🐻‍❄️ Polars: Zero to Hero

What it is, when to use it, and how to get started

👤 Luca Baggi
💼 ML Engineer

📍 Talk outline

🐻‍❄️ What is Polars?

🚀 What makes Polars so fast?

🗝️ Key concepts

🧪 Is Polars production ready?

❌ When should I *not* use Polars?

✋ Question time

🏷️ References

📚 Appendix 1: Small Polars compendium

🐻‍❄️ What is Polars?

In a nutshell

> Dataframes powered by a multithreaded, vectorized query engine, written in Rust

- A `DataFrame` frontend, i.e. work with Python and not a SQL table.
- Utilises all cores on your machine, efficiently (more on this later).
- Has a **query engine** with state-of-the-art algorithms.
- In-process, like `sqlite`.
- Has no other default dependencies (could run in an AWS lambda).

🐻‍❄️ What is Polars?

What it is not

- Not a distributed system like `apache/spark`: runs on one node (for now).

Polars, however, can increase your data processing capabilities so much that you will only need `pyspark` for truly big data, i.e. **complex transformations on more than 1TB**.

> Where the pipeline is simple Polars' streaming mode vastly outperforms Spark and is recommended for all dataset sizes. Palantir Technologies

🚀 What makes Polars so fast?

The key ingredients

1. Efficient in-memory representation of the data, following Apache Arrow specification
2. Custom file readers: CSV, parquet, including AWS, HuggingFace…
3. Work stealing, AKA efficient multithreading, thanks to `rayon` and Rust 🦀
4. State-of-the-art algorithms to manipulate data.
5. Extensive optimisations through lazy evaluation.

For a thorough introduction by its author, you should check out this and this videos.

🚀 What makes Polars so fast?

Apache Arrow

Arrow is a cross-language specification on how to represent data in memory.

1. It's a **columnar memory format** for high-performance analytical queries.
2. Native way to represent missing value (unlike `numpy`).
3. Efficient representation of strings categorical/enum data types.
4. Support for **nested datatypes** too: `arrays`, `lists` (arrays of heterogeneous length) and `structs` (dictionaries). See here for more.

🚀 What makes Polars so fast?

Extensive optimisations through lazy evaluation

Polars has two modes: **lazy** and **eager** (more on this later).

When in lazy mode, Polars builds a *query plan* which is optimised extensively - for example by removing unnecessary expressions as well as branches in the computation and automatically caching bits of data that will be re-used.

🗝️ Key Concepts

Lazy and eager mode

- In eager mode, the transformations you write are executed immediately.
- In lazy mode, the transformations you write are stored in a query plan.

  The plan is then executed when you call `.collect()`.

The lazy mode enables query optimisations that are not possible in the eager mode across the entire data transformation pipeline.

🗝️ Key Concepts

Lazy and eager mode: an example

The most powerful feature about Polars lazy mode is that **you don't have to do anything different to use it**.

```
# eager: every operation is executed in the same order it's written
pl.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})

# lazy: it's not a dataframe, it represents the *sequence of transformat
pl.LazyFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
```

See the appendix for more details and examples.

🗝️ Key Concepts

Streaming mode

Polars default engine tries to *load all data in memory*. In general, Polars tries to maximise memory usage (i.e., use all resources on your machine) to get the most from its query engine. When you have more data than your RAM, though, you can use the `streaming` mode.

In streaming mode, Polars reads data in batches, and only keeps the last batch in memory.

Note: streaming mode is only available in Lazy mode. Also, technically it's not stable yet.

🗝️ Key Concepts

Streaming mode: an example

This is all it takes to enable the streaming mode:

```python
data = pl.scan_csv("path/to/source.csv")

# do your transforms...

data.collect(streaming=True)
```

🗝️ Key Concepts

Contexts and expressions

Polars has a powerful syntax (technically, it’s a DSL, i.e. a Domain Specific Language) that allows writing data transformations in an expressive and powerful syntax. The two main concepts are *expressions* and *contexts*.

🗝️ Key Concepts

Expressions

An **expression** is a lazy representation of a data transformation. You can use expressions as building blocks to build more complex ones. Because expressions are **lazy, no computations have taken place yet**. That's what we need **contexts** for.

```
expression = pl.col("weight") / (pl.col("height") ** 2)
```

This doesn't do anything of its own. It represents the operation to compute the ratio of the weight of a person to the square of their height.

🗝️ Key Concepts

Contexts

Polars expressions need a context in which they are executed to produce a result. Depending on the context it is used in, the same Polars expression can produce different results.

```
data = pl.LazyFrame({"weight": [100, 200, 300], "height": [160, 170, 180
data.with_columns(expression.alias("BMI")).collect()
```

See the appendix for more details and examples.

🧪 Is Polars production ready?

Yes.

1. Most popular `pandas` alternative (~7M monthly downloads). Unlike other alternatives, is getting traction!
2. Is backed by a **company**, with a clear **product roadmap**: a paid Polars Cloud and, in the future, a distributed engine.
3. Has a broad pool of **maintainers** (>10) beyond the company: QuanSight, edge funds, former JP Morgan employees.
4. NumFOCUS affiliated project.

❌ When not to use Polars

Three cases that come to mind

1. Technically, the streaming engine is still in beta. A new version is being worked on and will be released soon.
2. Big ETL workloads where latency is a major requirement.
3. Real time/streaming (e.g. cannot connect to Kafka topics).
4. Excel manipulation and other esoteric file formats (Polars leverages `calamine` to read Excel files, but it might not be as documented as other file formats).

✋Question time

References

- Polars API Reference and User Guide.
- Polars blog with case studies.
- Polars Discord.
- A small series of Polars katas, by yours truly.
- Python Polars: The Definitive Guide
- Polars Cookbook.

📚 Appendix 1: Small Polars compendium

🪠 Eager I/O

```python
import polars as pl

# can be: csv, parquet, excel, json, database
data = pl.read_*("/path/to/source.*")
data.write_*("path/to/destination.*")
```

- Reference

📚 Appendix 1: Small Polars compendium

🪠 Lazy I/O

```
raw = pl.scan_*("/path/to/source.*") # creates a LazyFrame
raw = pl.scan_parquet("/path/to/*.parquet") # read_parquet works too
processed.sink_parquet("path/to/destination.parquet")
```

- IO

📚 Appendix 1: Small Polars compendium

🪠 What about other formats?

```
raw = pd.read_*("path/to/source.weird.format") # like stata, spss ...

data = pl.from_pandas(raw)
```

📚 Appendix 1: Small Polars compendium

😴 Lazy <=> Eager

```
# from eager to lazy (not recommended)
df: pl.DataFrame = pl.read_csv("path/to/source.csv")

lazy_df: pl.LazyFrame = df.lazy()

# from lazy to eager
lazy_df: pl.LazyFrame = pl.scan_csv("path/to/source.csv")

df = lazy_df.collect()
```

- Lazy API

📚 Appendix 1: Small Polars compendium

🛠️ Data wrangling: selection

```
raw.select(
   "col1", "col2"
   pl.col("col1", "col2"),
   pl.col(pl.DataType),        # any valid polars datatype
   pl.col("*"),
   pl.col("$A.*^]"),           # all columns that match a regex pattern
   pl.all(),
   pl.all().exclude( ... )     # names, regex, types...
)
```

- Polars column selection

📚 Appendix 1: Small Polars compendium

🛠️ Data wrangling: selection, but for cool kids

```python
import polars.selectors as cs

df.select(
   cs.contains("date"),
   cs.string() | cs.ends_with("_high") # uses set operations!
   ~cs.temporal()
)
```

- Selectors

📚 Appendix 1: Small Polars compendium

🛠️ Data wrangling: manipulate columns

```
(
  questions
  .with_columns(
    # work with dates
    pl.col("start", "end").dt.day().suffix("_day"),
    pl.col("time_spent").dt.seconds().cast(pl.UInt16).alias("sec"),
    # work with strings
    pl.col("id").str.replace("uuid_", ""),
    # work with arrays!
    pl.col("name").str.split(" ").arr.first().alias("first_name"),
    pl.col("name").str.split(" ").arr.last().alias("last_name"),
    # work with dictionaries
    pl.col("content").struct.field("nested_field")
  )
)
```

- Expressions

📚 Appendix 1: Small Polars compendium

🛠️ Data wrangling: filtering

```
(
  raw
  .sort("simulation_created_at")
  .filter(
    pl.col("simulation_platform").eq("Medicine"),
    # do this in SQL!
    pl.count().over("question_uid", "student_uid") == 1
  )
)
```

📚 Appendix 1: Small Polars compendium

🛠️ Data wrangling: `groupby`

```
(
  raw
  .groupby("question_uid")
  .agg(
    pl.col("correct", "time_spent").mean().suffix("_mean"),
    pl.col("student_uid").n_unique().shrink_dtype().alias("times_seen"),
    pl.col("question_category_path", "simulation_platform").first(),
  )
)
```

And it works for up- and down-sampling date types too (temporal aggregation)!

🙏 Thank you!

Please share your feedback! My address is lucabaggi [at] duck.com

LinkedIn

GitHub