
│ Bug    ┆ PinsirMega Pinsir   ┆ Pinsir              ┆ Butterfree          │
│ Dragon ┆ Dragonite           ┆ Dragonite           ┆ Dragonair           │
│ …      ┆ …                   ┆ …                   ┆ …                   │
│ Rock   ┆ Kabutops            ┆ Kabutops            ┆ Geodude             │
```

```
│ Water   ┆ Starmie                ┆ GyaradosMega Gyarados ┆ Blastoise              │
│ Water   ┆ Tentacruel             ┆ Kingler               ┆ BlastoiseMega Blastoise │
│ Water   ┆ Poliwag                ┆ Gyarados              ┆ Cloyster               │
└─────────┴────────────────────────┴───────────────────────┴────────────────────────┘
```

4.4.9 Folds

`Polars` provides expressions/methods for horizontal aggregations like `sum`, `min`, `mean`, etc. by setting the argument `axis=1`. However, when you need a more complex aggregation the default methods `Polars` supplies may not be sufficient. That's when `folds` come in handy.

The `fold` expression operates on columns for maximum speed. It utilizes the data layout very efficiently and often has vectorized execution.

MANUAL SUM

Let's start with an example by implementing the `sum` operation ourselves, with a `fold`.

Python Rust

API `fold`

```python
df = pl.DataFrame(
    {
        "a": [1, 2, 3],
        "b": [10, 20, 30],
    }
)

out = df.select(
    pl.fold(acc=pl.lit(0), function=lambda acc, x: acc + x, exprs=pl.all()).alias(
        "sum"
    ),
)
print(out)
```

API `fold_exprs`

```rust
let df = df!(
    "a" => &[1, 2, 3],
    "b" => &[10, 20, 30],
)?;

let out = df
    .lazy()
    .select([fold_exprs(lit(0), |acc, x| Ok(Some(acc + x)), [col("*")]).alias("sum")])
    .collect()?;
println!("{}", out);
```

```
shape: (3, 1)
┌─────┐
│ sum │
│ --- │
│ i64 │
╞═════╡
│ 11  │
│ 22  │
│ 33  │
└─────┘
```

The snippet above recursively applies the function `f(acc, x) -> acc` to an accumulator `acc` and a new column `x`. The function operates on columns individually and can take advantage of cache efficiency and vectorization.

CONDITIONAL

In the case where you'd want to apply a condition/predicate on all columns in a `DataFrame` a `fold` operation can be a very concise way to express this.

Python Rust

API `fold`

```
df = pl.DataFrame(
    {
        "a": [1, 2, 3],
        "b": [0, 1, 2],
    }
)

out = df.filter(
    pl.fold(
        acc=pl.lit(True),
        function=lambda acc, x: acc & x,
        exprs=pl.col("*") > 1,
    )
)
print(out)
```

API `fold_exprs`

```
let df = df!(
    "a" => &[1, 2, 3],
    "b" => &[0, 1, 2],
)?;

let out = df
    .lazy()
    .filter(fold_exprs(
        lit(true),
        |acc, x| Some(acc.bitand(&x)),
        [col("*").gt(1)],
    ))
    .collect()?;
println!("{}", out);
```

```
shape: (1, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 3   ┆ 2   │
└─────┴─────┘
```

In the snippet we filter all rows where **each** column value is `> 1`.

FOLDS AND STRING DATA

Folds could be used to concatenate string data. However, due to the materialization of intermediate columns, this operation will have squared complexity.

Therefore, we recommend using the `concat_str` expression for this.

Python Rust

API `concat_str`

```python
df = pl.DataFrame(
    {
        "a": ["a", "b", "c"],
        "b": [1, 2, 3],
    }
)

out = df.select(pl.concat_str(["a", "b"]))
print(out)
```

API `concat_str` · Available on feature concat_str

```rust
let df = df!(
    "a" => &["a", "b", "c"],
    "b" => &[1, 2, 3],
)?;

let out = df
    .lazy()
    .select([concat_str([col("a"), col("b")], "")])
    .collect()?;
println!("{:?}", out);
```

```
shape: (3, 1)
┌─────┐
│ a   │
│ --- │
│ str │
╞═════╡
│ a1  │
│ b2  │
│ c3  │
└─────┘
```

4.4.10 Lists and Arrays

`Polars` has first-class support for `List` columns: that is, columns where each row is a list of homogenous elements, of varying lengths. `Polars` also has an `Array` datatype, which is analogous to `numpy`'s `ndarray` objects, where the length is identical across rows.

Note: this is different from Python's `list` object, where the elements can be of any type. Polars can store these within columns, but as a generic `Object` datatype that doesn't have the special list manipulation features that we're about to discuss.

Powerful `List` manipulation

Let's say we had the following data from different weather stations across a state. When the weather station is unable to get a result, an error code is recorded instead of the actual temperature at that time.

Python

API `DataFrame`

```
weather = pl.DataFrame(
    {
        "station": ["Station " + str(x) for x in range(1, 6)],
        "temperatures": [
            "20 5 5 E1 7 13 19 9 6 20",
            "18 8 16 11 23 E2 8 E2 E2 E2 90 70 40",
            "19 24 E9 16 6 12 10 22",
            "E2 E0 15 7 8 10 E1 24 17 13 6",
            "14 8 E0 16 22 24 E1",
        ],
    }
)
print(weather)
```

```
shape: (5, 2)
┌───────────┬─────────────────────────────────┐
│ station   ┆ temperatures                    │
│ ---       ┆ ---                             │
│ str       ┆ str                             │
╞═══════════╪═════════════════════════════════╡
│ Station 1 ┆ 20 5 5 E1 7 13 19 9 6 20        │
│ Station 2 ┆ 18 8 16 11 23 E2 8 E2 E2 E2 90 7… │
│ Station 3 ┆ 19 24 E9 16 6 12 10 22          │
│ Station 4 ┆ E2 E0 15 7 8 10 E1 24 17 13 6   │
│ Station 5 ┆ 14 8 E0 16 22 24 E1             │
└───────────┴─────────────────────────────────┘
```

CREATING A `LIST` COLUMN

For the `weather` `DataFrame` created above, it's very likely we need to run some analysis on the temperatures that are captured by each station. To make this happen, we need to first be able to get individual temperature measurements. This is done by:

Python

API `str.split`

```
out = weather.with_columns(pl.col("temperatures").str.split(" "))
print(out)
```

```
shape: (5, 2)
┌───────────┬──────────────────────┐
│ station   ┆ temperatures         │
│ ---       ┆ ---                  │
│ str       ┆ list[str]            │
╞═══════════╪══════════════════════╡
│ Station 1 ┆ ["20", "5", … "20"]  │
│ Station 2 ┆ ["18", "8", … "40"]  │
│ Station 3 ┆ ["19", "24", … "22"] │
│ Station 4 ┆ ["E2", "E0", … "6"]  │
│ Station 5 ┆ ["14", "8", … "E1"]  │
└───────────┴──────────────────────┘
```

One way we could go post this would be to convert each temperature measurement into its own row:

Python

API `DataFrame.explode`

```
out = weather.with_columns(pl.col("temperatures").str.split(" ")).explode(
    "temperatures"
)
print(out)
```

```
shape: (49, 2)
┌───────────┬──────────────┐
│ station   ┆ temperatures │
│ ---       ┆ ---          │
│ str       ┆ str          │
╞═══════════╪══════════════╡
│ Station 1 ┆ 20           │
│ Station 1 ┆ 5            │
│ Station 1 ┆ 5            │
│ Station 1 ┆ E1           │
│ …         ┆ …            │
│ Station 5 ┆ 16           │
│ Station 5 ┆ 22           │
│ Station 5 ┆ 24           │
│ Station 5 ┆ E1           │
└───────────┴──────────────┘
```

However, in Polars, we often do not need to do this to operate on the `List` elements.

OPERATING ON `LIST` COLUMNS

Polars provides several standard operations on `List` columns. If we want the first three measurements, we can do a `head(3)`. The last three can be obtained via a `tail(3)`, or alternately, via `slice` (negative indexing is supported). We can also identify the number of observations via `lengths`. Let's see them in action:

Python

API `Expr.List`

```
out = weather.with_columns(pl.col("temperatures").str.split(" ")).with_columns(
    pl.col("temperatures").list.head(3).alias("top3"),
    pl.col("temperatures").list.slice(-3, 3).alias("bottom_3"),
    pl.col("temperatures").list.lengths().alias("obs"),
)
print(out)
```

```
shape: (5, 5)
┌───────────┬──────────────────────┬────────────────────┬────────────────────┬─────┐
│ station   ┆ temperatures         ┆ top3               ┆ bottom_3           ┆ obs │
│ ---       ┆ ---                  ┆ ---                ┆ ---                ┆ --- │
│ str       ┆ list[str]            ┆ list[str]          ┆ list[str]          ┆ u32 │
╞═══════════╪══════════════════════╪════════════════════╪════════════════════╪═════╡
│ Station 1 ┆ ["20", "5", … "20"]  ┆ ["20", "5", "5"]   ┆ ["9", "6", "20"]   ┆ 10  │
│ Station 2 ┆ ["18", "8", … "40"]  ┆ ["18", "8", "16"]  ┆ ["90", "70", "40"] ┆ 13  │
│ Station 3 ┆ ["19", "24", … "22"] ┆ ["19", "24", "E9"] ┆ ["12", "10", "22"] ┆ 8   │
│ Station 4 ┆ ["E2", "E0", … "6"]  ┆ ["E2", "E0", "15"] ┆ ["17", "13", "6"]  ┆ 11  │
│ Station 5 ┆ ["14", "8", … "E1"]  ┆ ["14", "8", "E0"]  ┆ ["22", "24", "E1"] ┆ 7   │
└───────────┴──────────────────────┴────────────────────┴────────────────────┴─────┘
```

`arr` then, `list` now

If you find references to the `arr` API on Stackoverflow or other sources, just replace `arr` with `list`, this was the old accessor for the `List` datatype. `arr` now refers to the newly introduced `Array` datatype (see below).

ELEMENT-WISE COMPUTATION WITHIN `LIST`S

If we need to identify the stations that are giving the most number of errors from the starting `DataFrame`, we need to:

1. Parse the string input as a `List` of string values (already done).
2. Identify those strings that can be converted to numbers.
3. Identify the number of non-numeric values (i.e. `null` values) in the list, by row.
4. Rename this output as `errors` so that we can easily identify the stations.

The third step requires a casting (or alternately, a regex pattern search) operation to be perform on each element of the list. We can do this using by applying the operation on each element by first referencing them in the `pl.element()` context, and then calling a suitable Polars expression on them. Let's see how:

Python

API `Expr.List` · API `element`

```
out = weather.with_columns(
    pl.col("temperatures")
    .str.split(" ")
    .list.eval(pl.element().cast(pl.Int64, strict=False).is_null())
    .list.sum()
    .alias("errors")
)
print(out)
```

```
shape: (5, 3)
┌───────────┬────────────────────────────────────┬────────┐
│ station   ┆ temperatures                       ┆ errors │
│ ---       ┆ ---                                ┆ ---    │
│ str       ┆ str                                ┆ u32    │
╞═══════════╪════════════════════════════════════╪════════╡
│ Station 1 ┆ 20 5 5 E1 7 13 19 9 6 20           ┆ 1      │
│ Station 2 ┆ 18 8 16 11 23 E2 8 E2 E2 E2 90 7…  ┆ 4      │
│ Station 3 ┆ 19 24 E9 16 6 12 10 22             ┆ 1      │
│ Station 4 ┆ E2 E0 15 7 8 10 E1 24 17 13 6      ┆ 3      │
│ Station 5 ┆ 14 8 E0 16 22 24 E1                ┆ 2      │
└───────────┴────────────────────────────────────┴────────┘
```

What if we chose the regex route (i.e. recognizing the presence of *any* alphabetical character?)

Python

API `str.contains`

```
out = weather.with_columns(
    pl.col("temperatures")
    .str.split(" ")
    .list.eval(pl.element().str.contains("(?i)[a-z]"))
    .list.sum()
    .alias("errors")
)
print(out)
```

```
shape: (5, 3)
┌───────────┬────────────────────────────────────┬────────┐
│ station   ┆ temperatures                       ┆ errors │
│ ---       ┆ ---                                ┆ ---    │
│ str       ┆ str                                ┆ u32    │
╞═══════════╪════════════════════════════════════╪════════╡
│ Station 1 ┆ 20 5 5 E1 7 13 19 9 6 20           ┆ 1      │
│ Station 2 ┆ 18 8 16 11 23 E2 8 E2 E2 E2 90 7…  ┆ 4      │
│ Station 3 ┆ 19 24 E9 16 6 12 10 22             ┆ 1      │
│ Station 4 ┆ E2 E0 15 7 8 10 E1 24 17 13 6      ┆ 3      │
│ Station 5 ┆ 14 8 E0 16 22 24 E1                ┆ 2      │
└───────────┴────────────────────────────────────┴────────┘
```

If you're unfamiliar with the `(?i)`, it's a good time to look at the documentation for the `str.contains` function in Polars! The rust regex crate provides a lot of additional regex flags that might come in handy.

Row-wise computations

This context is ideal for computing in row orientation.

We can apply **any** Polars operations on the elements of the list with the `list.eval` (`list().eval` in Rust) expression! These expressions run entirely on Polars' query engine and can run in parallel, so will be well optimized. Let's say we have another set of weather data across three days, for different stations:

Python

API `DataFrame`

```python
weather_by_day = pl.DataFrame(
    {
        "station": ["Station " + str(x) for x in range(1, 11)],
        "day_1": [17, 11, 8, 22, 9, 21, 20, 8, 8, 17],
        "day_2": [15, 11, 10, 8, 7, 14, 18, 21, 15, 13],
        "day_3": [16, 15, 24, 24, 8, 23, 19, 23, 16, 10],
    }
)
print(weather_by_day)
```

```
shape: (10, 4)
┌────────────┬───────┬───────┬───────┐
│ station    ┆ day_1 ┆ day_2 ┆ day_3 │
│ ---        ┆ ---   ┆ ---   ┆ ---   │
│ str        ┆ i64   ┆ i64   ┆ i64   │
╞════════════╪═══════╪═══════╪═══════╡
│ Station 1  ┆ 17    ┆ 15    ┆ 16    │
│ Station 2  ┆ 11    ┆ 11    ┆ 15    │
│ Station 3  ┆ 8     ┆ 10    ┆ 24    │
│ Station 4  ┆ 22    ┆ 8     ┆ 24    │
│ …          ┆ …     ┆ …     ┆ …     │
│ Station 7  ┆ 20    ┆ 18    ┆ 19    │
│ Station 8  ┆ 8     ┆ 21    ┆ 23    │
│ Station 9  ┆ 8     ┆ 15    ┆ 16    │
│ Station 10 ┆ 17    ┆ 13    ┆ 10    │
└────────────┴───────┴───────┴───────┘
```

Let's do something interesting, where we calculate the percentage rank of the temperatures by day, measured across stations. Pandas allows you to compute the percentages of the `rank` values. `Polars` doesn't provide a special function to do this directly, but because expressions are so versatile we can create our own percentage rank expression for highest temperature. Let's try that!

Python

API `list.eval`

```python
rank_pct = (pl.element().rank(descending=True) / pl.col("*").count()).round(2)

out = weather_by_day.with_columns(
    # create the list of homogeneous data
    pl.concat_list(pl.all().exclude("station")).alias("all_temps")
).select(
    # select all columns except the intermediate list
    pl.all().exclude("all_temps"),
    # compute the rank by calling `list.eval`
    pl.col("all_temps").list.eval(rank_pct, parallel=True).alias("temps_rank"),
)

print(out)
```

```
shape: (10, 5)
┌────────────┬───────┬───────┬───────┬────────────────────┐
│ station    ┆ day_1 ┆ day_2 ┆ day_3 ┆ temps_rank         │
│ ---        ┆ ---   ┆ ---   ┆ ---   ┆ ---                │
│ str        ┆ i64   ┆ i64   ┆ i64   ┆ list[f64]          │
╞════════════╪═══════╪═══════╪═══════╪════════════════════╡
│ Station 1  ┆ 17    ┆ 15    ┆ 16    ┆ [0.33, 1.0, 0.67]  │
│ Station 2  ┆ 11    ┆ 11    ┆ 15    ┆ [0.83, 0.83, 0.33] │
│ Station 3  ┆ 8     ┆ 10    ┆ 24    ┆ [1.0, 0.67, 0.33]  │
│ Station 4  ┆ 22    ┆ 8     ┆ 24    ┆ [0.67, 1.0, 0.33]  │
│ …          ┆ …     ┆ …     ┆ …     ┆ …                  │
│ Station 7  ┆ 20    ┆ 18    ┆ 19    ┆ [0.33, 1.0, 0.67]  │
│ Station 8  ┆ 8     ┆ 21    ┆ 23    ┆ [1.0, 0.67, 0.33]  │
│ Station 9  ┆ 8     ┆ 15    ┆ 16    ┆ [1.0, 0.67, 0.33]  │
│ Station 10 ┆ 17    ┆ 13    ┆ 10    ┆ [0.33, 0.67, 1.0]  │
└────────────┴───────┴───────┴───────┴────────────────────┘
```

Polars `Array` s

`Array` s are a new data type that was recently introduced, and are still pretty nascent in features that it offers. The major difference between a `List` and an `Array` is that the latter is limited to having the same number of elements per row, while a `List` can have a variable number of elements. Both still require that each element's data type is the same.

We can define `Array` columns in this manner:

Python

API `Array`

```
array_df = pl.DataFrame(
    [
        pl.Series("Array_1", [[1, 3], [2, 5]]),
        pl.Series("Array_2", [[1, 7, 3], [8, 1, 0]]),
    ],
    schema={"Array_1": pl.Array(2, pl.Int64), "Array_2": pl.Array(3, pl.Int64)},
)
print(array_df)
```

```
shape: (2, 2)
┌───────────────┬───────────────┐
│ Array_1       ┆ Array_2       │
│ ---           ┆ ---           │
│ array[i64, 2] ┆ array[i64, 3] │
╞═══════════════╪═══════════════╡
│ [1, 3]        ┆ [1, 7, 3]     │
│ [2, 5]        ┆ [8, 1, 0]     │
└───────────────┴───────────────┘
```

Basic operations are available on it:

Python

API `arr`

```
out = array_df.select(
    pl.col("Array_1").arr.min().suffix("_min"),
    pl.col("Array_2").arr.sum().suffix("_sum"),
)
print(out)
```

```
shape: (2, 2)
┌─────────────┬─────────────┐
│ Array_1_min ┆ Array_2_sum │
│ ---         ┆ ---         │
│ i64         ┆ i64         │
╞═════════════╪═════════════╡
│ 1           ┆ 11          │
│ 2           ┆ 9           │
└─────────────┴─────────────┘
```

Polars `Array` s are still being actively developed, so this section will likely change in the future.

4.4.11 User Defined functions

You should be convinced by now that polar expressions are so powerful and flexible that there is much less need for custom python functions than in other libraries.

Still, you need to have the power to be able to pass an expression's state to a third party library or apply your black box function over data in polars.

For this we provide the following expressions:

- `map`
- `apply`

To `map` or to `apply`.

These functions have an important distinction in how they operate and consequently what data they will pass to the user.

A `map` passes the `Series` backed by the `expression` as is.

`map` follows the same rules in both the `select` and the `groupby` context, this will mean that the `Series` represents a column in a `DataFrame`. Note that in the `groupby` context, that column is not yet aggregated!

Use cases for `map` are for instance passing the `Series` in an expression to a third party library. Below we show how we could use `map` to pass an expression column to a neural network model.

Python Rust

API `map`

```python
df.with_columns([
    pl.col("features").map(lambda s: MyNeuralNetwork.forward(s.to_numpy())).alias("activations")
])
```

```rust
df.with_columns([
    col("features").map(|s| Ok(my_nn.forward(s))).alias("activations")
])
```

Use cases for `map` in the `groupby` context are slim. They are only used for performance reasons, but can quite easily lead to incorrect results. Let me explain why.

Python Rust

API `map`

```
df = pl.DataFrame(
    {
        "keys": ["a", "a", "b"],
        "values": [10, 7, 1],
    }
)

out = df.groupby("keys", maintain_order=True).agg(
    pl.col("values").map(lambda s: s.shift()).alias("shift_map"),
    pl.col("values").shift().alias("shift_expression"),
)
print(df)
```

API `map`

```
let df = df!(
    "keys" => &["a", "a", "b"],
    "values" => &[10, 7, 1],
)?;

let out = df
    .lazy()
    .groupby(["keys"])
    .agg([
        col("values")
            .map(|s| Ok(s.shift(1)), GetOutput::default())
            .alias("shift_map"),
        col("values").shift(1).alias("shift_expression"),
    ])
    .collect()?;

println!("{}", out);
```

```
shape: (3, 2)
┌──────┬────────┐
│ keys ┆ values │
│ ---  ┆ ---    │
│ str  ┆ i64    │
╞══════╪════════╡
│ a    ┆ 10     │
│ a    ┆ 7      │
│ b    ┆ 1      │
└──────┴────────┘
```

In the snippet above we groupby the `"keys"` column. That means we have the following groups:

```
"a" -> [10, 7]
"b" -> [1]
```

If we would then apply a `shift` operation to the right, we'd expect:

```
"a" -> [null, 10]
"b" -> [null]
```

Now, let's print and see what we've got.

```
print(out)
```

```
shape: (2, 3)
┌──────┬────────────┬──────────────────┐
│ keys ┆ shift_map  ┆ shift_expression │
│ ---  ┆ ---        ┆ ---              │
│ str  ┆ list[i64]  ┆ list[i64]        │
╞══════╪════════════╪══════════════════╡
│ a    ┆ [null, 10] ┆ [null, 10]       │
│ b    ┆ [7]        ┆ [null]           │
└──────┴────────────┴──────────────────┘
```

Ouch.. we clearly get the wrong results here. Group `"b"` even got a value from group `"a"` 😵.

This went horribly wrong, because the `map` applies the function before we aggregate! So that means the whole column `[10, 7, 1]` got shifted to `[null, 10, 7]` and was then aggregated.

So my advice is to never use `map` in the `groupby` context unless you know you need it and know what you are doing.

To `apply`

Luckily we can fix previous example with `apply`. `apply` works on the smallest logical elements for that operation.

That is:

- `select context` -> single elements
- `groupby context` -> single groups

So with `apply` we should be able to fix our example:

Python Rust

API `apply`

```
out = df.groupby("keys", maintain_order=True).agg(
    pl.col("values").apply(lambda s: s.shift()).alias("shift_map"),
    pl.col("values").shift().alias("shift_expression"),
)
print(out)
```

API `apply`

```
let out = df
    .clone()
    .lazy()
    .groupby([col("keys")])
    .agg([
        col("values")
            .apply(|s| Ok(s.shift(1)), GetOutput::default())
            .alias("shift_map"),
        col("values").shift(1).alias("shift_expression"),
    ])
    .collect()?;
println!("{}", out);
```

```
shape: (2, 3)
┌──────┬────────────┬──────────────────┐
│ keys ┆ shift_map  ┆ shift_expression │
│ ---  ┆ ---        ┆ ---              │
│ str  ┆ list[i64]  ┆ list[i64]        │
╞══════╪════════════╪══════════════════╡
│ a    ┆ [null, 10] ┆ [null, 10]       │
│ b    ┆ [null]     ┆ [null]           │
└──────┴────────────┴──────────────────┘
```

And observe, a valid result! 🎉

`apply` in the `select` context

In the `select` context, the `apply` expression passes elements of the column to the python function.

Note that you are now running python, this will be slow.

Let's go through some examples to see what to expect. We will continue with the `DataFrame` we defined at the start of this section and show an example with the `apply` function and a counter example where we use the expression API to achieve the same goals.

ADDING A COUNTER

In this example we create a global `counter` and then add the integer `1` to the global state at every element processed. Every iteration the result of the increment will be added to the element value.

Note, this example isn't provided in Rust. The reason is that the global `counter` value would lead to data races when this apply is evaluated in parallel. It would be possible to wrap it in a `Mutex` to protect the variable, but that would be obscuring the point of the example. This is a case where the Python Global Interpreter Lock's performance tradeoff provides some safety guarantees.

Python Rust

API `apply`

```python
counter = 0


def add_counter(val: int) -> int:
    global counter
    counter += 1
    return counter + val


out = df.select(
    pl.col("values").apply(add_counter).alias("solution_apply"),
    (pl.col("values") + pl.arange(1, pl.count() + 1)).alias("solution_expr"),
)
print(out)
```

API `apply`

```

```

```
shape: (3, 2)
┌────────────────┬───────────────┐
│ solution_apply ┆ solution_expr │
│ ---            ┆ ---           │
│ i64            ┆ i64           │
╞════════════════╪═══════════════╡
│ 11             ┆ 11            │
│ 9              ┆ 9             │
│ 4              ┆ 4             │
└────────────────┴───────────────┘
```

COMBINING MULTIPLE COLUMN VALUES

If we want to have access to values of different columns in a single `apply` function call, we can create `struct` data type. This data type collects those columns as fields in the `struct`. So if we'd create a struct from the columns `"keys"` and `"values"`, we would get the following struct elements:

```
[
    {"keys": "a", "values": 10},
    {"keys": "a", "values": 7},
    {"keys": "b", "values": 1},
]
```

In Python, those would be passed as `dict` to the calling python function and can thus be indexed by `field: str`. In rust, you'll get a `Series` with the `Struct` type. The fields of the struct can then be indexed and downcast.

Python Rust

API `apply` · API `struct`

```
out = df.select(
    pl.struct(["keys", "values"])
    .apply(lambda x: len(x["keys"]) + x["values"])
    .alias("solution_apply"),
    (pl.col("keys").str.lengths() + pl.col("values")).alias("solution_expr"),
)
print(out)
```

API `apply` · API `Struct` · Available on feature dtype-struct

```
let out = df
    .lazy()
    .select([
        // pack to struct to get access to multiple fields in a custom `apply/map`
        as_struct(&[col("keys"), col("values")])
            // we will compute the len(a) + b
            .apply(
                |s| {
                    // downcast to struct
                    let ca = s.struct_()?;

                    // get the fields as Series
                    let s_a = &ca.fields()[0];
                    let s_b = &ca.fields()[1];

                    // downcast the `Series` to their known type
                    let ca_a = s_a.utf8()?;
                    let ca_b = s_b.i32()?;

                    // iterate both `ChunkedArrays`
                    let out: Int32Chunked = ca_a
                        .into_iter()
                        .zip(ca_b)
                        .map(|(opt_a, opt_b)| match (opt_a, opt_b) {
                            (Some(a), Some(b)) => Some(a.len() as i32 + b),
                            _ => None,
                        })
                        .collect();

                    Ok(out.into_series())
                },
                GetOutput::from_type(DataType::Int32),
            )
            .alias("solution_apply"),
        (col("keys").str().count_match(".") + col("values")).alias("solution_expr"),
    ])
    .collect()?;
println!("{}", out);
```

```
shape: (3, 2)
┌────────────────┬───────────────┐
│ solution_apply ┆ solution_expr │
│ ---            ┆ ---           │
│ i64            ┆ i64           │
╞════════════════╪═══════════════╡
│ 11             ┆ 11            │
│ 8              ┆ 8             │
│ 2              ┆ 2             │
└────────────────┴───────────────┘
```

`Structs` are covered in detail in the next section.

RETURN TYPES?

Custom python functions are black boxes for polars. We really don't know what kind of black arts you are doing, so we have to infer and try our best to understand what you meant.

As a user it helps to understand what we do to better utilize custom functions.

The data type is automatically inferred. We do that by waiting for the first non-null value. That value will then be used to determine the type of the `Series`.

The mapping of python types to polars data types is as follows:

- `int` -> `Int64`
- `float` -> `Float64`
- `bool` -> `Boolean`
- `str` -> `Utf8`
- `list[tp]` -> `List[tp]` (where the inner type is inferred with the same rules)
- `dict[str, [tp]]` -> `struct`
- `Any` -> `object` (Prevent this at all times)

Rust types map as follows:

- `i32` or `i64` -> `Int64`
- `f32` or `f64` -> `Float64`
- `bool` -> `Boolean`
- `String` or `str` -> `Utf8`
- `Vec<tp>` -> `List[tp]` (where the inner type is inferred with the same rules)

4.4.12 The Struct datatype

Polars `Struct` s are the idiomatic way of working with multiple columns. It is also a free operation i.e. moving columns into `Struct` s does not copy any data!

For this section, let's start with a `DataFrame` that captures the average rating of a few movies across some states in the U.S.:

Python

API `DataFrame`

```
ratings = pl.DataFrame(
    {
        "Movie": ["Cars", "IT", "ET", "Cars", "Up", "IT", "Cars", "ET", "Up", "ET"],
        "Theatre": ["NE", "ME", "IL", "ND", "NE", "SD", "NE", "IL", "IL", "SD"],
        "Avg_Rating": [4.5, 4.4, 4.6, 4.3, 4.8, 4.7, 4.7, 4.9, 4.7, 4.6],
        "Count": [30, 27, 26, 29, 31, 28, 28, 26, 33, 26],
    }
)
print(ratings)
```

```
shape: (10, 4)
┌───────┬─────────┬────────────┬───────┐
│ Movie ┆ Theatre ┆ Avg_Rating ┆ Count │
│ ---   ┆ ---     ┆ ---        ┆ ---   │
│ str   ┆ str     ┆ f64        ┆ i64   │
╞═══════╪═════════╪════════════╪═══════╡
│ Cars  ┆ NE      ┆ 4.5        ┆ 30    │
│ IT    ┆ ME      ┆ 4.4        ┆ 27    │
│ ET    ┆ IL      ┆ 4.6        ┆ 26    │
│ Cars  ┆ ND      ┆ 4.3        ┆ 29    │
│ …     ┆ …       ┆ …          ┆ …     │
│ Cars  ┆ NE      ┆ 4.7        ┆ 28    │
│ ET    ┆ IL      ┆ 4.9        ┆ 26    │
│ Up    ┆ IL      ┆ 4.7        ┆ 33    │
│ ET    ┆ SD      ┆ 4.6        ┆ 26    │
└───────┴─────────┴────────────┴───────┘
```

Encountering the `Struct` type

A common operation that will lead to a `Struct` column is the ever so popular `value_counts` function that is commonly used in exploratory data analysis. Checking the number of times a state appears the data will be done as so:

Python

API `value_counts`

```
out = ratings.select(pl.col("Theatre").value_counts(sort=True))
print(out)
```

```
shape: (5, 1)
┌───────────┐
│ Theatre   │
│ ---       │
│ struct[2] │
╞═══════════╡
│ {"NE",3}  │
│ {"IL",3}  │
│ {"SD",2}  │
│ {"ME",1}  │
│ {"ND",1}  │
└───────────┘
```

Quite unexpected an output, especially if coming from tools that do not have such a data type. We're not in peril though, to get back to a more familiar output, all we need to do is `unnest` the `Struct` column into its constituent columns:

Python

API `unnest`

```
out = ratings.select(pl.col("Theatre").value_counts(sort=True)).unnest("Theatre")
print(out)
```

```
shape: (5, 2)
┌─────────┬────────┐
│ Theatre ┆ counts │
│ ---     ┆ ---    │
│ str     ┆ u32    │
╞═════════╪════════╡
│ NE      ┆ 3      │
│ IL      ┆ 3      │
│ SD      ┆ 2      │
│ ME      ┆ 1      │
│ ND      ┆ 1      │
└─────────┴────────┘
```

Why `value_counts` returns a `Struct`

Polars expressions always have a `Fn(Series) -> Series` signature and `Struct` is thus the data type that allows us to provide multiple columns as input/ouput of an expression. In other words, all expressions have to return a `Series` object, and `Struct` allows us to stay consistent with that requirement.

Structs as `dict` s

Polars will interpret a `dict` sent to the `Series` constructor as a `Struct` :

Python

API `Series`

```
rating_Series = pl.Series(
    "ratings",
    [
        {"Movie": "Cars", "Theatre": "NE", "Avg_Rating": 4.5},
        {"Movie": "Toy Story", "Theatre": "ME", "Avg_Rating": 4.9},
    ],
)
print(rating_Series)
```

```
shape: (2,)
Series: 'ratings' [struct[3]]
[
    {"Cars","NE",4.5}
    {"Toy Story","ME",4.9}
]
```

Constructing `Series` objects

Note that `Series` here was constructed with the `name` of the series in the begninng, followed by the `values` . Providing the latter first is considered an anti-pattern in Polars, and must be avoided.

EXTRACTING INDIVIDUAL VALUES OF A `STRUCT`

Let's say that we needed to obtain just the `movie` value in the `Series` that we created above. We can use the `field` method to do so:

Python

API `field`

```
out = rating_Series.struct.field("Movie")
print(out)
```

```
shape: (2,)
Series: 'Movie' [str]
[
    "Cars"
    "Toy Story"
]
```

RENAMING INDIVIDUAL KEYS OF A `STRUCT`

What if we need to rename individual `field`s of a `Struct` column? We first convert the `rating_Series` object to a `DataFrame` so that we can view the changes easily, and then use the `rename_fields` method:

Python

API `rename_fields`

```
out = (
    rating_Series.to_frame()
    .select(pl.col("ratings").struct.rename_fields(["Film", "State", "Value"]))
    .unnest("ratings")
)
print(out)
```

```
shape: (2, 3)
┌───────────┬───────┬───────┐
│ Film      ┆ State ┆ Value │
│ ---       ┆ ---   ┆ ---   │
│ str       ┆ str   ┆ f64   │
╞═══════════╪═══════╪═══════╡
│ Cars      ┆ NE    ┆ 4.5   │
│ Toy Story ┆ ME    ┆ 4.9   │
└───────────┴───────┴───────┘
```

Practical use-cases of `Struct` columns

IDENTIFYING DUPLICATE ROWS

Let's get back to the `ratings` data. We want to identify cases where there are duplicates at a `Movie` and `Theatre` level. This is where the `Struct` datatype shines:

Python

API `is_duplicated` · API `struct`

```
out = ratings.filter(pl.struct("Movie", "Theatre").is_duplicated())
print(out)
```

```
shape: (4, 4)
┌───────┬─────────┬────────────┬───────┐
│ Movie ┆ Theatre ┆ Avg_Rating ┆ Count │
│ ---   ┆ ---     ┆ ---        ┆ ---   │
│ str   ┆ str     ┆ f64        ┆ i64   │
╞═══════╪═════════╪════════════╪═══════╡
│ Cars  ┆ NE      ┆ 4.5        ┆ 30    │
│ ET    ┆ IL      ┆ 4.6        ┆ 26    │
│ Cars  ┆ NE      ┆ 4.7        ┆ 28    │
│ ET    ┆ IL      ┆ 4.9        ┆ 26    │
└───────┴─────────┴────────────┴───────┘
```

We can identify the unique cases at this level also with `is_unique`!

MULTI-COLUMN RANKING

Suppose, given that we know there are duplicates, we want to choose which rank gets a higher priority. We define *Count* of ratings to be more important than the actual `Avg_Rating` themselves, and only use it to break a tie. We can then do:

Python

API `is_duplicated` · API `struct`

```
out = ratings.with_columns(
    pl.struct("Count", "Avg_Rating")
    .rank("dense", descending=True)
    .over("Movie", "Theatre")
    .alias("Rank")
).filter(pl.struct("Movie", "Theatre").is_duplicated())
print(out)
```

```
shape: (4, 5)
┌───────┬─────────┬────────────┬───────┬──────┐
│ Movie ┆ Theatre ┆ Avg_Rating ┆ Count ┆ Rank │
│ ---   ┆ ---     ┆ ---        ┆ ---   ┆ ---  │
│ str   ┆ str     ┆ f64        ┆ i64   ┆ u32  │
╞═══════╪═════════╪════════════╪═══════╪══════╡
│ Cars  ┆ NE      ┆ 4.5        ┆ 30    ┆ 1    │
│ ET    ┆ IL      ┆ 4.6        ┆ 26    ┆ 2    │
│ Cars  ┆ NE      ┆ 4.7        ┆ 28    ┆ 2    │
│ ET    ┆ IL      ┆ 4.9        ┆ 26    ┆ 1    │
└───────┴─────────┴────────────┴───────┴──────┘
```

That's a pretty complex set of requirements done very elegantly in Polars!

USING MULTI-COLUMN APPLY

This was discussed in the previous section on *User Defined Functions*.

4.4.13 Numpy

`Polars` expressions support `NumPy` ufuncs. See here for a list on all supported numpy functions.

This means that if a function is not provided by `Polars`, we can use `NumPy` and we still have fast columnar operation through the `NumPy` API.

EXAMPLE

Python

API `DataFrame` · API `log` · Available on feature numpy

```python
import polars as pl
import numpy as np

df = pl.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})

out = df.select(np.log(pl.all()).suffix("_log"))
print(out)
```

```
shape: (3, 2)
┌──────────┬──────────┐
│ a_log    ┆ b_log    │
│ ---      ┆ ---      │
│ f64      ┆ f64      │
╞══════════╪══════════╡
│ 0.0      ┆ 1.386294 │
│ 0.693147 ┆ 1.609438 │
│ 1.098612 ┆ 1.791759 │
└──────────┴──────────┘
```

INTEROPERABILITY

Polars `Series` have support for NumPy universal functions (ufuncs). Element-wise functions such as `np.exp()`, `np.cos()`, `np.div()`, etc. all work with almost zero overhead.

However, as a Polars-specific remark: missing values are a separate bitmask and are not visible by NumPy. This can lead to a window function or a `np.convolve()` giving flawed or incomplete results.

Convert a Polars `Series` to a NumPy array with the `.to_numpy()` method. Missing values will be replaced by `np.nan` during the conversion. If the `Series` does not include missing values, or those values are not desired anymore, the `.view()` method can be used instead, providing a zero-copy NumPy array of the data.

4.5 Transformations

4.5.1 Joins

Join strategies

`Polars` supports the following join strategies by specifying the `strategy` argument:

- `inner`
- `left`
- `outer`
- `cross`
- `asof`
- `semi`
- `anti`

INNER JOIN

An `inner` join produces a `DataFrame` that contains only the rows where the join key exists in both `DataFrames`. Let's take for example the following two `DataFrames`:

Python

API `DataFrame`

```
df_customers = pl.DataFrame(
    {
        "customer_id": [1, 2, 3],
        "name": ["Alice", "Bob", "Charlie"],
    }
)
print(df_customers)
```

```
shape: (3, 2)
┌─────────────┬─────────┐
│ customer_id ┆ name    │
│ ---         ┆ ---     │
│ i64         ┆ str     │
╞═════════════╪═════════╡
│ 1           ┆ Alice   │
│ 2           ┆ Bob     │
│ 3           ┆ Charlie │
└─────────────┴─────────┘
```

Python

API `DataFrame`

```
df_orders = pl.DataFrame(
    {
        "order_id": ["a", "b", "c"],
        "customer_id": [1, 2, 2],
        "amount": [100, 200, 300],
    }
)
print(df_orders)
```

```
shape: (3, 3)
┌──────────┬─────────────┬────────┐
│ order_id ┆ customer_id ┆ amount │
│ ---      ┆ ---         ┆ ---    │
│ str      ┆ i64         ┆ i64    │
╞══════════╪═════════════╪════════╡
│ a        ┆ 1           ┆ 100    │
│ b        ┆ 2           ┆ 200    │
│ c        ┆ 2           ┆ 300    │
└──────────┴─────────────┴────────┘
```

To get a `DataFrame` with the orders and their associated customer we can do an `inner` join on the `customer_id` column:

Python

API `join`

```
df_inner_customer_join = df_customers.join(df_orders, on="customer_id", how="inner")
print(df_inner_customer_join)
```

```
shape: (3, 4)
┌─────────────┬───────┬──────────┬────────┐
│ customer_id ┆ name  ┆ order_id ┆ amount │
│ ---         ┆ ---   ┆ ---      ┆ ---    │
│ i64         ┆ str   ┆ str      ┆ i64    │
╞═════════════╪═══════╪══════════╪════════╡
│ 1           ┆ Alice ┆ a        ┆ 100    │
│ 2           ┆ Bob   ┆ b        ┆ 200    │
│ 2           ┆ Bob   ┆ c        ┆ 300    │
└─────────────┴───────┴──────────┴────────┘
```

LEFT JOIN

The `left` join 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`. If we now take the example from above and want to have a `DataFrame` with all the customers and their associated orders (regardless of whether they have placed an order or not) we can do a `left` join:

Python

API `join`

```
df_left_join = df_customers.join(df_orders, on="customer_id", how="left")
print(df_left_join)
```

```
shape: (4, 4)
┌─────────────┬─────────┬──────────┬────────┐
│ customer_id ┆ name    ┆ order_id ┆ amount │
│ ---         ┆ ---     ┆ ---      ┆ ---    │
│ i64         ┆ str     ┆ str      ┆ i64    │
╞═════════════╪═════════╪══════════╪════════╡
│ 1           ┆ Alice   ┆ a        ┆ 100    │
│ 2           ┆ Bob     ┆ b        ┆ 200    │
│ 2           ┆ Bob     ┆ c        ┆ 300    │
│ 3           ┆ Charlie ┆ null     ┆ null   │
└─────────────┴─────────┴──────────┴────────┘
```

Notice, that the fields for the customer with the `customer_id` of `3` are null, as there are no orders for this customer.

OUTER JOIN

The `outer` join produces a `DataFrame` that contains all the rows from both `DataFrames`. Columns are null, if the join key does not exist in the source `DataFrame`. Doing an `outer` join on the two `DataFrames` from above produces a similar `DataFrame` to the `left` join:

Python

API `join`

```
df_outer_join = df_customers.join(df_orders, on="customer_id", how="outer")
print(df_outer_join)
```

```
shape: (4, 4)
┌─────────────┬─────────┬──────────┬────────┐
│ customer_id ┆ name    ┆ order_id ┆ amount │
│ ---         ┆ ---     ┆ ---      ┆ ---    │
│ i64         ┆ str     ┆ str      ┆ i64    │
╞═════════════╪═════════╪══════════╪════════╡
│ 1           ┆ Alice   ┆ a        ┆ 100    │
│ 2           ┆ Bob     ┆ b        ┆ 200    │
│ 2           ┆ Bob     ┆ c        ┆ 300    │
│ 3           ┆ Charlie ┆ null     ┆ null   │
└─────────────┴─────────┴──────────┴────────┘
```

CROSS JOIN

A `cross` join is a cartesian product of the two `DataFrames`. This means that every row in the left `DataFrame` is joined with every row in the right `DataFrame`. The `cross` join is useful for creating a `DataFrame` with all possible combinations of the columns in two `DataFrames`. Let's take for example the following two `DataFrames`.

Python

API `DataFrame`

```
df_colors = pl.DataFrame(
    {
        "color": ["red", "blue", "green"],
    }
)
print(df_colors)
```

```
shape: (3, 1)
┌───────┐
│ color │
│ ---   │
│ str   │
╞═══════╡
│ red   │
│ blue  │
│ green │
└───────┘
```

Python

API `DataFrame`

```
df_sizes = pl.DataFrame(
    {
        "size": ["S", "M", "L"],
    }
)
print(df_sizes)
```

```
shape: (3, 1)
┌──────┐
│ size │
│ ---  │
│ str  │
╞══════╡
│ S    │
│ M    │
│ L    │
└──────┘
```

We can now create a `DataFrame` containing all possible combinations of the colors and sizes with a `cross` join:

Python

API `join`

```
df_cross_join = df_colors.join(df_sizes, how="cross")
print(df_cross_join)
```

```
shape: (9, 2)
┌───────┬──────┐
│ color ┆ size │
│ ---   ┆ ---  │
│ str   ┆ str  │
╞═══════╪══════╡
│ red   ┆ S    │
│ red   ┆ M    │
│ red   ┆ L    │
│ blue  ┆ S    │
│ blue  ┆ M    │
│ blue  ┆ L    │
│ green ┆ S    │
│ green ┆ M    │
│ green ┆ L    │
└───────┴──────┘
```

The `inner`, `left`, `outer` and `cross` join strategies are standard amongst dataframe libraries. We provide more details on the less familiar `semi`, `anti` and `asof` join strategies below.

SEMI JOIN

Consider the following scenario: a car rental company has a `DataFrame` showing the cars that it owns with each car having a unique `id`.

Python

API `DataFrame`

```
df_cars = pl.DataFrame(
    {
        "id": ["a", "b", "c"],
        "make": ["ford", "toyota", "bmw"],
    }
)
print(df_cars)
```

```
shape: (3, 2)
┌─────┬────────┐
│ id  ┆ make   │
│ --- ┆ ---    │
│ str ┆ str    │
╞═════╪════════╡
│ a   ┆ ford   │
│ b   ┆ toyota │
│ c   ┆ bmw    │
└─────┴────────┘
```

The company has another `DataFrame` showing each repair job carried out on a vehicle.

Python

API `DataFrame`

```
df_repairs = pl.DataFrame(
    {
        "id": ["c", "c"],
        "cost": [100, 200],
    }
)
print(df_repairs)
```

```
shape: (2, 2)
┌─────┬──────┐
│ id  ┆ cost │
│ --- ┆ ---  │
│ str ┆ i64  │
╞═════╪══════╡
│ c   ┆ 100  │
│ c   ┆ 200  │
└─────┴──────┘
```

You want to answer this question: which of the cars have had repairs carried out?

An inner join does not answer this question directly as it produces a `DataFrame` with multiple rows for each car that has had multiple repair jobs:

Python

API `join`

```
df_inner_join = df_cars.join(df_repairs, on="id", how="inner")
print(df_inner_join)
```

```
shape: (2, 3)
┌─────┬──────┬──────┐
│ id  ┆ make ┆ cost │
│ --- ┆ ---  ┆ ---  │
│ str ┆ str  ┆ i64  │
╞═════╪══════╪══════╡
│ c   ┆ bmw  ┆ 100  │
```

```
│ c   ┆ bmw  ┆ 200 │
└─────┴──────┴─────┘
```

However, a semi join produces a single row for each car that has had a repair job carried out.

Python

API join

```python
df_semi_join = df_cars.join(df_repairs, on="id", how="semi")
print(df_semi_join)
```

```
shape: (1, 2)
┌─────┬──────┐
│ id  ┆ make │
│ --- ┆ ---  │
│ str ┆ str  │
╞═════╪══════╡
│ c   ┆ bmw  │
└─────┴──────┘
```

ANTI JOIN

Continuing this example, an alternative question might be: which of the cars have **not** had a repair job carried out? An anti join produces a `DataFrame` showing all the cars from `df_cars` where the `id` is not present in the `df_repairs` `DataFrame`.

Python

API join

```python
df_anti_join = df_cars.join(df_repairs, on="id", how="anti")
print(df_anti_join)
```

```
shape: (2, 2)
┌─────┬────────┐
│ id  ┆ make   │
│ --- ┆ ---    │
│ str ┆ str    │
╞═════╪════════╡
│ a   ┆ ford   │
│ b   ┆ toyota │
└─────┴────────┘
```

ASOF JOIN

An `asof` join is like a left join except that we match on nearest key rather than equal keys. In `Polars` we can do an asof join with the `join` method and specifying `strategy="asof"`. However, for more flexibility we can use the `join_asof` method.

Consider the following scenario: a stock market broker has a `DataFrame` called `df_trades` showing transactions it has made for different stocks.

Python

API DataFrame

```python
df_trades = pl.DataFrame(
    {
        "time": [
            datetime(2020, 1, 1, 9, 1, 0),
            datetime(2020, 1, 1, 9, 1, 0),
            datetime(2020, 1, 1, 9, 3, 0),
            datetime(2020, 1, 1, 9, 6, 0),
        ],
        "stock": ["A", "B", "B", "C"],
        "trade": [101, 299, 301, 500],
    }
)
print(df_trades)
```

```
shape: (4, 3)
┌─────────────────────┬───────┬───────┐
│ time                ┆ stock ┆ trade │
│ ---                 ┆ ---   ┆ ---   │
│ datetime[μs]        ┆ str   ┆ i64   │
```

```
╞═════════════════════╪═══════╪═══════╡
│ 2020-01-01 09:01:00 ┆ A     ┆ 101   │
│ 2020-01-01 09:01:00 ┆ B     ┆ 299   │
│ 2020-01-01 09:03:00 ┆ B     ┆ 301   │
│ 2020-01-01 09:06:00 ┆ C     ┆ 500   │
└─────────────────────┴───────┴───────┘
```

The broker has another `DataFrame` called `df_quotes` showing prices it has quoted for these stocks.

Python

API `DataFrame`

```
df_quotes = pl.DataFrame(
    {
        "time": [
            datetime(2020, 1, 1, 9, 0, 0),
            datetime(2020, 1, 1, 9, 2, 0),
            datetime(2020, 1, 1, 9, 4, 0),
            datetime(2020, 1, 1, 9, 6, 0),
        ],
        "stock": ["A", "B", "C", "A"],
        "quote": [100, 300, 501, 102],
    }
)

print(df_quotes)
```

```
shape: (4, 3)
┌─────────────────────┬───────┬───────┐
│ time                ┆ stock ┆ quote │
│ ---                 ┆ ---   ┆ ---   │
│ datetime[μs]        ┆ str   ┆ i64   │
╞═════════════════════╪═══════╪═══════╡
│ 2020-01-01 09:00:00 ┆ A     ┆ 100   │
│ 2020-01-01 09:02:00 ┆ B     ┆ 300   │
│ 2020-01-01 09:04:00 ┆ C     ┆ 501   │
│ 2020-01-01 09:06:00 ┆ A     ┆ 102   │
└─────────────────────┴───────┴───────┘
```

You want to produce a `DataFrame` showing for each trade the most recent quote provided *before* the trade. You do this with `join_asof` (using the default `strategy = "backward"`). To avoid joining between trades on one stock with a quote on another you must specify an exact preliminary join on the stock column with `by="stock"` .

Python

API `join_asof`

```
df_asof_join = df_trades.join_asof(df_quotes, on="time", by="stock")
print(df_asof_join)
```

```
shape: (4, 4)
┌─────────────────────┬───────┬───────┬───────┐
│ time                ┆ stock ┆ trade ┆ quote │
│ ---                 ┆ ---   ┆ ---   ┆ ---   │
│ datetime[μs]        ┆ str   ┆ i64   ┆ i64   │
╞═════════════════════╪═══════╪═══════╪═══════╡
│ 2020-01-01 09:01:00 ┆ A     ┆ 101   ┆ 100   │
│ 2020-01-01 09:01:00 ┆ B     ┆ 299   ┆ null  │
│ 2020-01-01 09:03:00 ┆ B     ┆ 301   ┆ 300   │
│ 2020-01-01 09:06:00 ┆ C     ┆ 500   ┆ 501   │
└─────────────────────┴───────┴───────┴───────┘
```

If you want to make sure that only quotes within a certain time range are joined to the trades you can specify the `tolerance` argument. In this case we want to make sure that the last preceding quote is within 1 minute of the trade so we set `tolerance = "1m"` .

Python

```
df_asof_tolerance_join = df_trades.join_asof(
    df_quotes, on="time", by="stock", tolerance="1m"
)
print(df_asof_tolerance_join)
```

```
shape: (4, 4)
┌─────────────────────┬───────┬───────┬───────┐
│ time                ┆ stock ┆ trade ┆ quote │
```

```
│ ---                 ┆ ---    ┆ ---    ┆ ---    │
│ datetime[μs]        ┆ str    ┆ i64    ┆ i64    │
╞═════════════════════╪════════╪════════╪════════╡
│ 2020-01-01 09:01:00 ┆ A      ┆ 101    ┆ 100    │
│ 2020-01-01 09:01:00 ┆ B      ┆ 299    ┆ null   │
│ 2020-01-01 09:03:00 ┆ B      ┆ 301    ┆ 300    │
│ 2020-01-01 09:06:00 ┆ C      ┆ 500    ┆ null   │
└─────────────────────┴────────┴────────┴────────┘
```

4.5.2 Concatenation

There are a number of ways to concatenate data from separate DataFrames:

- two dataframes with **the same columns** can be **vertically** concatenated to make a **longer** dataframe
- two dataframes with the **same number of rows** and **non-overlapping columns** can be **horizontally** concatenated to make a **wider** dataframe
- two dataframes with **different numbers of rows and columns** can be **diagonally** concatenated to make a dataframe which might be longer and/ or wider. Where column names overlap values will be vertically concatenated. Where column names do not overlap new rows and columns will be added. Missing values will be set as `null`

Vertical concatenation - getting longer

In a vertical concatenation you combine all of the rows from a list of `DataFrames` into a single longer `DataFrame`.

Python

API `concat`

```
df_v1 = pl.DataFrame(
    {
        "a": [1],
        "b": [3],
    }
)
df_v2 = pl.DataFrame(
    {
        "a": [2],
        "b": [4],
    }
)
df_vertical_concat = pl.concat(
    [
        df_v1,
        df_v2,
    ],
    how="vertical",
)
print(df_vertical_concat)
```

```
shape: (2, 2)
┌─────┬─────┐
│ a   ┆ b   │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 1   ┆ 3   │
│ 2   ┆ 4   │
└─────┴─────┘
```

Vertical concatenation fails when the dataframes do not have the same column names.

Horizontal concatenation - getting wider

In a horizontal concatenation you combine all of the columns from a list of `DataFrames` into a single wider `DataFrame`.

Python

API `concat`

```python
df_h1 = pl.DataFrame(
    {
        "l1": [1, 2],
        "l2": [3, 4],
    }
)
df_h2 = pl.DataFrame(
    {
        "r1": [5, 6],
        "r2": [7, 8],
        "r3": [9, 10],
    }
)
df_horizontal_concat = pl.concat(
    [
        df_h1,
        df_h2,
    ],
    how="horizontal",
)
print(df_horizontal_concat)
```

```
shape: (2, 5)
┌─────┬─────┬─────┬─────┬─────┐
│ l1  ┆ l2  ┆ r1  ┆ r2  ┆ r3  │
│ --- ┆ --- ┆ --- ┆ --- ┆ --- │
│ i64 ┆ i64 ┆ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╪═════╪═════╡
│ 1   ┆ 3   ┆ 5   ┆ 7   ┆ 9   │
│ 2   ┆ 4   ┆ 6   ┆ 8   ┆ 10  │
└─────┴─────┴─────┴─────┴─────┘
```

Horizontal concatenation fails when dataframes have overlapping columns or a different number of rows.

Diagonal concatenation - getting longer, wider and `null` ier

In a diagonal concatenation you combine all of the row and columns from a list of `DataFrames` into a single longer and/or wider `DataFrame`.

Python

API `concat`

```python
df_d1 = pl.DataFrame(
    {
        "a": [1],
        "b": [3],
    }
)
df_d2 = pl.DataFrame(
    {
        "a": [2],
        "d": [4],
    }
)

df_diagonal_concat = pl.concat(
    [
        df_d1,
        df_d2,
    ],
    how="diagonal",
)
print(df_diagonal_concat)
```

```
shape: (2, 3)
┌─────┬──────┬──────┐
│ a   ┆ b    ┆ d    │
│ --- ┆ ---  ┆ ---  │
│ i64 ┆ i64  ┆ i64  │
╞═════╪══════╪══════╡
│ 1   ┆ 3    ┆ null │
```

```
│ 2   │ null │ 4   │
└─────┴──────┴─────┘
```

Diagonal concatenation generates nulls when the column names do not overlap.

When the dataframe shapes do not match and we have an overlapping semantic key then we can join the dataframes instead of concatenating them.

Rechunking

Before a concatenation we have two dataframes `df1` and `df2`. Each column in `df1` and `df2` is in one or more chunks in memory. By default, during concatenation the chunks in each column are copied to a single new chunk - this is known as **rechunking**. Rechunking is an expensive operation, but is often worth it because future operations will be faster. If you do not want Polars to rechunk the concatenated `DataFrame` you specify `rechunk = False` when doing the concatenation.

4.5.3 Pivots

Pivot a column in a `DataFrame` and perform one of the following aggregations:

- first
- sum
- min
- max
- mean
- median

The pivot operation consists of a group by one, or multiple columns (these will be the new y-axis), the column that will be pivoted (this will be the new x-axis) and an aggregation.

Dataset

Python

API `DataFrame`

```python
df = pl.DataFrame(
    {
        "foo": ["A", "A", "B", "B", "C"],
        "N": [1, 2, 2, 4, 2],
        "bar": ["k", "l", "m", "n", "o"],
    }
)
print(df)
```

```
shape: (5, 3)
┌─────┬─────┬─────┐
│ foo ┆ N   ┆ bar │
│ --- ┆ --- ┆ --- │
│ str ┆ i64 ┆ str │
╞═════╪═════╪═════╡
│ A   ┆ 1   ┆ k   │
│ A   ┆ 2   ┆ l   │
│ B   ┆ 2   ┆ m   │
│ B   ┆ 4   ┆ n   │
│ C   ┆ 2   ┆ o   │
└─────┴─────┴─────┘
```

Eager

Python

API `pivot`

```python
out = df.pivot(index="foo", columns="bar", values="N", aggregate_function="first")
print(out)
```

```
shape: (3, 6)
┌─────┬──────┬──────┬──────┬──────┬──────┐
│ foo ┆ k    ┆ l    ┆ m    ┆ n    ┆ o    │
│ --- ┆ ---  ┆ ---  ┆ ---  ┆ ---  ┆ ---  │
│ str ┆ i64  ┆ i64  ┆ i64  ┆ i64  ┆ i64  │
╞═════╪══════╪══════╪══════╪══════╪══════╡
│ A   ┆ 1    ┆ 2    ┆ null ┆ null ┆ null │
│ B   ┆ null ┆ null ┆ 2    ┆ 4    ┆ null │
│ C   ┆ null ┆ null ┆ null ┆ null ┆ 2    │
└─────┴──────┴──────┴──────┴──────┴──────┘
```

Lazy

A polars `LazyFrame` always need to know the schema of a computation statically (before collecting the query). As a pivot's output schema depends on the data, and it is therefore impossible to determine the schema without running the query.

Polars could have abstracted this fact for you just like Spark does, but we don't want you to shoot yourself in the foot with a shotgun. The cost should be clear upfront.

Python

API `pivot`

```python
q = (
    df.lazy()
    .collect()
    .pivot(index="foo", columns="bar", values="N", aggregate_function="first")
    .lazy()
)
out = q.collect()
print(out)
```

```
shape: (3, 6)
┌─────┬──────┬──────┬──────┬──────┬──────┐
│ foo ┆ k    ┆ l    ┆ m    ┆ n    ┆ o    │
│ --- ┆ ---  ┆ ---  ┆ ---  ┆ ---  ┆ ---  │
│ str ┆ i64  ┆ i64  ┆ i64  ┆ i64  ┆ i64  │
╞═════╪══════╪══════╪══════╪══════╪══════╡
│ A   ┆ 1    ┆ 2    ┆ null ┆ null ┆ null │
│ B   ┆ null ┆ null ┆ 2    ┆ 4    ┆ null │
│ C   ┆ null ┆ null ┆ null ┆ null ┆ 2    │
└─────┴──────┴──────┴──────┴──────┴──────┘
```

4.5.4 Melts

Melt operations unpivot a DataFrame from wide format to long format

Dataset

Python

API `DataFrame`

```
import polars as pl

df = pl.DataFrame(
    {
        "A": ["a", "b", "a"],
        "B": [1, 3, 5],
        "C": [10, 11, 12],
        "D": [2, 4, 6],
    }
)
print(df)
```

```
shape: (3, 4)
┌─────┬─────┬─────┬─────┐
│ A   ┆ B   ┆ C   ┆ D   │
│ --- ┆ --- ┆ --- ┆ --- │
│ str ┆ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╪═════╡
│ a   ┆ 1   ┆ 10  ┆ 2   │
│ b   ┆ 3   ┆ 11  ┆ 4   │
│ a   ┆ 5   ┆ 12  ┆ 6   │
└─────┴─────┴─────┴─────┘
```

Eager + Lazy

`Eager` and `lazy` have the same API.

Python

API `melt`

```
out = df.melt(id_vars=["A", "B"], value_vars=["C", "D"])
print(out)
```

```
shape: (6, 4)
┌─────┬─────┬──────────┬───────┐
│ A   ┆ B   ┆ variable ┆ value │
│ --- ┆ --- ┆ ---      ┆ ---   │
│ str ┆ i64 ┆ str      ┆ i64   │
╞═════╪═════╪══════════╪═══════╡
│ a   ┆ 1   ┆ C        ┆ 10    │
│ b   ┆ 3   ┆ C        ┆ 11    │
│ a   ┆ 5   ┆ C        ┆ 12    │
│ a   ┆ 1   ┆ D        ┆ 2     │
│ b   ┆ 3   ┆ D        ┆ 4     │
│ a   ┆ 5   ┆ D        ┆ 6     │
└─────┴─────┴──────────┴───────┘
```

4.5.5 Time Series

Parsing

Polars has native support for parsing time series data and doing more sophisticated operations such as temporal grouping and resampling.

DATATYPES

`Polars` has the following datetime datatypes:

- `Date` : Date representation e.g. 2014-07-08. It is internally represented as days since UNIX epoch encoded by a 32-bit signed integer.
- `Datetime` : Datetime representation e.g. 2014-07-08 07:00:00. It is internally represented as a 64 bit integer since the Unix epoch and can have different units such as ns, us, ms.
- `Duration` : A time delta type that is created when subtracting `Date/Datetime` . Similar to `timedelta` in python.
- `Time` : Time representation, internally represented as nanoseconds since midnight.

PARSING DATES FROM A FILE

When loading from a CSV file `Polars` attempts to parse dates and times if the `try_parse_dates` flag is set to `True` :

Python

API `read_csv`

```
df = pl.read_csv("docs/src/data/appleStock.csv", try_parse_dates=True)
print(df)
```

```
shape: (100, 2)
┌────────────┬────────┐
│ Date       ┆ Close  │
│ ---        ┆ ---    │
│ date       ┆ f64    │
╞════════════╪════════╡
│ 1981-02-23 ┆ 24.62  │
│ 1981-05-06 ┆ 27.38  │
│ 1981-05-18 ┆ 28.0   │
│ 1981-09-25 ┆ 14.25  │
│ …          ┆ …      │
│ 2012-12-04 ┆ 575.85 │
│ 2013-07-05 ┆ 417.42 │
│ 2013-11-07 ┆ 512.49 │
│ 2014-02-25 ┆ 522.06 │
└────────────┴────────┘
```

On the other hand binary formats such as parquet have a schema that is respected by `Polars` .

CASTING STRINGS TO DATES

You can also cast a column of datetimes encoded as strings to a datetime type. You do this by calling the string `str.strptime` method and passing the format of the date string:

Python

API `read_csv` · API `strptime`

```
df = pl.read_csv("docs/src/data/appleStock.csv", try_parse_dates=False)

df = df.with_columns(pl.col("Date").str.strptime(pl.Date, format="%Y-%m-%d"))
print(df)
```

```
shape: (100, 2)
┌────────────┬────────┐
│ Date       ┆ Close  │
│ ---        ┆ ---    │
│ date       ┆ f64    │
╞════════════╪════════╡
│ 1981-02-23 ┆ 24.62  │
│ 1981-05-06 ┆ 27.38  │
│ 1981-05-18 ┆ 28.0   │
│ 1981-09-25 ┆ 14.25  │
│ …          ┆ …      │
```

```
│ 2012-12-04 ┆ 575.85 │
│ 2013-07-05 ┆ 417.42 │
│ 2013-11-07 ┆ 512.49 │
│ 2014-02-25 ┆ 522.06 │
└────────────┴────────┘
```

The strptime date formats can be found here..

EXTRACTING DATE FEATURES FROM A DATE COLUMN

You can extract data features such as the year or day from a date column using the `.dt` namespace on a date column:

Python

API `year`

```
df_with_year = df.with_columns(pl.col("Date").dt.year().alias("year"))
print(df_with_year)
```

```
shape: (100, 3)
┌────────────┬────────┬──────┐
│ Date       ┆ Close  ┆ year │
│ ---        ┆ ---    ┆ ---  │
│ date       ┆ f64    ┆ i32  │
╞════════════╪════════╪══════╡
│ 1981-02-23 ┆ 24.62  ┆ 1981 │
│ 1981-05-06 ┆ 27.38  ┆ 1981 │
│ 1981-05-18 ┆ 28.0   ┆ 1981 │
│ 1981-09-25 ┆ 14.25  ┆ 1981 │
│ …          ┆ …      ┆ …    │
│ 2012-12-04 ┆ 575.85 ┆ 2012 │
│ 2013-07-05 ┆ 417.42 ┆ 2013 │
│ 2013-11-07 ┆ 512.49 ┆ 2013 │
│ 2014-02-25 ┆ 522.06 ┆ 2014 │
└────────────┴────────┴──────┘
```

MIXED OFFSETS

If you have mixed offsets (say, due to crossing daylight saving time), then you can use `utc=True` and then convert to your time zone:

Python

API `strptime` · API `convert_time_zone` · Available on feature timezone

```
data = [
    "2021-03-27T00:00:00+0100",
    "2021-03-28T00:00:00+0100",
    "2021-03-29T00:00:00+0200",
    "2021-03-30T00:00:00+0200",
]
mixed_parsed = (
    pl.Series(data)
    .str.strptime(pl.Datetime, format="%Y-%m-%dT%H:%M:%S%z", utc=True)
    .dt.convert_time_zone("Europe/Brussels")
)
print(mixed_parsed)
```

```
shape: (4,)
Series: '' [datetime[μs, Europe/Brussels]]
[
    2021-03-27 00:00:00 CET
    2021-03-28 00:00:00 CET
    2021-03-29 00:00:00 CEST
    2021-03-30 00:00:00 CEST
]
```

Filtering

Filtering date columns works in the same way as with other types of columns using the `.filter` method.

Polars uses Python's native `datetime`, `date` and `timedelta` for equality comparisons between the datatypes `pl.Datetime`, `pl.Date` and `pl.Duration`.

In the following example we use a time series of Apple stock prices.

Python

API `read_csv`

```
import polars as pl
from datetime import datetime

df = pl.read_csv("docs/src/data/appleStock.csv", try_parse_dates=True)
print(df)
```

```
shape: (100, 2)
┌────────────┬────────┐
│ Date       ┆ Close  │
│ ---        ┆ ---    │
│ date       ┆ f64    │
╞════════════╪════════╡
│ 1981-02-23 ┆ 24.62  │
│ 1981-05-06 ┆ 27.38  │
│ 1981-05-18 ┆ 28.0   │
│ 1981-09-25 ┆ 14.25  │
│ …          ┆ …      │
│ 2012-12-04 ┆ 575.85 │
│ 2013-07-05 ┆ 417.42 │
│ 2013-11-07 ┆ 512.49 │
│ 2014-02-25 ┆ 522.06 │
└────────────┴────────┘
```

FILTERING BY SINGLE DATES

We can filter by a single date by casting the desired date string to a `Date` object in a filter expression:

Python

API `filter`

```
filtered_df = df.filter(
    pl.col("Date") == datetime(1995, 10, 16),
)
print(filtered_df)
```

```
shape: (1, 2)
┌────────────┬───────┐
│ Date       ┆ Close │
│ ---        ┆ ---   │
│ date       ┆ f64   │
╞════════════╪═══════╡
│ 1995-10-16 ┆ 36.13 │
└────────────┴───────┘
```

Note we are using the lowercase `datetime` method rather than the uppercase `Datetime` data type.

FILTERING BY A DATE RANGE

We can filter by a range of dates using the `is_between` method in a filter expression with the start and end dates:

Python

API `filter` · API `is_between`

```
filtered_range_df = df.filter(
    pl.col("Date").is_between(datetime(1995, 7, 1), datetime(1995, 11, 1)),
)
print(filtered_range_df)
```

```
shape: (2, 2)
┌────────────┬───────┐
│ Date       ┆ Close │
│ ---        ┆ ---   │
│ date       ┆ f64   │
╞════════════╪═══════╡
│ 1995-07-06 ┆ 47.0  │
│ 1995-10-16 ┆ 36.13 │
└────────────┴───────┘
```

FILTERING WITH NEGATIVE DATES

Say you are working with an archeologist and are dealing in negative dates. Polars can parse and store them just fine, but the Python `datetime` library does not. So for filtering, you should use attributes in the `.dt` namespace:

Python

API `strptime`

```python
ts = pl.Series(["-1300-05-23", "-1400-03-02"]).str.strptime(pl.Date)

negative_dates_df = pl.DataFrame({"ts": ts, "values": [3, 4]})

negative_dates_filtered_df = negative_dates_df.filter(pl.col("ts").dt.year() < -1300)
print(negative_dates_filtered_df)
```

```
shape: (1, 2)
┌─────────────┬────────┐
│ ts          ┆ values │
│ ---         ┆ ---    │
│ date        ┆ i64    │
╞═════════════╪════════╡
│ -1400-03-02 ┆ 4      │
└─────────────┴────────┘
```

Grouping

GROUPING BY FIXED WINDOWS

We can calculate temporal statistics using `groupby_dynamic` to group rows into days/months/years etc.

Annual average example

In following simple example we calculate the annual average closing price of Apple stock prices. We first load the data from CSV:

Python

API `upsample`

```
df = pl.read_csv("docs/src/data/appleStock.csv", try_parse_dates=True)
df = df.sort("Date")
print(df)
```

```
shape: (100, 2)
┌────────────┬────────┐
│ Date       ┆ Close  │
│ ---        ┆ ---    │
│ date       ┆ f64    │
╞════════════╪════════╡
│ 1981-02-23 ┆ 24.62  │
│ 1981-05-06 ┆ 27.38  │
│ 1981-05-18 ┆ 28.0   │
│ 1981-09-25 ┆ 14.25  │
│ …          ┆ …      │
│ 2012-12-04 ┆ 575.85 │
│ 2013-07-05 ┆ 417.42 │
│ 2013-11-07 ┆ 512.49 │
│ 2014-02-25 ┆ 522.06 │
└────────────┴────────┘
```

Info

The dates are sorted in ascending order - if they are not sorted in this way the `groupby_dynamic` output will not be correct!

To get the annual average closing price we tell `groupby_dynamic` that we want to:

- group by the `Date` column on an annual (`1y`) basis
- take the mean values of the `Close` column for each year:

Python

API `groupby_dynamic`

```
annual_average_df = df.groupby_dynamic("Date", every="1y").agg(pl.col("Close").mean())

df_with_year = annual_average_df.with_columns(pl.col("Date").dt.year().alias("year"))
print(df_with_year)
```

The annual average closing price is then:

```
shape: (34, 3)
┌────────────┬───────────┬──────┐
│ Date       ┆ Close     ┆ year │
│ ---        ┆ ---       ┆ ---  │
│ date       ┆ f64       ┆ i32  │
╞════════════╪═══════════╪══════╡
│ 1981-01-01 ┆ 23.5625   ┆ 1981 │
│ 1982-01-01 ┆ 11.0      ┆ 1982 │
│ 1983-01-01 ┆ 30.543333 ┆ 1983 │
│ 1984-01-01 ┆ 27.583333 ┆ 1984 │
│ …          ┆ …         ┆ …    │
│ 2011-01-01 ┆ 368.225   ┆ 2011 │
│ 2012-01-01 ┆ 560.965   ┆ 2012 │
│ 2013-01-01 ┆ 464.955   ┆ 2013 │
│ 2014-01-01 ┆ 522.06    ┆ 2014 │
└────────────┴───────────┴──────┘
```

Parameters for `groupby_dynamic`

A dynamic window is defined by a:

- **every**: indicates the interval of the window
- **period**: indicates the duration of the window
- **offset**: can be used to offset the start of the windows

The value for `every` sets how often the groups start. The time period values are flexible - for example we could take:

- the average over 2 year intervals by replacing `1y` with `2y`
- the average over 18 month periods by replacing `1y` with `1y6mo`

We can also use the `period` parameter to set how long the time period for each group is. For example, if we set the `every` parameter to be `1y` and the `period` parameter to be `2y` then we would get groups at one year intervals where each groups spanned two years.

If the `period` parameter is not specified then it is set equal to the `every` parameter so that if the `every` parameter is set to be `1y` then each group spans `1y` as well.

Because ***every*** does not have to be equal to ***period***, we can create many groups in a very flexible way. They may overlap or leave boundaries between them.

Let's see how the windows for some parameter combinations would look. Let's start out boring. 🥱

- every: 1 day -> `"1d"`
- period: 1 day -> `"1d"`

```
this creates adjacent windows of the same size
|--|
   |--|
      |--|
```

- every: 1 day -> `"1d"`
- period: 2 days -> `"2d"`

```
these windows have an overlap of 1 day
|----|
   |----|
      |----|
```

- every: 2 days -> `"2d"`
- period: 1 day -> `"1d"`

```
this would leave gaps between the windows
data points that in these gaps will not be a member of any group
|--|
       |--|
              |--|
```

`truncate`

The `truncate` parameter is a Boolean variable that determines what datetime value is associated with each group in the output. In the example above the first data point is on 23rd February 1981. If `truncate = True` (the default) then the date for the first year in the annual average is 1st January 1981. However, if `truncate = False` then the date for the first year in the annual average is the date of the first data point on 23rd February 1981. Note that `truncate` only affects what's shown in the `Date` column and does not affect the window boundaries.

Using expressions in `groupby_dynamic`

We aren't restricted to using simple aggregations like `mean` in a groupby operation - we can use the full range of expressions available in Polars.

In the snippet below we create a `date range` with every **day** (`"1d"`) in 2021 and turn this into a `DataFrame` .

Then in the `groupby_dynamic` we create dynamic windows that start every **month** (`"1mo"`) and have a window length of `1` month. The values that match these dynamic windows are then assigned to that group and can be aggregated with the powerful expression API.

Below we show an example where we use **groupby_dynamic** to compute:

- the number of days until the end of the month
- the number of days in a month

Python

API `groupby_dynamic` · API `explode` · API `date_range`

```
df = (
    pl.date_range(
        start=datetime(2021, 1, 1),
        end=datetime(2021, 12, 31),
        interval="1d",
        eager=True,
    )
    .alias("time")
    .to_frame()
)

out = (
    df.groupby_dynamic("time", every="1mo", period="1mo", closed="left")
    .agg(
        [
            pl.col("time").cumcount().reverse().head(3).alias("day/eom"),
            ((pl.col("time") - pl.col("time").first()).last().dt.days() + 1).alias(
                "days_in_month"
            ),
        ]
    )
    .explode("day/eom")
)
print(out)
```

```
shape: (36, 3)
┌─────────────────────┬─────────┬───────────────┐
│ time                ┆ day/eom ┆ days_in_month │
│ ---                 ┆ ---     ┆ ---           │
│ datetime[μs]        ┆ u32     ┆ i64           │
╞═════════════════════╪═════════╪═══════════════╡
│ 2021-01-01 00:00:00 ┆ 30      ┆ 31            │
│ 2021-01-01 00:00:00 ┆ 29      ┆ 31            │
│ 2021-01-01 00:00:00 ┆ 28      ┆ 31            │
│ 2021-02-01 00:00:00 ┆ 27      ┆ 28            │
│ …                   ┆ …       ┆ …             │
│ 2021-11-01 00:00:00 ┆ 27      ┆ 30            │
│ 2021-12-01 00:00:00 ┆ 30      ┆ 31            │
│ 2021-12-01 00:00:00 ┆ 29      ┆ 31            │
│ 2021-12-01 00:00:00 ┆ 28      ┆ 31            │
└─────────────────────┴─────────┴───────────────┘
```

GROUPING BY ROLLING WINDOWS

The rolling groupby, `groupby_rolling`, is another entrance to the `groupby` context. But different from the `groupby_dynamic` the windows are not fixed by a parameter `every` and `period`. In a rolling groupby the windows are not fixed at all! They are determined by the values in the `index_column`.

So imagine having a time column with the values `{2021-01-06, 2021-01-10}` and a `period="5d"` this would create the following windows:

```
2021-01-01   2021-01-06
    |----------|

       2021-01-05   2021-01-10
             |----------|
```

Because the windows of a rolling groupby are always determined by the values in the `DataFrame` column, the number of groups is always equal to the original `DataFrame`.

COMBINING GROUPBY'S

Rolling and dynamic groupby's can be combined with normal groupby operations.

Below is an example with a dynamic groupby.

Python

API `DataFrame`

```
df = pl.DataFrame(
    {
        "time": pl.date_range(
            start=datetime(2021, 12, 16),
            end=datetime(2021, 12, 16, 3),
            interval="30m",
            eager=True,
        ),
        "groups": ["a", "a", "a", "b", "b", "a", "a"],
    }
)
print(df)
```

```
shape: (7, 2)
┌─────────────────────┬────────┐
│ time                ┆ groups │
│ ---                 ┆ ---    │
│ datetime[μs]        ┆ str    │
╞═════════════════════╪════════╡
│ 2021-12-16 00:00:00 ┆ a      │
│ 2021-12-16 00:30:00 ┆ a      │
│ 2021-12-16 01:00:00 ┆ a      │
│ 2021-12-16 01:30:00 ┆ b      │
│ 2021-12-16 02:00:00 ┆ b      │
│ 2021-12-16 02:30:00 ┆ a      │
│ 2021-12-16 03:00:00 ┆ a      │
└─────────────────────┴────────┘
```

Python

API `groupby_dynamic`

```
out = df.groupby_dynamic(
    "time",
    every="1h",
    closed="both",
    by="groups",
    include_boundaries=True,
).agg(
    [
        pl.count(),
    ]
)
print(out)
```

```
shape: (7, 5)
┌────────┬─────────────────────┬─────────────────────┬─────────────────────┬───────┐
│ groups ┆ _lower_boundary     ┆ _upper_boundary     ┆ time                ┆ count │
│ ---    ┆ ---                 ┆ ---                 ┆ ---                 ┆ ---   │
│ str    ┆ datetime[μs]        ┆ datetime[μs]        ┆ datetime[μs]        ┆ u32   │
╞════════╪═════════════════════╪═════════════════════╪═════════════════════╪═══════╡
│ a      ┆ 2021-12-15 23:00:00 ┆ 2021-12-16 00:00:00 ┆ 2021-12-15 23:00:00 ┆ 1     │
│ a      ┆ 2021-12-16 00:00:00 ┆ 2021-12-16 01:00:00 ┆ 2021-12-16 00:00:00 ┆ 3     │
│ a      ┆ 2021-12-16 01:00:00 ┆ 2021-12-16 02:00:00 ┆ 2021-12-16 01:00:00 ┆ 1     │
│ a      ┆ 2021-12-16 02:00:00 ┆ 2021-12-16 03:00:00 ┆ 2021-12-16 02:00:00 ┆ 2     │
│ a      ┆ 2021-12-16 03:00:00 ┆ 2021-12-16 04:00:00 ┆ 2021-12-16 03:00:00 ┆ 1     │
│ b      ┆ 2021-12-16 01:00:00 ┆ 2021-12-16 02:00:00 ┆ 2021-12-16 01:00:00 ┆ 2     │
│ b      ┆ 2021-12-16 02:00:00 ┆ 2021-12-16 03:00:00 ┆ 2021-12-16 02:00:00 ┆ 1     │
└────────┴─────────────────────┴─────────────────────┴─────────────────────┴───────┘
```

Resampling

We can resample by either:

- upsampling (moving data to a higher frequency)
- downsampling (moving data to a lower frequency)
- combinations of these e.g. first upsample and then downsample

DOWNSAMPLING TO A LOWER FREQUENCY

`Polars` views downsampling as a special case of the **groupby** operation and you can do this with `groupby_dynamic` and `groupby_rolling` - see the temporal groupby page for examples.

UPSAMPLING TO A HIGHER FREQUENCY

Let's go through an example where we generate data at 30 minute intervals:

Python

API `DataFrame` · API `date_range`

```
df = pl.DataFrame(
    {
        "time": pl.date_range(
            start=datetime(2021, 12, 16),
            end=datetime(2021, 12, 16, 3),
            interval="30m",
            eager=True,
        ),
        "groups": ["a", "a", "a", "b", "b", "a", "a"],
        "values": [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0],
    }
)
print(df)
```

```
shape: (7, 3)
┌─────────────────────┬────────┬────────┐
│ time                ┆ groups ┆ values │
│ ---                 ┆ ---    ┆ ---    │
│ datetime[μs]        ┆ str    ┆ f64    │
╞═════════════════════╪════════╪════════╡
│ 2021-12-16 00:00:00 ┆ a      ┆ 1.0    │
│ 2021-12-16 00:30:00 ┆ a      ┆ 2.0    │
│ 2021-12-16 01:00:00 ┆ a      ┆ 3.0    │
│ 2021-12-16 01:30:00 ┆ b      ┆ 4.0    │
│ 2021-12-16 02:00:00 ┆ b      ┆ 5.0    │
│ 2021-12-16 02:30:00 ┆ a      ┆ 6.0    │
│ 2021-12-16 03:00:00 ┆ a      ┆ 7.0    │
└─────────────────────┴────────┴────────┘
```

Upsampling can be done by defining the new sampling interval. By upsampling we are adding in extra rows where we do not have data. As such upsampling by itself gives a DataFrame with nulls. These nulls can then be filled with a fill strategy or interpolation.

Upsampling strategies

In this example we upsample from the original 30 minutes to 15 minutes and then use a `forward` strategy to replace the nulls with the previous non-null value:

Python

API `upsample`

```
out1 = df.upsample(time_column="time", every="15m").fill_null(strategy="forward")
print(out1)
```

```
shape: (13, 3)
┌─────────────────────┬────────┬────────┐
│ time                ┆ groups ┆ values │
│ ---                 ┆ ---    ┆ ---    │
│ datetime[μs]        ┆ str    ┆ f64    │
╞═════════════════════╪════════╪════════╡
│ 2021-12-16 00:00:00 ┆ a      ┆ 1.0    │
│ 2021-12-16 00:15:00 ┆ a      ┆ 1.0    │
│ 2021-12-16 00:30:00 ┆ a      ┆ 2.0    │
│ 2021-12-16 00:45:00 ┆ a      ┆ 2.0    │
```

```
│ …                   ┆ …      ┆ …      │
│ 2021-12-16 02:15:00 ┆ b      ┆ 5.0    │
│ 2021-12-16 02:30:00 ┆ a      ┆ 6.0    │
│ 2021-12-16 02:45:00 ┆ a      ┆ 6.0    │
│ 2021-12-16 03:00:00 ┆ a      ┆ 7.0    │
└─────────────────────┴────────┴────────┘
```

In this example we instead fill the nulls by linear interpolation:

Python

API upsample · API interpolate · API fill_null

```python
out2 = (
    df.upsample(time_column="time", every="15m")
    .interpolate()
    .fill_null(strategy="forward")
)
print(out2)
```

```
shape: (13, 3)
┌─────────────────────┬────────┬────────┐
│ time                ┆ groups ┆ values │
│ ---                 ┆ ---    ┆ ---    │
│ datetime[μs]        ┆ str    ┆ f64    │
╞═════════════════════╪════════╪════════╡
│ 2021-12-16 00:00:00 ┆ a      ┆ 1.0    │
│ 2021-12-16 00:15:00 ┆ a      ┆ 1.5    │
│ 2021-12-16 00:30:00 ┆ a      ┆ 2.0    │
│ 2021-12-16 00:45:00 ┆ a      ┆ 2.5    │
│ …                   ┆ …      ┆ …      │
│ 2021-12-16 02:15:00 ┆ b      ┆ 5.5    │
│ 2021-12-16 02:30:00 ┆ a      ┆ 6.0    │
│ 2021-12-16 02:45:00 ┆ a      ┆ 6.5    │
│ 2021-12-16 03:00:00 ┆ a      ┆ 7.0    │
└─────────────────────┴────────┴────────┘
```

Time zones

> **Tom Scott**
>
> You really should never, ever deal with time zones if you can help it

The `Datetime` datatype can have a time zone associated with it. Examples of valid time zones are:

- `None` : no time zone, also known as "time zone naive";
- `UTC` : Coordinated Universal Time;
- `Asia/Kathmandu` : time zone in "area/location" format. See the list of tz database time zones to see what's available;
- `+01:00` : fixed offsets. May be useful when parsing, but you almost certainly want the "Area/Location" format above instead as it will deal with irregularities such as DST (Daylight Saving Time) for you.

Note that, because a `Datetime` can only have a single time zone, it is impossible to have a column with multiple time zones. If you are parsing data with multiple offsets, you may want to pass `utc=True` to convert them all to a common time zone (`UTC`), see parsing dates and times.

The main methods for setting and converting between time zones are:

- `dt.convert_time_zone` : convert from one time zone to another;
- `dt.replace_time_zone` : set/unset/change time zone;

Let's look at some examples of common operations:

Python

API `strptime` · API `replace_time_zone` · Available on feature timezone

```python
ts = ["2021-03-27 03:00", "2021-03-28 03:00"]
tz_naive = pl.Series("tz_naive", ts).str.strptime(pl.Datetime)
tz_aware = tz_naive.dt.replace_time_zone("UTC").rename("tz_aware")
time_zones_df = pl.DataFrame([tz_naive, tz_aware])
print(time_zones_df)
```

```
shape: (2, 2)
┌─────────────────────┬─────────────────────────┐
│ tz_naive            ┆ tz_aware                │
│ ---                 ┆ ---                     │
│ datetime[μs]        ┆ datetime[μs, UTC]       │
╞═════════════════════╪═════════════════════════╡
│ 2021-03-27 03:00:00 ┆ 2021-03-27 03:00:00 UTC │
│ 2021-03-28 03:00:00 ┆ 2021-03-28 03:00:00 UTC │
└─────────────────────┴─────────────────────────┘
```

Python

API `convert_time_zone` · API `replace_time_zone` · Available on feature timezone

```python
time_zones_operations = time_zones_df.select(
    [
        pl.col("tz_aware")
        .dt.replace_time_zone("Europe/Brussels")
        .alias("replace time zone"),
        pl.col("tz_aware")
        .dt.convert_time_zone("Asia/Kathmandu")
        .alias("convert time zone"),
        pl.col("tz_aware").dt.replace_time_zone(None).alias("unset time zone"),
    ]
)
print(time_zones_operations)
```

```
shape: (2, 3)
┌───────────────────────────────┬──────────────────────────────┬─────────────────────┐
│ replace time zone             ┆ convert time zone            ┆ unset time zone     │
│ ---                           ┆ ---                          ┆ ---                 │
│ datetime[μs, Europe/Brussels] ┆ datetime[μs, Asia/Kathmandu] ┆ datetime[μs]        │
╞═══════════════════════════════╪══════════════════════════════╪═════════════════════╡
```

```
│ 2021-03-27 03:00:00 CET        │ 2021-03-27 08:45:00 +0545     │ 2021-03-27 03:00:00 │
│ 2021-03-28 03:00:00 CEST       │ 2021-03-28 08:45:00 +0545     │ 2021-03-28 03:00:00 │
└────────────────────────────────┴───────────────────────────────┴─────────────────────┘
```

4.6 Lazy API

4.6.1 Usage

With the lazy API, Polars doesn't run each query line-by-line but instead processes the full query end-to-end. To get the most out of Polars it is important that you use the lazy API because:

- the lazy API allows Polars to apply automatic query optimization with the query optimizer
- the lazy API allows you to work with larger than memory datasets using streaming
- the lazy API can catch schema errors before processing the data

Here we see how to use the lazy API starting from either a file or an existing `DataFrame`.

Using the lazy API from a file

In the ideal case we would use the lazy API right from a file as the query optimizer may help us to reduce the amount of data we read from the file.

We create a lazy query from the Reddit CSV data and apply some transformations.

By starting the query with `pl.scan_csv` we are using the lazy API.

Python

API `scan_csv` · API `with_columns` · API `filter` · API `col`

```
q1 = (
    pl.scan_csv(f"docs/src/data/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
)
```

A `pl.scan_` function is available for a number of file types including CSV, IPC, Parquet and JSON.

In this query we tell Polars that we want to:

- load data from the Reddit CSV file
- convert the `name` column to uppercase
- apply a filter to the `comment_karma` column

The lazy query will not be executed at this point. See this page on executing lazy queries for more on running lazy queries.

Using the lazy API from a `DataFrame`

An alternative way to access the lazy API is to call `.lazy` on a `DataFrame` that has already been created in memory.

Python

API `lazy`

```
q3 = pl.DataFrame({"foo": ["a", "b", "c"], "bar": [0, 1, 2]}).lazy()
```

By calling `.lazy` we convert the `DataFrame` to a `LazyFrame`.

4.6.2 Optimizations

If you use `Polars`' lazy API, `Polars` will run several optimizations on your query. Some of them are executed up front, others are determined just in time as the materialized data comes in.

Here is a non-complete overview of optimizations done by polars, what they do and how often they run.

Optimization	Explanation	runs
Predicate pushdown	Applies filters as early as possible/ at scan level.	1 time
Projection pushdown	Select only the columns that are needed at the scan level.	1 time
Slice pushdown	Only load the required slice from the scan level. Don't materialize sliced outputs (e.g. join.head(10)).	1 time
Common subplan elimination	Cache subtrees/file scans that are used by multiple subtrees in the query plan.	1 time
Simplify expressions	Various optimizations, such as constant folding and replacing expensive operations with faster alternatives.	until fixed point
Join ordering	Estimates the branches of joins that should be executed first in order to reduce memory pressure.	1 time
Type coercion	Coerce types such that operations succeed and run on minimal required memory.	until fixed point
Cardinality estimation	Estimates cardinality in order to determine optimal groupby strategy.	0/n times; dependent on query

4.6.3 Schema

The schema of a Polars `DataFrame` or `LazyFrame` sets out the names of the columns and their datatypes. You can see the schema with the `.schema` method on a `DataFrame` or `LazyFrame`

Python

API `DataFrame` · API `lazy`

```
q3 = pl.DataFrame({"foo": ["a", "b", "c"], "bar": [0, 1, 2]}).lazy()

print(q3.schema)
```

```
{'foo': Utf8, 'bar': Int64}
```

The schema plays an important role in the lazy API.

Type checking in the lazy API

One advantage of the lazy API is that Polars will check the schema before any data is processed. This check happens when you execute your lazy query.

We see how this works in the following simple example where we call the `.round` expression on the integer `bar` column.

Python

API `lazy` · API `with_columns`

```
pl.DataFrame({"foo": ["a", "b", "c"], "bar": [0, 1, 2]}).lazy().with_columns(
    pl.col("bar").round(0)
)
```

The `.round` expression is only valid for columns with a floating point dtype. Calling `.round` on an integer column means the operation will raise a `SchemaError`.

If we executed this query in eager mode the error would only be found once the data had been processed in all earlier steps.

When we execute a lazy query Polars checks for any potential `SchemaError` before the time-consuming step of actually processing the data in the pipeline.

The lazy API must know the schema

In the lazy API the Polars query optimizer must be able to infer the schema at every step of a query plan. This means that operations where the schema is not knowable in advance cannot be used with the lazy API.

The classic example of an operation where the schema is not knowable in advance is a `.pivot` operation. In a `.pivot` the new column names come from data in one of the columns. As these column names cannot be known in advance a `.pivot` is not available in the lazy API.

Dealing with operations not available in the lazy API

If your pipeline includes an operation that is not available in the lazy API it is normally best to:

- run the pipeline in lazy mode up until that point
- execute the pipeline with `.collect` to materialize a `DataFrame`
- do the non-lazy operation on the `DataFrame`
- convert the output back to a `LazyFrame` with `.lazy` and continue in lazy mode

We show how to deal with a non-lazy operation in this example where we:

- create a simple `DataFrame`
- convert it to a `LazyFrame` with `.lazy`
- do a transformation using `.with_columns`
- execute the query before the pivot with `.collect` to get a `DataFrame`
- do the `.pivot` on the `DataFrame`
- convert back in lazy mode
- do a `.filter`
- finish by executing the query with `.collect` to get a `DataFrame`

Python

API `collect` · API `pivot` · API `filter`

```
lazy_eager_query = (
    pl.DataFrame(
        {
            "id": ["a", "b", "c"],
            "month": ["jan", "feb", "mar"],
            "values": [0, 1, 2],
        }
    )
    .lazy()
    .with_columns((2 * pl.col("values")).alias("double_values"))
    .collect()
    .pivot(
        index="id", columns="month", values="double_values", aggregate_function="first"
    )
    .lazy()
    .filter(pl.col("mar").is_null())
    .collect()
)
print(lazy_eager_query)
```

```
shape: (2, 4)
┌─────┬──────┬──────┬──────┐
│ id  ┆ jan  ┆ feb  ┆ mar  │
│ --- ┆ ---  ┆ ---  ┆ ---  │
│ str ┆ i64  ┆ i64  ┆ i64  │
╞═════╪══════╪══════╪══════╡
│ a   ┆ 0    ┆ null ┆ null │
│ b   ┆ null ┆ 2    ┆ null │
└─────┴──────┴──────┴──────┘
```

4.6.4 Query Plan

For any lazy query `Polars` has both:

- a non-optimized plan with the set of steps code as we provided it and
- an optimized plan with changes made by the query optimizer

We can understand both the non-optimized and optimized query plans with visualization and by printing them as text.

Below we consider the following query:

Python

```
q1 = (
    pl.scan_csv(f"docs/src/data/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
)
```

Non-optimized query plan

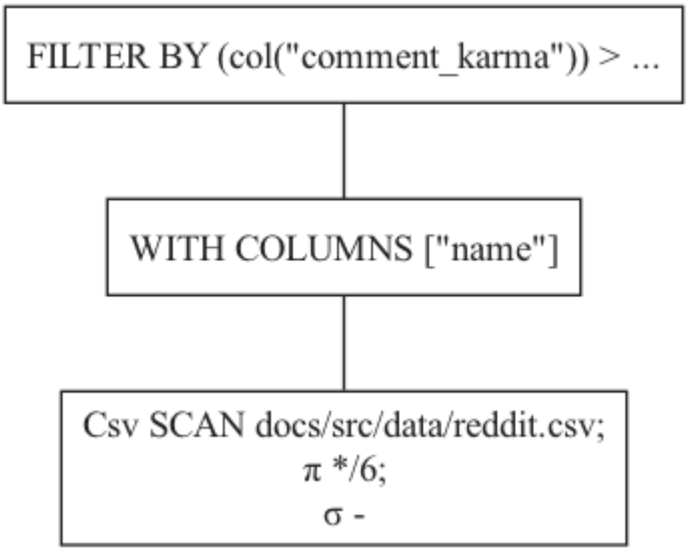
GRAPHVIZ VISUALIZATION

First we visualise the non-optimized plan by setting `optimized=False`.

Python

API `show_graph`

```
q1.show_graph(optimized=False)
```



The query plan visualization should be read from bottom to top. In the visualization:

- each box corresponds to a stage in the query plan
- the `sigma` stands for `SELECTION` and indicates any filter conditions
- the `pi` stands for `PROJECTION` and indicates choosing a subset of columns

PRINTED QUERY PLAN

We can also print the non-optimized plan with `explain(optimized=False)`

Python

API `explain`

```
q1.explain(optimized=False)
```

```
FILTER [(col("comment_karma")) > (0)] FROM WITH_COLUMNS:
 [col("name").str.uppercase()]

    CSV SCAN data/reddit.csv
    PROJECT */6 COLUMNS
```

The printed plan should also be read from bottom to top. This non-optimized plan is roughly equal to:

- read from the `data/reddit.csv` file
- read all 6 columns (where the * wildcard in PROJECT */6 COLUMNS means take all columns)
- transform the `name` column to uppercase
- apply a filter on the `comment_karma` column

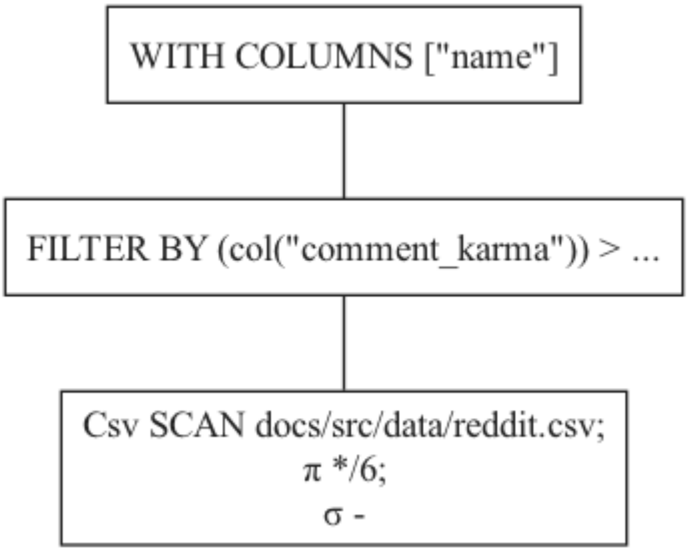
Optimized query plan

Now we visualize the optimized plan with `show_graph`.

Python

API `show_graph`

```
q1.show_graph()
```



We can also print the optimized plan with `explain`

Python

API `explain`

```
q1.explain()
```

```
WITH_COLUMNS:
[col("name").str.uppercase()]

   CSV SCAN data/reddit.csv
   PROJECT */6 COLUMNS
   SELECTION: [(col("comment_karma")) > (0)]
```

The optimized plan is to:

- read the data from the Reddit CSV
- apply the filter on the `comment_karma` column while the CSV is being read line-by-line
- transform the `name` column to uppercase

In this case the query optimizer has identified that the `filter` can be applied while the CSV is read from disk rather than reading the whole file into memory and then applying the filter. This optimization is called *Predicate Pushdown*.

4.6.5 Query execution

Our example query on the Reddit dataset is:

Python

API scan_csv

```
q1 = (
    pl.scan_csv("docs/src/data/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
)
```

If we were to run the code above on the Reddit CSV the query would not be evaluated. Instead Polars takes each line of code, adds it to the internal query graph and optimizes the query graph.

When we execute the code Polars executes the optimized query graph by default.

EXECUTION ON THE FULL DATASET

We can execute our query on the full dataset by calling the `.collect` method on the query.

Python

API scan_csv · API collect

```
q4 = (
    pl.scan_csv(f"docs/data/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
    .collect()
)
```

```
shape: (14_029, 6)
┌─────────┬───────────────────────────┬─────────────┬────────────┬───────────────┬────────────┐
│ id      ┆ name                      ┆ created_utc ┆ updated_on ┆ comment_karma ┆ link_karma │
│ ---     ┆ ---                       ┆ ---         ┆ ---        ┆ ---           ┆ ---        │
│ i64     ┆ str                       ┆ i64         ┆ i64        ┆ i64           ┆ i64        │
╞═════════╪═══════════════════════════╪═════════════╪════════════╪═══════════════╪════════════╡
│ 6       ┆ TAOJIANLONG_JASONBROKEN   ┆ 1397113510  ┆ 1536527864 ┆ 4             ┆ 0          │
│ 17      ┆ SSAIG_JASONBROKEN         ┆ 1397113544  ┆ 1536527864 ┆ 1             ┆ 0          │
│ 19      ┆ FDBVFDSSDGFDS_JASONBROKEN ┆ 1397113552  ┆ 1536527864 ┆ 3             ┆ 0          │
│ 37      ┆ IHATEWHOWEARE_JASONBROKEN ┆ 1397113636  ┆ 1536527864 ┆ 61            ┆ 0          │
│ …       ┆ …                         ┆ …           ┆ …          ┆ …             ┆ …          │
│ 1229384 ┆ DSFOX                     ┆ 1163177415  ┆ 1536497412 ┆ 44411         ┆ 7917       │
│ 1229459 ┆ NEOCARTY                  ┆ 1163177859  ┆ 1536533090 ┆ 40            ┆ 0          │
│ 1229587 ┆ TEHSMA                    ┆ 1163178847  ┆ 1536497412 ┆ 14794         ┆ 5707       │
│ 1229621 ┆ JEREMYLOW                 ┆ 1163179075  ┆ 1536497412 ┆ 411           ┆ 1063       │
└─────────┴───────────────────────────┴─────────────┴────────────┴───────────────┴────────────┘
```

Above we see that from the 10 million rows there are 14,029 rows that match our predicate.

With the default `collect` method Polars processes all of your data as one batch. This means that all the data has to fit into your available memory at the point of peak memory usage in your query.

EXECUTION ON LARGER-THAN-MEMORY DATA

If your data requires more memory than you have available Polars may be able to process the data in batches using *streaming* mode. To use streaming mode you simply pass the `streaming=True` argument to `collect`

Python

API `scan_csv` · API `collect`

```
q5 = (
    pl.scan_csv(f"docs/data/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
    .collect(streaming=True)
)
```

We look at streaming in more detail here.

EXECUTION ON A PARTIAL DATASET

While you're writing, optimizing or checking your query on a large dataset, querying all available data may lead to a slow development process.

You can instead execute the query with the `.fetch` method. The `.fetch` method takes a parameter `n_rows` and tries to 'fetch' that number of rows at the data source. The number of rows cannot be guaranteed, however, as the lazy API does not count how many rows there are at each stage of the query.

Here we "fetch" 100 rows from the source file and apply the predicates.

Python

API `scan_csv` · API `collect` · API `fetch`

```
q9 = (
    pl.scan_csv(f"docs/data/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
    .fetch(n_rows=int(100))
)
```

```
shape: (27, 6)
┌───────┬───────────────────────────┬─────────────┬────────────┬───────────────┬────────────┐
│ id    ┆ name                      ┆ created_utc ┆ updated_on ┆ comment_karma ┆ link_karma │
│ ---   ┆ ---                       ┆ ---         ┆ ---        ┆ ---           ┆ ---        │
│ i64   ┆ str                       ┆ i64         ┆ i64        ┆ i64           ┆ i64        │
╞═══════╪═══════════════════════════╪═════════════╪════════════╪═══════════════╪════════════╡
│ 6     ┆ TAOJIANLONG_JASONBROKEN   ┆ 1397113510  ┆ 1536527864 ┆ 4             ┆ 0          │
│ 17    ┆ SSAIG_JASONBROKEN         ┆ 1397113544  ┆ 1536527864 ┆ 1             ┆ 0          │
│ 19    ┆ FDBVFDSSDGFDS_JASONBROKEN ┆ 1397113552  ┆ 1536527864 ┆ 3             ┆ 0          │
│ 37    ┆ IHATEWHOWEARE_JASONBROKEN ┆ 1397113636  ┆ 1536527864 ┆ 61            ┆ 0          │
│ …     ┆ …                         ┆ …           ┆ …          ┆ …             ┆ …          │
│ 77763 ┆ LUNCHY                    ┆ 1137599510  ┆ 1536528275 ┆ 65            ┆ 0          │
│ 77765 ┆ COMPOSTELLAS              ┆ 1137474000  ┆ 1536528276 ┆ 6             ┆ 0          │
│ 77766 ┆ GENERICBOB                ┆ 1137474000  ┆ 1536528276 ┆ 291           ┆ 14         │
│ 77768 ┆ TINHEADNED                ┆ 1139665457  ┆ 1536497404 ┆ 4434          ┆ 103        │
└───────┴───────────────────────────┴─────────────┴────────────┴───────────────┴────────────┘
```

4.6.6 Streaming

🚧**Under Construction** 🚧

This section is still under development. Want to help out? Consider contributing and making a pull request to our repository. Please read our Contribution Guidelines on how to proceed.

4.7 IO

4.7.1 CSV

Read & Write

Reading a CSV file should look familiar:

Python Rust NodeJS

API read_csv

```
df = pl.read_csv("path.csv")
```

API CsvReader · Available on feature csv

```
use polars::prelude::*;

let df = CsvReader::from_path("path.csv").unwrap().finish().unwrap();
```

API readCSV

```
df = pl.readCSV("path.csv")
```

Writing a CSV file is similar with the `write_csv` function:

Python Rust NodeJS

API write_csv

```
df = pl.DataFrame({"foo": [1, 2, 3], "bar": [None, "bak", "baz"]})
df.write_csv("path.csv")
```

API CsvWriter · Available on feature csv

```
let mut df = df!(
    "foo" => &[1, 2, 3],
    "bar" => &[None, Some("bak"), Some("baz")],
)
.unwrap();

let mut file = std::fs::File::create("path.csv").unwrap();
CsvWriter::new(&mut file).finish(&mut df).unwrap();
```

API writeCSV

```
df = pl.DataFrame({ foo: [1, 2, 3], bar: [null, "bak", "baz"] });
df.writeCSV("path.csv");
```

Scan

`Polars` allows you to *scan* a CSV input. Scanning delays the actual parsing of the file and instead returns a lazy computation holder called a `LazyFrame`.

Python Rust NodeJS

API `scan_csv`

```
df = pl.scan_csv("path.csv")
```

API `LazyCsvReader` · Available on feature csv

```
let df = LazyCsvReader::new("./test.csv").finish().unwrap();
```

API `scanCSV`

```
df = pl.scanCSV("path.csv");
```

If you want to know why this is desirable, you can read more about these `Polars` optimizations here.

4.7.2 Parquet

Loading or writing `Parquet` files is lightning fast. `Pandas` uses `PyArrow` - `Python` bindings exposed by `Arrow` - to load `Parquet` files into memory, but it has to copy that data into `Pandas` memory. With `Polars` there is no extra cost due to copying as we read `Parquet` directly into `Arrow` memory and *keep it there*.

Read

Python Rust NodeJS

API `read_parquet`

```
df = pl.read_parquet("path.parquet")
```

API `ParquetReader` · Available on feature parquet

```
let mut file = std::fs::File::open("path.parquet").unwrap();

let df = ParquetReader::new(&mut file).finish().unwrap();
```

API `readParquet`

```
df = pl.readParquet("path.parquet")
```

Write

Python Rust NodeJS

API `write_parquet`

```
df = pl.DataFrame({"foo": [1, 2, 3], "bar": [None, "bak", "baz"]})
df.write_parquet("path.parquet")
```

API `ParquetWriter` · Available on feature parquet

```
let mut df = df!(
    "foo" => &[1, 2, 3],
    "bar" => &[None, Some("bak"), Some("baz")],
)
.unwrap();

let mut file = std::fs::File::create("path.parquet").unwrap();
ParquetWriter::new(&mut file).finish(&mut df).unwrap();
```

API `writeParquet`

```
df = pl.DataFrame({ foo: [1, 2, 3], bar: [null, "bak", "baz"] });
df.writeParquet("path.parquet");
```

Scan

`Polars` allows you to *scan* a `Parquet` input. Scanning delays the actual parsing of the file and instead returns a lazy computation holder called a `LazyFrame`.

Python Rust NodeJS

API `scan_parquet`

```
df = pl.scan_parquet("path.parquet")
```

API `scan_parquet` · Available on feature parquet

```
let args = ScanArgsParquet::default();
let df = LazyFrame::scan_parquet("./file.parquet",args).unwrap();
```

```
df = pl.scanParquet("path.parquet");
```

If you want to know why this is desirable, you can read more about those `Polars` optimizations here.

4.7.3 JSON files

Read & Write

JSON

Reading a JSON file should look familiar:

Python Rust

API `read_json`

```
df = pl.read_json("path.json")
```

API `JsonReader` · Available on feature json

```
use polars::prelude::*;

let mut file = std::fs::File::open("path.json").unwrap();
let df = JsonReader::new(&mut file).finish().unwrap();
```

NEWLINE DELIMITED JSON

JSON objects that are delimited by newlines can be read into polars in a much more performant way than standard json.

Python Rust

API `read_ndjson`

```
df = pl.read_ndjson("path.json")
```

API `JsonLineReader` · Available on feature json

```
let mut file = std::fs::File::open("path.json").unwrap();
let df = JsonLineReader::new(&mut file).finish().unwrap();
```

Write

Python Rust

API `write_json` · API `write_ndjson`

```python
df = pl.DataFrame({"foo": [1, 2, 3], "bar": [None, "bak", "baz"]})
# json
df.write_json("path.json")
# ndjson
df.write_ndjson("path.json")
```

API `JsonWriter` · API `JsonWriter` · Available on feature json

```rust
let mut df = df!(
    "foo" => &[1, 2, 3],
    "bar" => &[None, Some("bak"), Some("baz")],
)
.unwrap();

let mut file = std::fs::File::create("path.json").unwrap();

// json
JsonWriter::new(&mut file)
    .with_json_format(JsonFormat::Json)
    .finish(&mut df)
    .unwrap();

// ndjson
JsonWriter::new(&mut file)
    .with_json_format(JsonFormat::JsonLines)
    .finish(&mut df)
    .unwrap();
```

Scan

`Polars` allows you to *scan* a JSON input **only for newline delimited json**. Scanning delays the actual parsing of the file and instead returns a lazy computation holder called a `LazyFrame`.

Python Rust

API `scan_ndjson`

```python
df = pl.scan_ndjson("path.json")
```

API `LazyJsonLineReader` · Available on feature json

```rust
let df = LazyJsonLineReader::new("path.json".to_string()).finish().unwrap();
```

4.7.4 Multiple

Dealing with multiple files.

`Polars` can deal with multiple files differently depending on your needs and memory strain.

Let's create some files to give us some context:

Python

API `write_csv`

```
import polars as pl

df = pl.DataFrame({"foo": [1, 2, 3], "bar": [None, "ham", "spam"]})

for i in range(5):
    df.write_csv(f"my_many_files_{i}.csv")
```

Reading into a single `DataFrame`

To read multiple files into a single `DataFrame`, we can use globbing patterns:

Python

API `read_csv`

```
df = pl.read_csv("my_many_files_*.csv")
print(df)
```

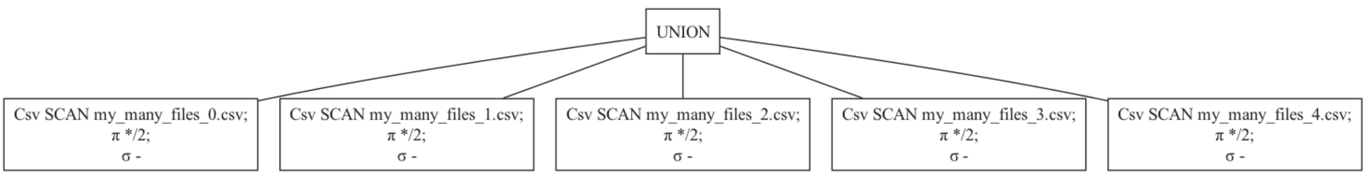
```
shape: (15, 2)
┌─────┬──────┐
│ foo ┆ bar  │
│ --- ┆ ---  │
│ i64 ┆ str  │
╞═════╪══════╡
│ 1   ┆ null │
│ 2   ┆ ham  │
│ 3   ┆ spam │
│ 1   ┆ null │
│ …   ┆ …    │
│ 3   ┆ spam │
│ 1   ┆ null │
│ 2   ┆ ham  │
│ 3   ┆ spam │
└─────┴──────┘
```

To see how this works we can take a look at the query plan. Below we see that all files are read separately and concatenated into a single `DataFrame`. `Polars` will try to parallelize the reading.

Python

API `show_graph`

```
pl.scan_csv("my_many_files_*.csv").show_graph()
```



Reading and processing in parallel

If your files don't have to be in a single table you can also build a query plan for each file and execute them in parallel on the `Polars` thread pool.

All query plan execution is embarrassingly parallel and doesn't require any communication.

Python

API `scan_csv`

```python
import polars as pl
import glob

queries = []
for file in glob.glob("my_many_files_*.csv"):
    q = pl.scan_csv(file).groupby("bar").agg([pl.count(), pl.sum("foo")])
    queries.append(q)

dataframes = pl.collect_all(queries)
print(dataframes)
```

```
[shape: (3, 3)
┌──────┬───────┬─────┐
│ bar  ┆ count ┆ foo │
│ ---  ┆ ---   ┆ --- │
│ str  ┆ u32   ┆ i64 │
╞══════╪═══════╪═════╡
│ null ┆ 1     ┆ 1   │
│ spam ┆ 1     ┆ 3   │
│ ham  ┆ 1     ┆ 2   │
└──────┴───────┴─────┘, shape: (3, 3)
┌──────┬───────┬─────┐
│ bar  ┆ count ┆ foo │
│ ---  ┆ ---   ┆ --- │
│ str  ┆ u32   ┆ i64 │
╞══════╪═══════╪═════╡
│ null ┆ 1     ┆ 1   │
│ ham  ┆ 1     ┆ 2   │
│ spam ┆ 1     ┆ 3   │
└──────┴───────┴─────┘, shape: (3, 3)
┌──────┬───────┬─────┐
│ bar  ┆ count ┆ foo │
│ ---  ┆ ---   ┆ --- │
│ str  ┆ u32   ┆ i64 │
╞══════╪═══════╪═════╡
│ null ┆ 1     ┆ 1   │
│ ham  ┆ 1     ┆ 2   │
│ spam ┆ 1     ┆ 3   │
└──────┴───────┴─────┘, shape: (3, 3)
┌──────┬───────┬─────┐
│ bar  ┆ count ┆ foo │
│ ---  ┆ ---   ┆ --- │
│ str  ┆ u32   ┆ i64 │
╞══════╪═══════╪═════╡
│ null ┆ 1     ┆ 1   │
│ spam ┆ 1     ┆ 3   │
│ ham  ┆ 1     ┆ 2   │
└──────┴───────┴─────┘, shape: (3, 3)
┌──────┬───────┬─────┐
│ bar  ┆ count ┆ foo │
│ ---  ┆ ---   ┆ --- │
│ str  ┆ u32   ┆ i64 │
╞══════╪═══════╪═════╡
│ ham  ┆ 1     ┆ 2   │
│ spam ┆ 1     ┆ 3   │
│ null ┆ 1     ┆ 1   │
└──────┴───────┴─────┘]
```

4.7.5 Databases

Read from a database

We can read from a database with Polars using the `pl.read_database` function. To use this function you need an SQL query string and a connection string called a `connection_uri`.

For example, the following snippet shows the general patterns for reading all columns from the `foo` table in a Postgres database:

Python

API `read_database` · Available on feature connectorx

```
import polars as pl

connection_uri = "postgres://username:password@server:port/database"
query = "SELECT * FROM foo"

pl.read_database(query=query, connection_uri=connection_uri)
```

ENGINES

Polars doesn't manage connections and data transfer from databases by itself. Instead external libraries (known as *engines*) handle this. At present Polars can use two engines to read from databases:

- ConnectorX and
- ADBC

ConnectorX

ConnectorX is the default engine and supports numerous databases including Postgres, Mysql, SQL Server and Redshift. ConnectorX is written in Rust and stores data in Arrow format to allow for zero-copy to Polars.

To read from one of the supported databases with `ConnectorX` you need to activate the additional dependancy `ConnectorX` when installing Polars or install it manually with

```
$ pip install connectorx
```

ADBC

ADBC (Arrow Database Connectivity) is an engine supported by the Apache Arrow project. ADBC aims to be both an API standard for connecting to databases and libraries implementing this standard in a range of languages.

It is still early days for ADBC so support for different databases is still limited. At present drivers for ADBC are only available for Postgres and SQLite. To install ADBC you need to install the driver for your database. For example to install the driver for SQLite you run

```
$ pip install adbc-driver-sqlite
```

As ADBC is not the default engine you must specify the engine as an argument to `pl.read_database`

Python

API `read_database`

```
connection_uri = "postgres://username:password@server:port/database"
query = "SELECT * FROM foo"

pl.read_database(query=query, connection_uri=connection_uri, engine="adbc")
```

Write to a database

We can write to a database with Polars using the `pl.write_database` function.

ENGINES

As with reading from a database above Polars uses an *engine* to write to a database. The currently supported engines are:

- SQLAlchemy and
- Arrow Database Connectivity (ADBC)

SQLAlchemy

With the default engine SQLAlchemy you can write to any database supported by SQLAlchemy. To use this engine you need to install SQLAlchemy and Pandas

```
$ pip install SQLAlchemy pandas
```

In this example, we write the `DataFrame` to a table called `records` in the database

Python

API `write_database`

```
connection_uri = "postgres://username:password@server:port/database"
df = pl.DataFrame({"foo": [1, 2, 3]})

df.write_database(table_name="records",  connection_uri=connection_uri)
```

In the SQLAlchemy approach Polars converts the `DataFrame` to a Pandas `DataFrame` backed by PyArrow and then uses SQLAlchemy methods on a Pandas `DataFrame` to write to the database.

ADBC

As with reading from a database you can also use ADBC to write to a SQLite or Posgres database. As shown above you need to install the appropriate ADBC driver for your database.

Python

API `write_database`

```
connection_uri = "postgres://username:password@server:port/database"
df = pl.DataFrame({"foo": [1, 2, 3]})

df.write_database(table_name="records", connection_uri=connection_uri, engine="adbc")
```

4.7.6 AWS

Under Construction

This section is still under development. Want to help out? Consider contributing and making a pull request to our repository. Please read our Contribution Guidelines on how to proceed.

To read from or write to an AWS bucket, additional dependencies are needed in Rust:

Rust

```
$ cargo add aws_sdk_s3 aws_config tokio --features tokio/full
```

In the next few snippets we'll demonstrate interacting with a `Parquet` file located on an AWS bucket.

Read

Load a `.parquet` file using:

Python Rust

API `from_arrow` · Available on feature fsspec · Available on feature pyarrow

```python
import polars as pl
import pyarrow.parquet as pq
import s3fs

fs = s3fs.S3FileSystem()
bucket = "<YOUR_BUCKET>"
path = "<YOUR_PATH>"

dataset = pq.ParquetDataset(f"s3://{bucket}/{path}", filesystem=fs)
df = pl.from_arrow(dataset.read())
```

```rust
use aws_sdk_s3::Region;

use aws_config::meta::region::RegionProviderChain;
use aws_sdk_s3::Client;
use std::borrow::Cow;

use polars::prelude::*;

#[tokio::main]
async fn main() {
    let bucket = "<YOUR_BUCKET>";
    let path = "<YOUR_PATH>";

    let config = aws_config::from_env().load().await;
    let client = Client::new(&config);

    let req = client.get_object().bucket(bucket).key(path);

    let res = req.clone().send().await.unwrap();
    let bytes = res.body.collect().await.unwrap();
    let bytes = bytes.into_bytes();

    let cursor = std::io::Cursor::new(bytes);

    let df = CsvReader::new(cursor).finish().unwrap();

    println!("{:?}", df);
}
```

4.7.7 Google BigQuery

To read or write from GBQ, additional dependencies are needed:

Python

```
$ pip install google-cloud-bigquery
```

Read

We can load a query into a `DataFrame` like this:

Python

API `from_arrow` · Available on feature fsspec · Available on feature pyarrow

```
import polars as pl
from google.cloud import bigquery

client = bigquery.Client()

# Perform a query.
QUERY = (
    'SELECT name FROM `bigquery-public-data.usa_names.usa_1910_2013` '
    'WHERE state = "TX" '
    'LIMIT 100')
query_job = client.query(QUERY)  # API request
rows = query_job.result()  # Waits for query to finish

df = pl.from_arrow(rows.to_arrow())
```

Write

Under Construction

This section is still under development. Want to help out? Consider contributing and making a pull request to our repository. Please read our Contribution Guidelines on how to proceed.

4.8 SQL

4.8.1 Introduction

While Polars does support writing queries in SQL, it's recommended that users familiarize themselves with the expression syntax for more readable and expressive code. As a primarily DataFrame library, new features will typically be added to the expression API first. However, if you already have an existing SQL codebase or prefer to use SQL, Polars also offers support for SQL queries.

> **Note**
>
> In Polars, there is no separate SQL engine because Polars translates SQL queries into expressions, which are then executed using its built-in execution engine. This approach ensures that Polars maintains its performance and scalability advantages as a native DataFrame library while still providing users with the ability to work with SQL queries.

Context

Polars uses the `SQLContext` to manage SQL queries . The context contains a dictionary mapping `DataFrames` and `LazyFrames` names to their corresponding datasets[1]. The example below starts a `SQLContext` :

Python

API `SQLContext`

```
ctx = pl.SQLContext()
```

Register Dataframes

There are 2 ways to register DataFrames in the `SQLContext` :

- register all `LazyFrames` and `DataFrames` in the global namespace
- register them one by one

Python

API `SQLContext`

```
df = pl.DataFrame({"a": [1, 2, 3]})
lf = pl.LazyFrame({"b": [4, 5, 6]})

# Register all dataframes in the global namespace: registers both df and lf
ctx = pl.SQLContext(register_globals=True)

# Other option: register dataframe df as "df" and lazyframe lf as "lf"
ctx = pl.SQLContext(df=df, lf=lf)
```

We can also register Pandas DataFrames by converting them to Polars first.

Python

API `SQLContext`

```
import pandas as pd

df_pandas = pd.DataFrame({"c": [7, 8, 9]})
ctx = pl.SQLContext(df_pandas=pl.from_pandas(df_pandas))
```

> **Note**
>
> Converting a Pandas DataFrame backed by Numpy to Polars triggers a conversion to the Arrow format. This conversion has a computation cost. Converting a Pandas DataFrame backed by Arrow on the other hand will be free or almost free.

Once the `SQLContext` is initialized, we can register additional Dataframes or unregister existing Dataframes with:

- `register`
- `register_globals`
- `register_many`
- `unregister`

Execute queries and collect results

SQL queries are always executed in lazy mode to benefit from lazy optimizations, so we have 2 options to collect the result:

- Set the parameter `eager_execution` to True in `SQLContext`. With this parameter, Polars will automatically collect SQL results
- Set the parameter `eager` to True when executing a query with `execute`, or collect the result with `collect`.

We execute SQL queries by calling `execute` on a `SQLContext`.

Python

API `register` · API `execute`

```python
# For local files use scan_csv instead
pokemon = pl.read_csv(
    "https://gist.githubusercontent.com/ritchie46/cac6b337ea52281aa23c049250a4ff03/raw/89a957ff3919d90e6ef2d34235e6bf22304f3366/pokemon.csv"
)
ctx = pl.SQLContext(register_globals=True, eager_execution=True)
df_small = ctx.execute("SELECT * from pokemon LIMIT 5")
print(df_small)
```

```
shape: (5, 13)
┌─────┬───────────────────────┬────────┬────────┬───┬─────────┬───────┬────────────┬───────────┐
│ #   ┆ Name                  ┆ Type 1 ┆ Type 2 ┆ … ┆ Sp. Def ┆ Speed ┆ Generation ┆ Legendary │
│ --- ┆ ---                   ┆ ---    ┆ ---    ┆   ┆ ---     ┆ ---   ┆ ---        ┆ ---       │
│ i64 ┆ str                   ┆ str    ┆ str    ┆   ┆ i64     ┆ i64   ┆ i64        ┆ bool      │
╞═════╪═══════════════════════╪════════╪════════╪═══╪═════════╪═══════╪════════════╪═══════════╡
│ 1   ┆ Bulbasaur             ┆ Grass  ┆ Poison ┆ … ┆ 65      ┆ 45    ┆ 1          ┆ false     │
│ 2   ┆ Ivysaur               ┆ Grass  ┆ Poison ┆ … ┆ 80      ┆ 60    ┆ 1          ┆ false     │
│ 3   ┆ Venusaur              ┆ Grass  ┆ Poison ┆ … ┆ 100     ┆ 80    ┆ 1          ┆ false     │
│ 3   ┆ VenusaurMega Venusaur ┆ Grass  ┆ Poison ┆ … ┆ 120     ┆ 80    ┆ 1          ┆ false     │
│ 4   ┆ Charmander            ┆ Fire   ┆ null   ┆ … ┆ 50      ┆ 65    ┆ 1          ┆ false     │
└─────┴───────────────────────┴────────┴────────┴───┴─────────┴───────┴────────────┴───────────┘
```

Execute queries from multiple sources

SQL queries can be executed just as easily from multiple sources. In the example below, we register :

- a CSV file loaded lazily
- a NDJSON file loaded lazily
- a Pandas DataFrame

And we join them together with SQL. Lazy reading allows to only load the necessary rows and columns from the files.

In the same way, it's possible to register cloud datalakes (S3, Azure Data Lake). A PyArrow dataset can point to the datalake, then Polars can read it with `scan_pyarrow_dataset`.

Python

API `register` · API `execute`

```python
# Input data:
# products_masterdata.csv with schema {'product_id': Int64, 'product_name': Utf8}
# products_categories.json with schema {'product_id': Int64, 'category': Utf8}
# sales_data is a Pandas DataFrame with schema {'product_id': Int64, 'sales': Int64}

ctx = pl.SQLContext(
    products_masterdata=pl.scan_csv("products_masterdata.csv"),
    products_categories=pl.scan_ndjson("products_categories.json"),
    sales_data=pl.from_pandas(sales_data),
    eager_execution=True,
)

query = """
SELECT
    product_id,
    product_name,
    category,
    sales
FROM
    products_masterdata
LEFT JOIN products_categories USING (product_id)
LEFT JOIN sales_data USING (product_id)
"""

print(ctx.execute(query))
```

```
shape: (5, 4)
┌────────────┬──────────────┬────────────┬───────┐
│ product_id ┆ product_name ┆ category   ┆ sales │
│ ---        ┆ ---          ┆ ---        ┆ ---   │
│ i64        ┆ str          ┆ str        ┆ i64   │
╞════════════╪══════════════╪════════════╪═══════╡
│ 1          ┆ Product A    ┆ Category 1 ┆ 100   │
│ 2          ┆ Product B    ┆ Category 1 ┆ 200   │
│ 3          ┆ Product C    ┆ Category 2 ┆ 150   │
│ 4          ┆ Product D    ┆ Category 2 ┆ 250   │
│ 5          ┆ Product E    ┆ Category 3 ┆ 300   │
└────────────┴──────────────┴────────────┴───────┘
```

Compatibility

Polars does not support the full SQL language, in Polars you are allowed to:

- Write a `CREATE` statements `CREATE TABLE xxx AS ...`
- Write a `SELECT` statements with all generic elements (`GROUP BY`, `WHERE`, `ORDER`, `LIMIT`, `JOIN`, ...)
- Write Common Table Expressions (CTE's) (`WITH tablename AS`)
- Show an overview of all tables `SHOW TABLES`

The following is not yet supported:

- `INSERT`, `UPDATE` or `DELETE` statements
- Table aliasing (e.g. `SELECT p.Name from pokemon AS p`)
- Meta queries such as `ANALYZE`, `EXPLAIN`

In the upcoming sections we will cover each of the statements in more details.

---

1. Additionally it also tracks the common table expressions as well. ↩

4.8.2 SHOW TABLES

In Polars, the `SHOW TABLES` statement is used to list all the tables that have been registered in the current `SQLContext`. When you register a DataFrame with the `SQLContext`, you give it a name that can be used to refer to the DataFrame in subsequent SQL statements. The `SHOW TABLES` statement allows you to see a list of all the registered tables, along with their names.

The syntax for the `SHOW TABLES` statement in Polars is as follows:

```
SHOW TABLES
```

Here's an example of how to use the `SHOW TABLES` statement in Polars:

Python

API `register` · API `execute`

```python
# Create some DataFrames and register them with the SQLContext
df1 = pl.LazyFrame(
    {
        "name": ["Alice", "Bob", "Charlie", "David"],
        "age": [25, 30, 35, 40],
    }
)
df2 = pl.LazyFrame(
    {
        "name": ["Ellen", "Frank", "Gina", "Henry"],
        "age": [45, 50, 55, 60],
    }
)
ctx = pl.SQLContext(mytable1=df1, mytable2=df2)

tables = ctx.execute("SHOW TABLES", eager=True)

print(tables)
```

```
shape: (2, 1)
┌──────────┐
│ name     │
│ ---      │
│ str      │
╞══════════╡
│ mytable1 │
│ mytable2 │
└──────────┘
```

In this example, we create two DataFrames and register them with the `SQLContext` using different names. We then execute a `SHOW TABLES` statement using the `execute()` method of the `SQLContext` object, which returns a DataFrame containing a list of all the registered tables and their names. The resulting DataFrame is then printed using the `print()` function.

Note that the `SHOW TABLES` statement only lists tables that have been registered with the current `SQLContext`. If you register a DataFrame with a different `SQLContext` or in a different Python session, it will not appear in the list of tables returned by `SHOW TABLES`.

4.8.3 SELECT

In Polars SQL, the `SELECT` statement is used to retrieve data from a table into a `DataFrame`. The basic syntax of a `SELECT` statement in Polars SQL is as follows:

```
SELECT column1, column2, ...
FROM table_name;
```

Here, `column1`, `column2`, etc. are the columns that you want to select from the table. You can also use the wildcard `*` to select all columns. `table_name` is the name of the table or that you want to retrieve data from. In the sections below we will cover some of the more common SELECT variants

Python

API `register` · API `execute`

```
df = pl.DataFrame(
    {
        "city": [
            "New York",
            "Los Angeles",
            "Chicago",
            "Houston",
            "Phoenix",
            "Amsterdam",
        ],
        "country": ["USA", "USA", "USA", "USA", "USA", "Netherlands"],
        "population": [8399000, 3997000, 2705000, 2320000, 1680000, 900000],
    }
)

ctx = pl.SQLContext(population=df, eager_execution=True)

print(ctx.execute("SELECT * FROM population"))
```

```
shape: (6, 3)
┌─────────────┬─────────────┬────────────┐
│ city        ┆ country     ┆ population │
│ ---         ┆ ---         ┆ ---        │
│ str         ┆ str         ┆ i64        │
╞═════════════╪═════════════╪════════════╡
│ New York    ┆ USA         ┆ 8399000    │
│ Los Angeles ┆ USA         ┆ 3997000    │
│ Chicago     ┆ USA         ┆ 2705000    │
│ Houston     ┆ USA         ┆ 2320000    │
│ Phoenix     ┆ USA         ┆ 1680000    │
│ Amsterdam   ┆ Netherlands ┆ 900000     │
└─────────────┴─────────────┴────────────┘
```

GROUP BY

The `GROUP BY` statement is used to group rows in a table by one or more columns and compute aggregate functions on each group.

Python

API `execute`

```
result = ctx.execute(
    """
        SELECT country, AVG(population) as avg_population
        FROM population
        GROUP BY country
    """
)
print(result)
```

```
shape: (2, 2)
┌─────────────┬────────────────┐
│ country     ┆ avg_population │
│ ---         ┆ ---            │
│ str         ┆ f64            │
╞═════════════╪════════════════╡
│ Netherlands ┆ 900000.0       │
│ USA         ┆ 3.8202e6       │
└─────────────┴────────────────┘
```

ORDER BY

The `ORDER BY` statement is used to sort the result set of a query by one or more columns in ascending or descending order.

Python

API `execute`

```python
result = ctx.execute(
    """
        SELECT city, population
        FROM population
        ORDER BY population
    """
)
print(result)
```

```
shape: (6, 2)
┌─────────────┬────────────┐
│ city        ┆ population │
│ ---         ┆ ---        │
│ str         ┆ i64        │
╞═════════════╪════════════╡
│ Amsterdam   ┆ 900000     │
│ Phoenix     ┆ 1680000    │
│ Houston     ┆ 2320000    │
│ Chicago     ┆ 2705000    │
│ Los Angeles ┆ 3997000    │
│ New York    ┆ 8399000    │
└─────────────┴────────────┘
```

JOIN

Python

API `register_many` · API `execute`

```python
income = pl.DataFrame(
    {
        "city": [
            "New York",
            "Los Angeles",
            "Chicago",
            "Houston",
            "Amsterdam",
            "Rotterdam",
            "Utrecht",
        ],
        "country": [
            "USA",
            "USA",
            "USA",
            "USA",
            "Netherlands",
            "Netherlands",
            "Netherlands",
        ],
        "income": [55000, 62000, 48000, 52000, 42000, 38000, 41000],
    }
)
ctx.register_many(income=income)
result = ctx.execute(
    """
        SELECT country, city, income, population
        FROM population
        LEFT JOIN income on population.city = income.city
    """
)
print(result)
```

```
shape: (6, 4)
┌─────────────┬─────────────┬────────┬────────────┐
│ country     ┆ city        ┆ income ┆ population │
│ ---         ┆ ---         ┆ ---    ┆ ---        │
│ str         ┆ str         ┆ i64    ┆ i64        │
╞═════════════╪═════════════╪════════╪════════════╡
│ USA         ┆ New York    ┆ 55000  ┆ 8399000    │
│ USA         ┆ Los Angeles ┆ 62000  ┆ 3997000    │
│ USA         ┆ Chicago     ┆ 48000  ┆ 2705000    │
│ USA         ┆ Houston     ┆ 52000  ┆ 2320000    │
│ USA         ┆ Phoenix     ┆ null   ┆ 1680000    │
```

```
│ Netherlands ┆ Amsterdam   ┆ 42000 ┆ 900000    │
└─────────────┴─────────────┴───────┴───────────┘
```

FUNCTIONS

Polars provides a wide range of SQL functions, including:

- Mathematical functions: `ABS`, `EXP`, `LOG`, `ASIN`, `ACOS`, `ATAN`, etc.
- String functions: `LOWER`, `UPPER`, `LTRIM`, `RTRIM`, `STARTS_WITH`, `ENDS_WITH`.
- Aggregation functions: `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`, `STDDEV`, `FIRST` etc.
- Array functions: `EXPLODE`, `UNNEST`, `ARRAY_SUM`, `ARRAY_REVERSE`, etc.

For a full list of supported functions go the API documentation. The example below demonstrates how to use a function in a query

Python

API `query`

```python
result = ctx.execute(
    """
        SELECT city, population
        FROM population
        WHERE STARTS_WITH(country,'U')
    """
)
print(result)
```

```
shape: (5, 2)
┌─────────────┬────────────┐
│ city        ┆ population │
│ ---         ┆ ---        │
│ str         ┆ i64        │
╞═════════════╪════════════╡
│ New York    ┆ 8399000    │
│ Los Angeles ┆ 3997000    │
│ Chicago     ┆ 2705000    │
│ Houston     ┆ 2320000    │
│ Phoenix     ┆ 1680000    │
└─────────────┴────────────┘
```

TABLE FUNCTIONS

In the examples earlier we first generated a DataFrame which we registered in the `SQLContext`. Polars also support directly reading from CSV, Parquet, JSON and IPC in your SQL query using table functions `read_xxx`.

Python

API `execute`

```python
result = ctx.execute(
    """
        SELECT *
        FROM read_csv('docs/src/data/iris.csv')
    """
)
print(result)
```

```
shape: (150, 5)
┌──────────────┬─────────────┬──────────────┬─────────────┬───────────┐
│ sepal_length ┆ sepal_width ┆ petal_length ┆ petal_width ┆ species   │
│ ---          ┆ ---         ┆ ---          ┆ ---         ┆ ---       │
│ f64          ┆ f64         ┆ f64          ┆ f64         ┆ str       │
╞══════════════╪═════════════╪══════════════╪═════════════╪═══════════╡
│ 5.1          ┆ 3.5         ┆ 1.4          ┆ 0.2         ┆ Setosa    │
│ 4.9          ┆ 3.0         ┆ 1.4          ┆ 0.2         ┆ Setosa    │
│ 4.7          ┆ 3.2         ┆ 1.3          ┆ 0.2         ┆ Setosa    │
│ 4.6          ┆ 3.1         ┆ 1.5          ┆ 0.2         ┆ Setosa    │
│ …            ┆ …           ┆ …            ┆ …           ┆ …         │
│ 6.3          ┆ 2.5         ┆ 5.0          ┆ 1.9         ┆ Virginica │
│ 6.5          ┆ 3.0         ┆ 5.2          ┆ 2.0         ┆ Virginica │
│ 6.2          ┆ 3.4         ┆ 5.4          ┆ 2.3         ┆ Virginica │
│ 5.9          ┆ 3.0         ┆ 5.1          ┆ 1.8         ┆ Virginica │
└──────────────┴─────────────┴──────────────┴─────────────┴───────────┘
```

4.8.4 CREATE

In Polars, the `SQLContext` provides a way to execute SQL statements against `LazyFrames` and `DataFrames` using SQL syntax. One of the SQL statements that can be executed using `SQLContext` is the `CREATE TABLE` statement, which is used to create a new table.

The syntax for the `CREATE TABLE` statement in Polars is as follows:

```
CREATE TABLE table_name
AS
SELECT ...
```

In this syntax, `table_name` is the name of the new table that will be created, and `SELECT ...` is a SELECT statement that defines the data that will be inserted into the table.

Here's an example of how to use the `CREATE TABLE` statement in Polars:

Python

API `register` · API `execute`

```python
data = {"name": ["Alice", "Bob", "Charlie", "David"], "age": [25, 30, 35, 40]}
df = pl.LazyFrame(data)

ctx = pl.SQLContext(my_table=df, eager_execution=True)

result = ctx.execute(
    """
    CREATE TABLE older_people
    AS
    SELECT * FROM my_table WHERE age > 30
"""
)

print(ctx.execute("SELECT * FROM older_people"))
```

```
shape: (2, 2)
┌─────────┬─────┐
│ name    ┆ age │
│ ---     ┆ --- │
│ str     ┆ i64 │
╞═════════╪═════╡
│ Charlie ┆ 35  │
│ David   ┆ 40  │
└─────────┴─────┘
```

In this example, we use the `execute()` method of the `SQLContext` to execute a `CREATE TABLE` statement that creates a new table called `older_people` based on a SELECT statement that selects all rows from the `my_table` DataFrame where the `age` column is greater than 30.

Note

Note that the result of a `CREATE TABLE` statement is not the table itself. The table is registered in the `SQLContext`. In case you want to turn the table back to a `DataFrame` you can use a `SELECT * FROM ...` statement

4.8.5 Common Table Expressions

Common Table Expressions (CTEs) are a feature of SQL that allow you to define a temporary named result set that can be referenced within a SQL statement. CTEs provide a way to break down complex SQL queries into smaller, more manageable pieces, making them easier to read, write, and maintain.

A CTE is defined using the `WITH` keyword followed by a comma-separated list of subqueries, each of which defines a named result set that can be used in subsequent queries. The syntax for a CTE is as follows:

```
WITH cte_name AS (
    subquery
)
SELECT ...
```

In this syntax, `cte_name` is the name of the CTE, and `subquery` is the subquery that defines the result set. The CTE can then be referenced in subsequent queries as if it were a table or view.

CTEs are particularly useful when working with complex queries that involve multiple levels of subqueries, as they allow you to break down the query into smaller, more manageable pieces that are easier to understand and debug. Additionally, CTEs can help improve query performance by allowing the database to optimize and cache the results of subqueries, reducing the number of times they need to be executed.

Polars supports Common Table Expressions (CTEs) using the WITH clause in SQL syntax. Below is an example

Python

API `register` · API `execute`

```
ctx = pl.SQLContext()
df = pl.LazyFrame(
    {"name": ["Alice", "Bob", "Charlie", "David"], "age": [25, 30, 35, 40]}
)
ctx.register("my_table", df)

result = ctx.execute(
    """
    WITH older_people AS (
        SELECT * FROM my_table WHERE age > 30
    )
    SELECT * FROM older_people WHERE STARTS_WITH(name,'C')
""",
    eager=True,
)

print(result)
```

```
shape: (1, 2)
┌─────────┬─────┐
│ name    ┆ age │
│ ---     ┆ --- │
│ str     ┆ i64 │
╞═════════╪═════╡
│ Charlie ┆ 35  │
└─────────┴─────┘
```

In this example, we use the `execute()` method of the `SQLContext` to execute a SQL query that includes a CTE. The CTE selects all rows from the `my_table` LazyFrame where the `age` column is greater than 30 and gives it the alias `older_people`. We then execute a second SQL query that selects all rows from the `older_people` CTE where the `name` column starts with the letter 'C'.

4.9 Migrating

4.9.1 Coming from Pandas

Here we set out the key points that anyone who has experience with `Pandas` and wants to try `Polars` should know. We include both differences in the concepts the libraries are built on and differences in how you should write `Polars` code compared to `Pandas` code.

Differences in concepts between `Polars` and `Pandas`

`POLARS` DOES NOT HAVE A MULTI-INDEX/INDEX

`Pandas` gives a label to each row with an index. `Polars` does not use an index and each row is indexed by its integer position in the table.

Polars aims to have predictable results and readable queries, as such we think an index does not help us reach that objective. We believe the semantics of a query should not change by the state of an index or a `reset_index` call.

In Polars a DataFrame will always be a 2D table with heterogeneous data-types. The data-types may have nesting, but the table itself will not. Operations like resampling will be done by specialized functions or methods that act like 'verbs' on a table explicitly stating the columns that that 'verb' operates on. As such, it is our conviction that not having indices make things simpler, more explicit, more readable and less error-prone.

Note that an 'index' data structure as known in databases will be used by polars as an optimization technique.

`POLARS` USES APACHE ARROW ARRAYS TO REPRESENT DATA IN MEMORY WHILE `PANDAS` USES `NUMPY` ARRAYS

`Polars` represents data in memory with Arrow arrays while `Pandas` represents data in memory with `Numpy` arrays. Apache Arrow is an emerging standard for in-memory columnar analytics that can accelerate data load times, reduce memory usage and accelerate calculations.

`Polars` can convert data to `Numpy` format with the `to_numpy` method.

`POLARS` HAS MORE SUPPORT FOR PARALLEL OPERATIONS THAN `PANDAS`

`Polars` exploits the strong support for concurrency in Rust to run many operations in parallel. While some operations in `Pandas` are multi-threaded the core of the library is single-threaded and an additional library such as `Dask` must be used to parallelize operations.

`POLARS` CAN LAZILY EVALUATE QUERIES AND APPLY QUERY OPTIMIZATION

Eager evaluation is when code is evaluated as soon as you run the code. Lazy evaluation is when running a line of code means that the underlying logic is added to a query plan rather than being evaluated.

`Polars` supports eager evaluation and lazy evaluation whereas `Pandas` only supports eager evaluation. The lazy evaluation mode is powerful because `Polars` carries out automatic query optimization when it examines the query plan and looks for ways to accelerate the query or reduce memory usage.

`Dask` also supports lazy evaluation when it generates a query plan. However, `Dask` does not carry out query optimization on the query plan.

Key syntax differences

Users coming from `Pandas` generally need to know one thing...

```
polars != pandas
```

If your `Polars` code looks like it could be `Pandas` code, it might run, but it likely runs slower than it should.

Let's go through some typical `Pandas` code and see how we might rewrite it in `Polars`.

SELECTING DATA

As there is no index in `Polars` there is no `.loc` or `iloc` method in `Polars` - and there is also no `SettingWithCopyWarning` in `Polars`.

However, the best way to select data in `Polars` is to use the expression API. For example, if you want to select a column in `Pandas` you can do one of the following:

```
df['a']
df.loc[:,'a']
```

but in `Polars` you would use the `.select` method:

```
df.select('a')
```

If you want to select rows based on the values then in `Polars` you use the `.filter` method:

```
df.filter(pl.col('a') < 10)
```

As noted in the section on expressions below, `Polars` can run operations in `.select` and `filter` in parallel and `Polars` can carry out query optimization on the full set of data selection criteria.

BE LAZY

Working in lazy evaluation mode is straightforward and should be your default in `Polars` as the lazy mode allows `Polars` to do query optimization.

We can run in lazy mode by either using an implicitly lazy function (such as `scan_csv`) or explicitly using the `lazy` method.

Take the following simple example where we read a CSV file from disk and do a groupby. The CSV file has numerous columns but we just want to do a groupby on one of the id columns (`id1`) and then sum by a value column (`v1`). In `Pandas` this would be:

```
    df = pd.read_csv(csv_file, usecols=['id1','v1'])
    grouped_df = df.loc[:,['id1','v1']].groupby('id1').sum('v1')
```

In `Polars` you can build this query in lazy mode with query optimization and evaluate it by replacing the eager `Pandas` function `read_csv` with the implicitly lazy `Polars` function `scan_csv`:

```
    df = pl.scan_csv(csv_file)
    grouped_df = df.groupby('id1').agg(pl.col('v1').sum()).collect()
```

`Polars` optimizes this query by identifying that only the `id1` and `v1` columns are relevant and so will only read these columns from the CSV. By calling the `.collect` method at the end of the second line we instruct `Polars` to eagerly evaluate the query.

If you do want to run this query in eager mode you can just replace `scan_csv` with `read_csv` in the `Polars` code.

Read more about working with lazy evaluation in the lazy API section.

EXPRESS YOURSELF

A typical `Pandas` script consists of multiple data transformations that are executed sequentially. However, in `Polars` these transformations can be executed in parallel using expressions.

Column assignment

We have a dataframe `df` with a column called `value`. We want to add two new columns, a column called `tenXValue` where the `value` column is multiplied by 10 and a column called `hundredXValue` where the `value` column is multiplied by 100.

In `Pandas` this would be:

```
df["tenXValue"] = df["value"] * 10
df["hundredXValue"] = df["value"] * 100
```

These column assignments are executed sequentially.

In `Polars` we add columns to `df` using the `.with_columns` method and name them with the `.alias` method:

```
df.with_columns(
    (pl.col("value") * 10).alias("tenXValue"),
    (pl.col("value") * 100).alias("hundredXValue"),
)
```

These column assignments are executed in parallel.

Column assignment based on predicate

In this case we have a dataframe `df` with columns `a`, `b` and `c`. We want to re-assign the values in column `a` based on a condition. When the value in column `c` is equal to 2 then we replace the value in `a` with the value in `b`.

In `Pandas` this would be:

```
df.loc[df["c"] == 2, "a"] = df.loc[df["c"] == 2, "b"]
```

while in `Polars` this would be:

```
df.with_columns(
    pl.when(pl.col("c") == 2)
    .then(pl.col("b"))
    .otherwise(pl.col("a")).alias("a")
)
```

The `Polars` way is pure in that the original `DataFrame` is not modified. The `mask` is also not computed twice as in `Pandas` (you could prevent this in `Pandas`, but that would require setting a temporary variable).

Additionally `Polars` can compute every branch of an `if -> then -> otherwise` in parallel. This is valuable, when the branches get more expensive to compute.

Filtering

We want to filter the dataframe `df` with housing data based on some criteria.

In `Pandas` you filter the dataframe by passing Boolean expressions to the `loc` method:

```
df.loc[(df['sqft_living'] > 2500) & (df['price'] < 300000)]
```

while in `Polars` you call the `filter` method:

```
df.filter(
    (pl.col("m2_living") > 2500) & (pl.col("price") < 300000)
)
```

The query optimizer in `Polars` can also detect if you write multiple filters separately and combine them into a single filter in the optimized plan.

`Pandas` transform

The `Pandas` documentation demonstrates an operation on a groupby called `transform`. In this case we have a dataframe `df` and we want a new column showing the number of rows in each group.

In `Pandas` we have:

```
df = pd.DataFrame({
    "type": ["m", "n", "o", "m", "m", "n", "n"],
    "c": [1, 1, 1, 2, 2, 2, 2],
})

df["size"] = df.groupby("c")["type"].transform(len)
```

Here `Pandas` does a groupby on `"c"`, takes column `"type"`, computes the group length and then joins the result back to the original `DataFrame` producing:

```
   c type size
0  1    m    3
1  1    n    3
2  1    o    3
3  2    m    4
4  2    m    4
5  2    n    4
6  2    n    4
```

In `Polars` the same can be achieved with `window` functions:

```
df.select(
    pl.all(),
    pl.col("type").count().over("c").alias("size")
)
```

```
shape: (7, 3)
┌─────┬──────┬──────┐
│ c   ┆ type ┆ size │
│ --- ┆ ---  ┆ ---  │
│ i64 ┆ str  ┆ u32  │
╞═════╪══════╪══════╡
│ 1   ┆ m    ┆ 3    │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┤
│ 1   ┆ n    ┆ 3    │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┤
│ 1   ┆ o    ┆ 3    │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┤
│ 2   ┆ m    ┆ 4    │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┤
│ 2   ┆ m    ┆ 4    │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┤
│ 2   ┆ n    ┆ 4    │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┤
│ 2   ┆ n    ┆ 4    │
└─────┴──────┴──────┘
```

Because we can store the whole operation in a single expression, we can combine several `window` functions and even combine different groups!

`Polars` will cache window expressions that are applied over the same group, so storing them in a single `select` is both convenient **and** optimal. In the following example we look at a case where we are calculating group statistics over `"c"` twice:

```
df.select(
    pl.all(),
    pl.col("c").count().over("c").alias("size"),
    pl.col("c").sum().over("type").alias("sum"),
    pl.col("c").reverse().over("c").flatten().alias("reverse_type")
)
```

```
shape: (7, 5)
┌─────┬──────┬──────┬─────┬──────────────┐
│ c   ┆ type ┆ size ┆ sum ┆ reverse_type │
│ --- ┆ ---  ┆ ---  ┆ --- ┆ ---          │
│ i64 ┆ str  ┆ u32  ┆ i64 ┆ i64          │
╞═════╪══════╪══════╪═════╪══════════════╡
│ 1   ┆ m    ┆ 3    ┆ 5   ┆ 2            │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ 1   ┆ n    ┆ 3    ┆ 5   ┆ 2            │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ 1   ┆ o    ┆ 3    ┆ 1   ┆ 2            │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ 2   ┆ m    ┆ 4    ┆ 5   ┆ 2            │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ 2   ┆ m    ┆ 4    ┆ 5   ┆ 1            │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ 2   ┆ n    ┆ 4    ┆ 5   ┆ 1            │
├╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌╌┼╌╌╌╌╌┼╌╌╌╌╌╌╌╌╌╌╌╌╌╌┤
│ 2   ┆ n    ┆ 4    ┆ 5   ┆ 1            │
└─────┴──────┴──────┴─────┴──────────────┘
```

Missing data

`Pandas` uses `NaN` and/or `None` values to indicate missing values depending on the dtype of the column. In addition the behaviour in `Pandas` varies depending on whether the default dtypes or optional nullable arrays are used. In `Polars` missing data corresponds to a `null` value for all data types.

For float columns `Polars` permits the use of `NaN` values. These `NaN` values are not considered to be missing data but instead a special floating point value.

In `Pandas` an integer column with missing values is cast to be a float column with `NaN` values for the missing values (unless using optional nullable integer dtypes). In `Polars` any missing values in an integer column are simply `null` values and the column remains an integer column.

See the missing data section for more details.

4.9.2 Coming from Apache Spark

Column-based API vs. Row-based API

Whereas the `Spark` `DataFrame` is analogous to a collection of rows, a `Polars` `DataFrame` is closer to a collection of columns. This means that you can combine columns in `Polars` in ways that are not possible in `Spark`, because `Spark` preserves the relationship of the data in each row.

Consider this sample dataset:

```
import polars as pl

df = pl.DataFrame({
    "foo": ["a", "b", "c", "d", "d"],
    "bar": [1, 2, 3, 4, 5],
})

dfs = spark.createDataFrame(
    [
        ("a", 1),
        ("b", 2),
        ("c", 3),
        ("d", 4),
        ("d", 5),
    ],
    schema=["foo", "bar"],
)
```

EXAMPLE 1: COMBINING `HEAD` AND `SUM`

In `Polars` you can write something like this:

```
df.select(
    pl.col("foo").sort().head(2),
    pl.col("bar").filter(pl.col("foo") == "d").sum()
)
```

Output:

```
shape: (2, 2)
┌─────┬─────┐
│ foo ┆ bar │
│ --- ┆ --- │
│ str ┆ i64 │
╞═════╪═════╡
│ a   ┆ 9   │
├╌╌╌╌╌┼╌╌╌╌╌┤
│ b   ┆ 9   │
└─────┴─────┘
```

The expressions on columns `foo` and `bar` are completely independent. Since the expression on `bar` returns a single value, that value is repeated for each value output by the expression on `foo`. But `a` and `b` have no relation to the data that produced the sum of `9`.

To do something similar in `Spark`, you'd need to compute the sum separately and provide it as a literal:

```
from pyspark.sql.functions import col, sum, lit

bar_sum = (
    dfs
    .where(col("foo") == "d")
    .groupBy()
    .agg(sum(col("bar")))
    .take(1)[0][0]
)

(
    dfs
    .orderBy("foo")
    .limit(2)
    .withColumn("bar", lit(bar_sum))
    .show()
)
```

Output:

```
+---+---+
|foo|bar|
+---+---+
|  a|  9|
```

```
|  b|  9|
+---+---+
```

EXAMPLE 2: COMBINING TWO `HEAD` S

In `Polars` you can combine two different `head` expressions on the same DataFrame, provided that they return the same number of values.

```
df.select(
    pl.col("foo").sort().head(2),
    pl.col("bar").sort(descending=True).head(2),
)
```

Output:

```
shape: (3, 2)
┌─────┬─────┐
│ foo ┆ bar │
│ --- ┆ --- │
│ str ┆ i64 │
╞═════╪═════╡
│ a   ┆ 5   │
├╌╌╌╌╌┼╌╌╌╌╌┤
│ b   ┆ 4   │
└─────┴─────┘
```

Again, the two `head` expressions here are completely independent, and the pairing of `a` to `5` and `b` to `4` results purely from the juxtaposition of the two columns output by the expressions.

To accomplish something similar in `Spark`, you would need to generate an artificial key that enables you to join the values in this way.

```
from pyspark.sql import Window
from pyspark.sql.functions import row_number

foo_dfs = (
    dfs
    .withColumn(
        "rownum",
        row_number().over(Window.orderBy("foo"))
    )
)

bar_dfs = (
    dfs
    .withColumn(
        "rownum",
        row_number().over(Window.orderBy(col("bar").desc()))
    )
)

(
    foo_dfs.alias("foo")
    .join(bar_dfs.alias("bar"), on="rownum")
    .select("foo.foo", "bar.bar")
    .limit(2)
    .show()
)
```

Output:

```
+---+---+
|foo|bar|
+---+---+
|  a|  5|
|  b|  4|
+---+---+
```

4.10 Misc

4.10.1 Multiprocessing

TLDR: if you find that using Python's built-in `multiprocessing` module together with Polars results in a Polars error about multiprocessing methods, you should make sure you are using `spawn`, not `fork`, as the starting method:

Python

```python
from multiprocessing import get_context


def my_fun(s):
    print(s)


with get_context("spawn").Pool() as pool:
    pool.map(my_fun, ["input1", "input2", ...])
```

When not to use multiprocessing

Before we dive into the details, it is important to emphasize that Polars has been built from the start to use all your CPU cores. It does this by executing computations which can be done in parallel in separate threads. For example, requesting two expressions in a `select` statement can be done in parallel, with the results only being combined at the end. Another example is aggregating a value within groups using `groupby().agg(<expr>)`, each group can be evaluated separately. It is very unlikely that the `multiprocessing` module can improve your code performance in these cases.

See the optimizations section for more optimizations.

When to use multiprocessing

Although Polars is multithreaded, other libraries may be single-threaded. When the other library is the bottleneck, and the problem at hand is parallelizable, it makes sense to use multiprocessing to gain a speed up.

The problem with the default multiprocessing config

SUMMARY

The Python multiprocessing documentation lists the three methods to create a process pool:

1. spawn
2. fork
3. forkserver

The description of fork is (as of 2022-10-15):

> The parent process uses os.fork() to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic.
>
> Available on Unix only. The default on Unix.

The short summary is: Polars is multithreaded as to provide strong performance out-of-the-box. Thus, it cannot be combined with `fork`. If you are on Unix (Linux, BSD, etc), you are using `fork`, unless you explicitly override it.

The reason you may not have encountered this before is that pure Python code, and most Python libraries, are (mostly) single threaded. Alternatively, you are on Windows or MacOS, on which `fork` is not even available as a method (for MacOS it was up to Python 3.7).

Thus one should use `spawn`, or `forkserver`, instead. `spawn` is available on all platforms and the safest choice, and hence the recommended method.

EXAMPLE

The problem with `fork` is in the copying of the parent's process. Consider the example below, which is a slightly modified example posted on the Polars issue tracker:

Python

```python
import multiprocessing
import polars as pl


def test_sub_process(df: pl.DataFrame, job_id):
    df_filtered = df.filter(pl.col("a") > 0)
    print(f"Filtered (job_id: {job_id})", df_filtered, sep="\n")


def create_dataset():
    return pl.DataFrame({"a": [0, 2, 3, 4, 5], "b": [0, 4, 5, 56, 4]})


def setup():
    # some setup work
    df = create_dataset()
    df.write_parquet("/tmp/test.parquet")


def main():
    test_df = pl.read_parquet("/tmp/test.parquet")

    for i in range(0, 5):
        proc = multiprocessing.get_context("spawn").Process(
            target=test_sub_process, args=(test_df, i)
        )
        proc.start()
        proc.join()

        print(f"Executed sub process {i}")


if __name__ == "__main__":
    setup()
    main()
```

Using `fork` as the method, instead of `spawn`, will cause a dead lock. Please note: Polars will not even start and raise the error on multiprocessing method being set wrong, but if the check had not been there, the deadlock would exist.

The fork method is equivalent to calling `os.fork()`, which is a system call as defined in the POSIX standard:

> A process shall be created with a single thread. If a multi-threaded process calls fork(), the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal-safe operations until such time as one of the exec functions is called.

In contrast, `spawn` will create a completely new fresh Python interpreter, and not inherit the state of mutexes.

So what happens in the code example? For reading the file with `pl.read_parquet` the file has to be locked. Then `os.fork()` is called, copying the state of the parent process, including mutexes. Thus all child processes will copy the file lock in an acquired state, leaving them hanging indefinitely waiting for the file lock to be released, which never happens.

What makes debugging these issues tricky is that `fork` can work. Change the example to not having the call to `pl.read_parquet` :

Python

```python
import multiprocessing
import polars as pl


def test_sub_process(df: pl.DataFrame, job_id):
    df_filtered = df.filter(pl.col("a") > 0)
    print(f"Filtered (job_id: {job_id})", df_filtered, sep="\n")


def create_dataset():
    return pl.DataFrame({"a": [0, 2, 3, 4, 5], "b": [0, 4, 5, 56, 4]})


def main():
    test_df = create_dataset()

    for i in range(0, 5):
        proc = multiprocessing.get_context("fork").Process(
            target=test_sub_process, args=(test_df, i)
        )
        proc.start()
        proc.join()

        print(f"Executed sub process {i}")


if __name__ == "__main__":
    main()
```

This works fine. Therefore debugging these issues in larger code bases, i.e. not the small toy examples here, can be a real pain, as a seemingly unrelated change can break your multiprocessing code. In general, one should therefore never use the `fork` start method with multithreaded libraries unless there are very specific requirements that cannot be met otherwise.

PRO'S AND CONS OF FORK

Based on the example, you may think, why is `fork` available in Python to start with?

First, probably because of historical reasons: `spawn` was added to Python in version 3.4, whilst `fork` has been part of Python from the 2.x series.

Second, there are several limitations for `spawn` and `forkserver` that do not apply to `fork` , in particular all arguments should be pickable. See the Python multiprocessing docs for more information.

Third, because it is faster to create new processes compared to `spawn` , as `spawn` is effectively `fork` + creating a brand new Python process without the locks by calling execv. Hence the warning in the Python docs that it is slower: there is more overhead to `spawn` . However, in almost all cases, one would like to use multiple processes to speed up computations that take multiple minutes or even hours, meaning the overhead is negligible in the grand scheme of things. And more importantly, it actually works in combination with multithreaded libraries.

Fourth, `spawn` starts a new process, and therefore it requires code to be importable, in contrast to `fork` . In particular, this means that when using `spawn` the relevant code should not be in the global scope, such as in Jupyter notebooks or in plain scripts. Hence in the examples above, we define functions where we spawn within, and run those functions from a `__main__` clause. This is not an issue for typical projects, but during quick experimentation in notebooks it could fail.

References

1. https://docs.python.org/3/library/multiprocessing.html
2. https://pythonspeed.com/articles/python-multiprocessing/
3. https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html
4. https://bnikolic.co.uk/blog/python/parallelism/2019/11/13/python-forkserver-preload.html

4.10.2 Alternatives

These are some tools that share similar functionality to what polars does.

- Pandas

  A very versatile tool for small data. Read 10 things I hate about pandas written by the author himself. Polars has solved all those 10 things. Polars is a versatile tool for small and large data with a more predictable, less ambiguous, and stricter API.

- Pandas the API

  The API of pandas was designed for in memory data. This makes it a poor fit for performant analysis on large data (read anything that does not fit into RAM). Any tool that tries to distribute that API will likely have a suboptimal query plan compared to plans that follow from a declarative API like SQL or Polars' API.

- Dask

  Parallelizes existing single-threaded libraries like `NumPy` and `Pandas`. As a consumer of those libraries Dask therefore has less control over low level performance and semantics. Those libraries are treated like a black box. On a single machine the parallelization effort can also be seriously stalled by pandas strings. Pandas strings, by default, are stored as python objects in numpy arrays meaning that any operation on them is GIL bound and therefore single threaded. This can be circumvented by multi-processing but has a non-trivial cost.

- Modin

  Similar to Dask

- Vaex

  Vaexs method of out-of-core analysis is memory mapping files. This works until it doesn't. For instance parquet or csv files first need to be read and converted to a file format that can be memory mapped. Another downside is that the OS determines when pages will be swapped. Operations that need a full data shuffle, such as sorts, have terrible performance on memory mapped data. Polars' out of core processing is not based on memory mapping, but on streaming data in batches (and spilling to disk if needed), we control which data must be hold in memory, not the OS, meaning that we don't have unexpected IO stalls.

- DuckDB

  Polars and DuckDB have many similarities. DuckDB is focused on providing an in-process OLAP Sqlite alternative, Polars is focused on providing a scalable `DataFrame` interface to many languages. Those different front-ends lead to different optimization strategies and different algorithm prioritization. The interoperability between both is zero-copy. See more: https://duckdb.org/docs/guides/python/polars

- Spark

  Spark is designed for distributed workloads and uses the JVM. The setup for spark is complicated and the startup-time is slow. On a single machine Polars has much better performance characteristics. If you need to process TB's of data Spark is a better choice.

- CuDF

  GPU's and CuDF are fast! However, GPU's are not readily available and expensive in production. The amount of memory available on a GPU is often a fraction of the available RAM. This (and out-of-core) processing means that Polars can handle much larger data-sets. Next to that Polars can be close in performance to CuDF. CuDF doesn't optimize your query, so is not uncommon that on ETL jobs Polars will be faster because it can elide unneeded work and materializations.

- Any

  Polars is written in Rust. This gives it strong safety, performance and concurrency guarantees. Polars is written in a modular manner. Parts of polars can be used in other query programs and can be added as a library.

4.10.3 Reference Guides

The api documentations with details on function / object signatures can be found here:

- NodeJS
- Python
- Rust

4.10.4 Contributing

See the `CONTRIBUTING.md` if you would like to contribute to the `Polars` project.

If you're new to this we recommend starting out with contributing examples to the Python API documentation. The Python API docs are generated from the docstrings of the Python wrapper located in `polars/py-polars`.

Here is an example commit that adds a docstring.

If you spot any gaps in this User Guide you can submit fixes to the `pola-rs/polars-book` repo.

Happy hunting!