Pandas vs Polars

The Ultimate Battle for High Performance in Python

Muhammad Adeel Sultan Khan

Data Scientist Agtech, Silicon Valley, California

Pearson

Personal Background

- Muhammad Adeel Sultan Khan
- Ag tech Data Scientist Silicon Valley tech company over 5 years
- Treasury Finance 8 years experience
- MSIS Data Science Analytics, Santa Clara University
- MBA San Jose State University
- CFA Level 1 exam
- BCS & MBA MIS, Institute of Business Administration

Pearson

Agenda

- **Segment 1: Introduction Pandas vs. Polars**
  - What is Pandas
  - What is the purpose and function of Polars
  - What makes Polars faster and more efficient than Pandas
  - Why Polars is more efficient for large datasets
  - How Polars handles larger than RAM data
  - What is Polars query optimization claim
- **Segment 2: Strengths & limitations of Polars & Pandas**
  - Pandas vs Polars; syntax and speed comparison
  - How can we improve data mining performance
- **Segment 3: When to choose Polars**
  - Examples of high-performance Pandas alternatives
  - Why choose Polars
  - Polars execution of code
  - Examples of multithreading and parallel computing
  - Installing Polars in Python
  - Demo of Polars vs Pandas data analysis

Pearson

Poll Question

- How familiar are you with using pandas for data science tasks like data mining? (one selection)
  - Beginner
  - Intermediate
  - Expert

What is Pandas

- Pandas is python's powerful & open source data manipulation & data analysis tool
- Pandas key functions are data loading, aligning, manipulating & merging
- Pandas has two data structures series(1-d) & dataframe(2-d)
- Pandas has usecases in finance, statistics, healthcare, engineering, iot sensor data
- Pandas help in the most complex data transformation tasks
- Pandas helps in preparing data that is used for building ML & AI models

Pearson

pandas

Pandas Data Types

Pandas is well suited for various types of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data need not be labeled at all to be placed into a pandas data structure

Pandas main tasks

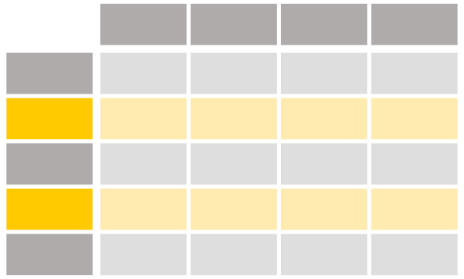
- Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data
- Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
- Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations
- Powerful, flexible group by functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
- Make it easy to convert ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects

Pandas main tasks

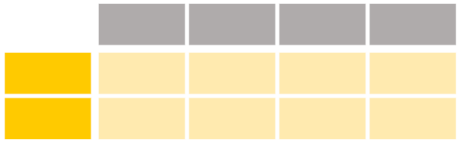
- Intuitive merging and joining data sets
- Flexible reshaping and pivoting of data sets
- Hierarchical labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast HDF5 format
- Time series-specific functionality: date range generation and frequency conversion, moving window statistics, date shifting, and lagging.
- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets

Pandas data handling

```
In [1]: import pandas as pd
```



Selecting rows

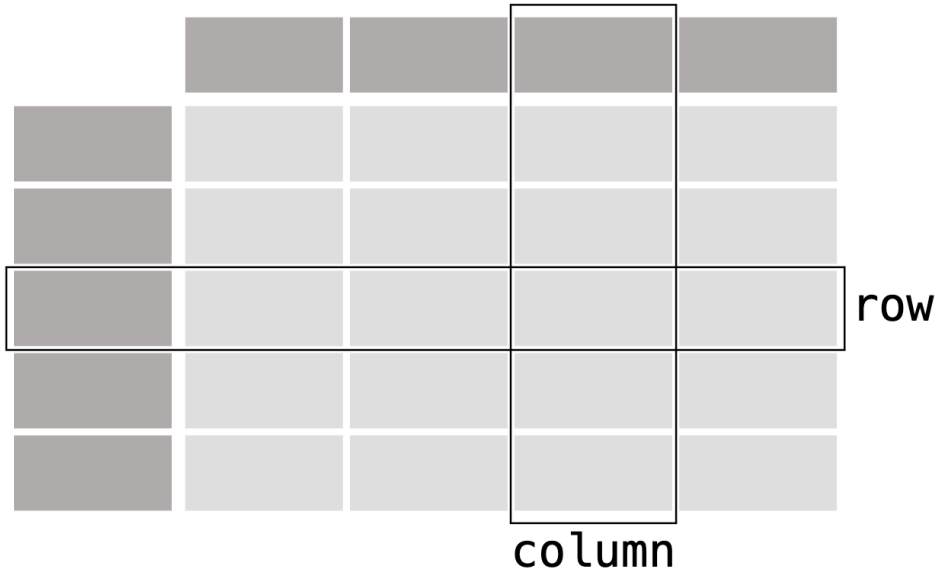


Reading & writing data

read_*

to_*

DataFrame



Selecting columns

Data Mining with Pandas

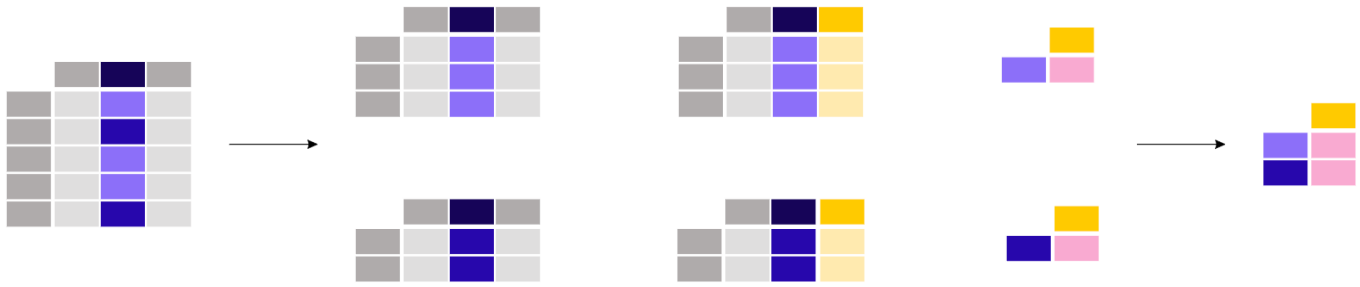
- Pandas help calculate summary statistics

```
In [4]: titanic["Age"].mean()
Out[4]: 29.69911764705882
```

```
In [5]: titanic[["Age", "Fare"]].median()
Out[5]:
Age     28.0000
Fare    14.4542
dtype: float64
```

```
In [10]: titanic.groupby("Sex")["Age"].mean()
Out[10]:
Sex
female    27.915709
male      30.726645
Name: Age, dtype: float64
```

```
titanic      .groupby("Sex")      ["Age"]      .mean()
```



```
In [2]: titanic = pd.read_csv("data/titanic.csv")

In [3]: titanic.head()
Out[3]:
   PassengerId  Survived  Pclass  ...     Fare Cabin  Embarked
0            1         0       3  ...   7.2500   NaN         S
1            2         1       1  ...  71.2833   C85         C
2            3         1       3  ...   7.9250   NaN         S
3            4         1       1  ...  53.1000  C123         S
4            5         0       3  ...   8.0500   NaN         S

[5 rows x 12 columns]
```

```
In [7]: titanic.agg(
   ...:     {
   ...:         "Age": ["min", "max", "median", "skew"],
   ...:         "Fare": ["min", "max", "median", "mean"],
   ...:     }
   ...: )
   ...:
Out[7]:
                Age         Fare
min        0.420000     0.000000
max       80.000000   512.329200
median    28.000000    14.454200
skew       0.389108          NaN
mean            NaN    32.204208
```

Pandas package version

- 2.2.0 (Jan 19 2024)
- 2.1.4 (Dec 8 2023)
- 2.0.3 (Jun 28 2023)
- 1.5.3 (Jan 18 2023)
- 1.4.4 (Aug 31 2022)
- 1.3.5 (Dec 12 2021)
- 1.2.5 (Jun 22 2021)
- 1.1.5 (Dec 7 2020)
- 0.4 (Oct 9 2011)

Installing from PyPI

pandas can be installed via pip from PyPI.

```
pip install pandas
```

pandas requires the following dependencies.

Package	Minimum supported version
NumPy	1.22.4
python-dateutil	2.8.2
pytz	2020.1
tzdata	2022.1

Poll Question

- What are the two most popular pandas data structures?
  - Series: 1d labelled array
  - DataFrame: 2d labelled tabular structure
  - Lists
  - Tree

What is Polars

- Polars is a high performance dataframe package for data analysis
- It is designed to be faster, efficient & provide parallel processing
- Polars is especially suited for larger than memory datasets (>GB)
- Polars is A Python & Rust library; blazingly fast & memory efficient
- Polars utilizes multiple CPU cores at once
- Polars provides smooth transition from pandas with minimum hastle
- Polars enhances performance & utilizes machine resources efficiently
- Polars supports lazy evaluation: delaying query/operation execution until needed

01
Fast

02
Easy to use

03
Open source

Pearson

What is the purpose/function of Polars

- ***<u>The goal of Polars is to provide lightning fast DataFrame library that:</u>***
  - 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 combines the flexibility and user-friendliness of Python with the speed and scalability of Rust

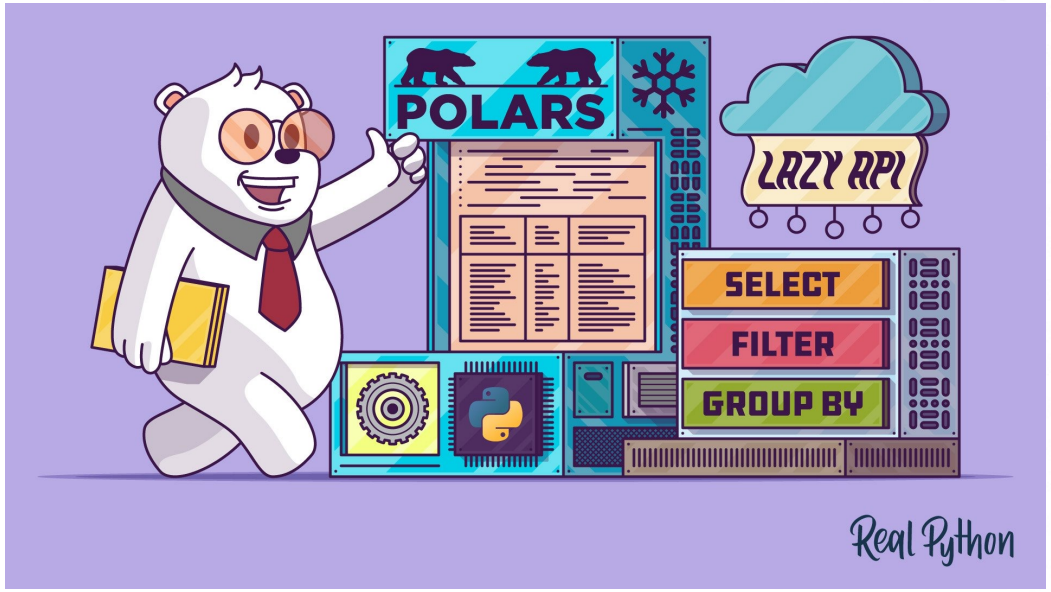
DataFrames for the new era

Pearson

Polars

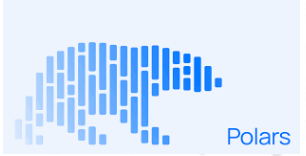
What is the purpose/function of Polars

- Polars is written in Rust which gives it C/C++ performance and allows it to fully control performance critical parts in a query engine. Rust's ability to be immediately compiled into machine code without the use of an interpreter can make it faster than Python.
- Polars enables to:
  - Reduce redundant copies.
  - Traverse memory cache efficiently.
  - Minimize contention in parallelism.
  - Process data in chunks.
  - Reuse memory allocations.



Polars Key Features

- ***Powerful API***: Polars offers easy to use API for data manipulation & analysis tasks. It includes wide range of data aggregation & transformation operations including filtering, sorting & joining
- ***High Performance:*** Polars leverages multithreading by using parallelized data processing executed on multiple threads of multi-core processors
- ***Optimized Queries:*** Polars offers query optimization techniques using optimized data access patterns & minimized data movement
- ***Hybrid Streaming(larger than RAM datasets):*** Polars uses streaming method to work on larger-than-memory datasets by using small batches of data



Polars Key Features

- Streaming implies working on data in batches rather than at once
- Streaming is only used in lazy mode in polars
- ***Lazy & Eager execution:*** Polars support both lazy and eager execution(used by pandas) of data. Lazy execution defers operations until results are needed, while eager execution executes operations immediately when called.

```python
query = pl.scan_csv("iris.csv").group_by("species").agg(
    pl.col("sepal_width").mean().alias("mean_sepal_width")
)
query.collect(streaming=True)
```

Pearson

Polars

Polars Key Features

```python
query.explain(streaming=True)
```

To determine which parts of your query are streaming, use the explain method

```
--- STREAMING
AGGREGATE
        [col("sepal_width").mean().alias("mean_sepal_width")] BY [col("species")] FROM

    Csv SCAN iris.csv
    PROJECT 2/5 COLUMNS  --- END STREAMING

  DF []; PROJECT */0 COLUMNS; SELECTION: "None"
```

Pearson

Polars

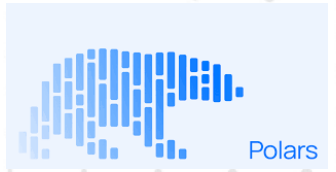
What makes Polars faster & more efficient than Pandas

- ***Performance***: Polars is designed for high performance, leveraging ***Rust's multi-threading, SIMD*** instructions, and query optimizations. In contrast, Pandas is primarily single-threaded, which may not be as efficient for large datasets or complex operations compared to Polars. Combining the ability to do streaming, as well as lazy execution, and Polars has a strong claim to being the better package for datasets above GB size.
- ***Memory Model*****:** Polars uses Apache Arrow Columnar Format as its memory model, optimized for data processing while Pandas, on the other hand, uses a row-based memory format. This makes Pandas less efficient for certain operations (which are quite common) like column-wise computations and aggregations.

https://www.einblick.ai/tools/what-is-polars-python-compare-pandas/

What makes Polars faster & more efficient than Pandas

- ***Syntax and API***: Polars has a syntax and API that is similar to Pandas, making it easy for users familiar with Pandas to transition to Polars
- ***Data Types***: Polars provides a more extensive set of data types compared to Pandas, including advanced data types like Date64, datetime, and Interval, as well as support for custom data types, providing greater flexibility for handling diverse data types.
- ***Rust advantage*****:** Since polars is written in Rust, it utilizes the benefits of concurrency which allows parallelism. This enables Polars to use all machine cores while Pandas uses only single core to carry out data processing operations**.**



Pearson

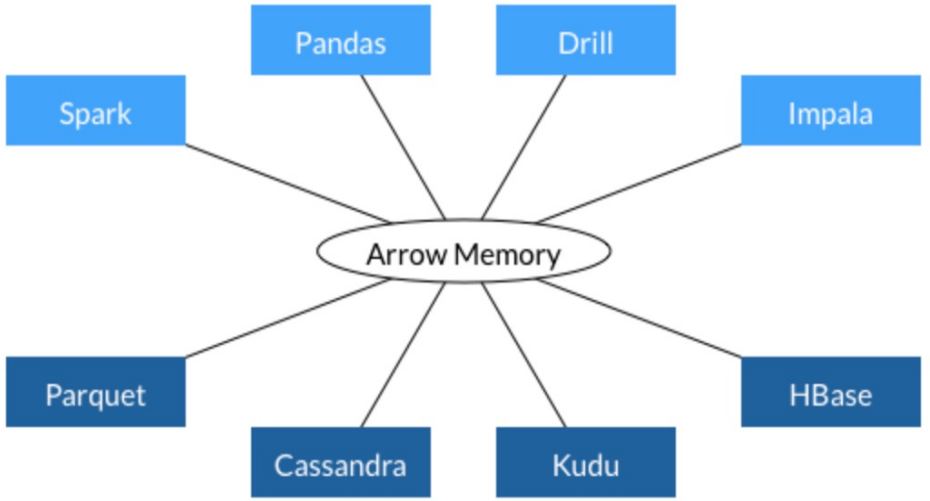
https://www.einblick.ai/tools/what-is-polars-python-compare-pandas/

What makes Polars faster & more efficient than Pandas

- Pandas 2.0 is designed to support Apache Arrow for data memory thus leading to performance improvements
- However, Polars continues to be faster than Pandas 2.0 based on:
  - **Rust advantage**; Rust's ability to be compiled into machine code without use of interpreter makes it faster than Python.
  - Parallelism
  - Interoperability (no need to convert data into other types in memory)
  - Query Optimization (lazy & eager)
  - Expressive API (avoids Pandas apply method which loops over all rows of dataframe)

Why is Polars so fast

- ***Written in Rust***: low level language with minimal runtime & memory related bugs, provides concurrency & parallelism. Pandas is built on python and numpy which makes it slow
- ***Based on Arrow***: Apache Arrow is language independent memory format that provides interoperability which:
  - Speeds up performance as it avoids converting data to other formats in data pipeline
  - Arrow standardizes the in-memory data format
  - Arrow is memory efficient - two processes can share the same data without needing to make a copy.



Apache Arrow Advantage

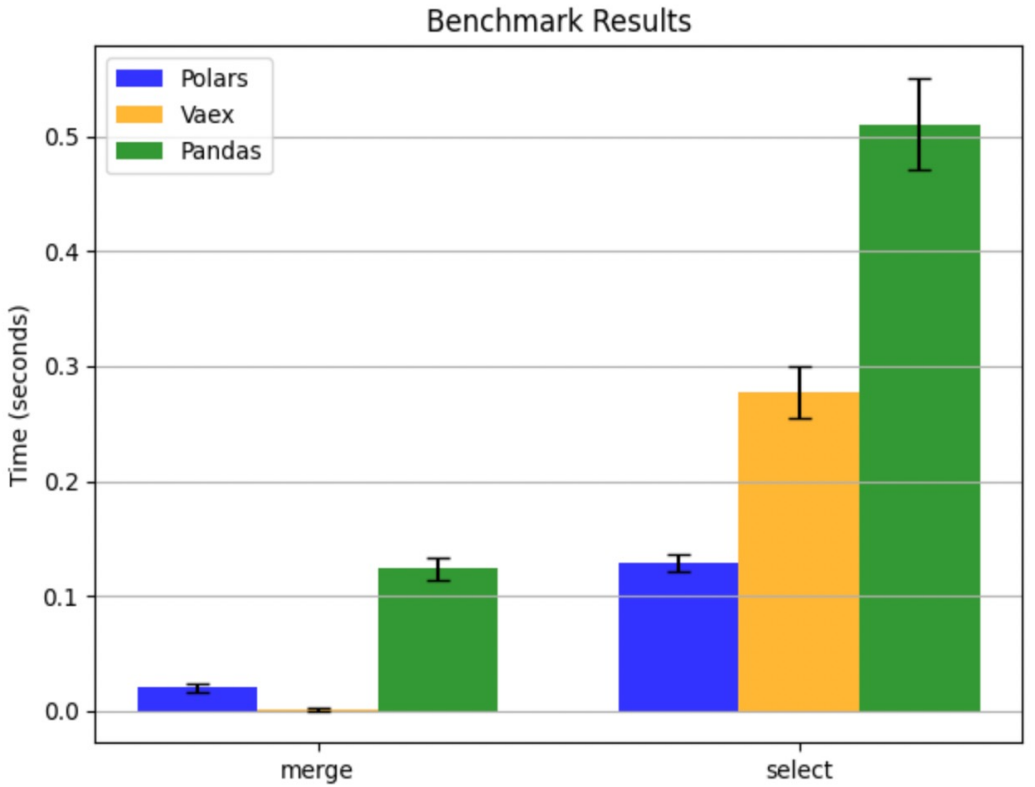
- Apache arrow contributes to polar's fast performance
- Arrow was created by Wes McKinney to address big datasets issues with pandas
- It is also the backend for pandas 2.0, a more performant version of pandas released in March of this year.
- Pandas 2.0 is built on PyArrow, the Polars team built their own Arrow implementation
- Arrow has built-in support for a wider range of data types than pandas. As pandas is based on NumPy, it is excellent at handling integer and float columns, but struggles with other data types.
- In contrast, Arrow has support for datetime, boolean, binary, and even complex column types, such as those containing lists. In addition, Arrow is able to natively handle missing data, which requires a workaround in NumPy.
- Finally, Arrow uses columnar data storage, which means that, regardless of the data type, all columns are stored in a continuous block of memory. This not only makes parallelism easier, but also makes data retrieval faster.

Why is Polars so fast

- Pandas 2.0 is built on PyArrow but Polars has its own implementation of Arrow
- Arrow has built-in support for wide range of data types than pandas
- Pandas is built on NumPy, it supports integer & float only with ease
- Polars' Arrow supports datetime, boolean, binary and lists datatypes
- Arrow can handle missing data which requires imputation in

  pandas numpy
- Polars Arrow memory uses columnar data storage which stores

  all columns in a continuous block of memory regardless of data type
- This supports both parallelism and faster data retrieval



Pearson

Polars

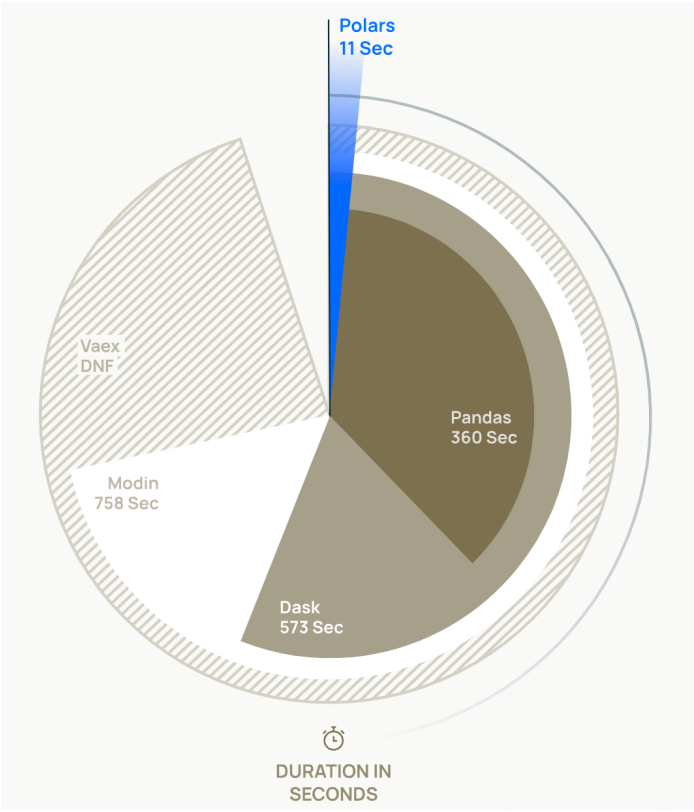
Rust Advantage

- ***Performance*:** Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.
- ***Reliability*:** Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time
- ***Productivity*:** Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more

https://www.rust-lang.org/

Why use Polars over Pandas

- Polars memory requirement for data operations is much smaller than Pandas
- Pandas require 5 to 10 times as much RAM as the size of a dataset
- Polars require 2 to 4 times as much RAM as the size of the dataset
- Polars is 10 and 100 times faster than Pandas for operations
- Polars was benchmarked against other solutions using TPCH benchmark
- The benchmark results showed Polars had performance gains over others due to its parallel execution engine, vectorization with SIMD(single instruction multiple data) protocol
- The results of the benchmark showed Polars can achieve 30x performance gains over pandas



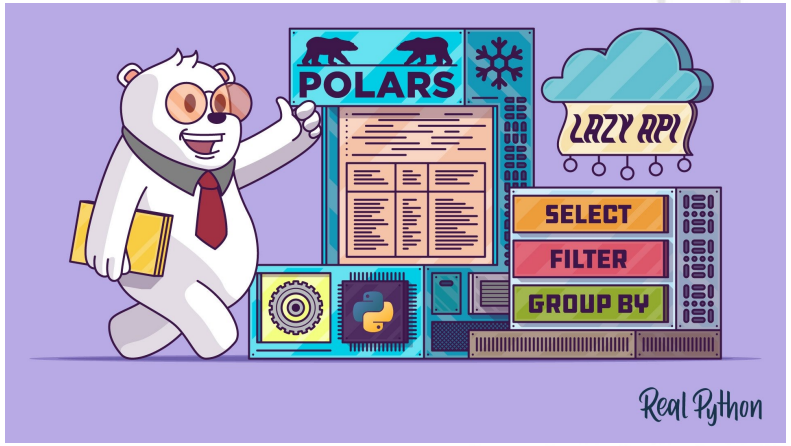
Pearson

Polars

Poll Question

- Which programming language polars is built on? (single selection)
  - R
  - Java
  - Rust
  - Javascript

Why Polars is efficient for large datasets



- Polars is fast and efficient for large datasets mainly due to:
  - ***Rust advantage****:* compiles code without an interpreter, low resource usage
  - ***Memory usage***: efficient memory allocation less copying of dataframes
  - ***Parallel execution***: utilizes all CPU cores to process multiple instruction simultaneously
  - ***Lazy evaluation***: operations are computed only when necessary
  - Polars has been specifically designed to handle large datasets efficiently. With its lazy evaluation strategy and parallel execution capabilities, Polars excels at processing substantial amounts of data swiftly

Pearson

Why Polars is efficient for large datasets

- **Polars** is a Python library for data manipulation that utilizes **Apache Arrow** as its underlying data structure.
- With Polars, users can efficiently work with large datasets in memory using tools like filtering, aggregation, grouping, and manipulation.
- **Apache Arrow** is designed for efficient in-memory processing and transferring data between different systems and languages. Its columnar data format enables efficient data processing than rows, while support for zero-copy memory allocation ensures that Polars can handle very large datasets with minimal memory usage. It saves the CPU from copying data from one memory to another.

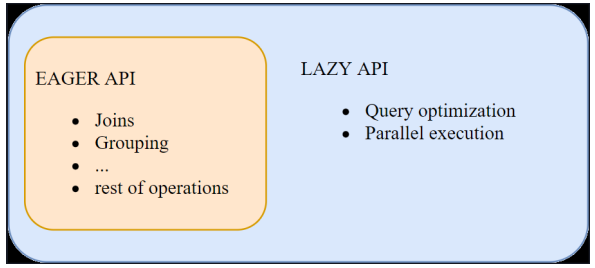
Pearson

What is Polars Query Optimization claim

- ***Query Optimization*:** Polars processes code in either ***lazy*** or ***eager*** execution. Pandas uses eager execution carrying out operations in the order of the written code. Lazy strategy evaluates all the operations and maps out the most efficient way to process:
- The following code finds the mean of column Number1 for each of the categories “A” and “B” in Category:

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

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



Pearson

Polars

How Polars handles larger than RAM data

- Polars uses streaming mode for larger than memory datasets on a single machine
- Streaming mode uses batches of data rather than all at once like a for loop processing each batch in turn. Streaming occurs in lazy mode by using streaming=True
- Polars can handle some larger-than-memory data even without streaming
- Polars also uses predicate pushdown to filter dataframes without reading all data into memory
- Predicate pushdown applies filter as early as possible at scan level to filter data at the source, reducing the amount of data transmitted and processed

Pearson

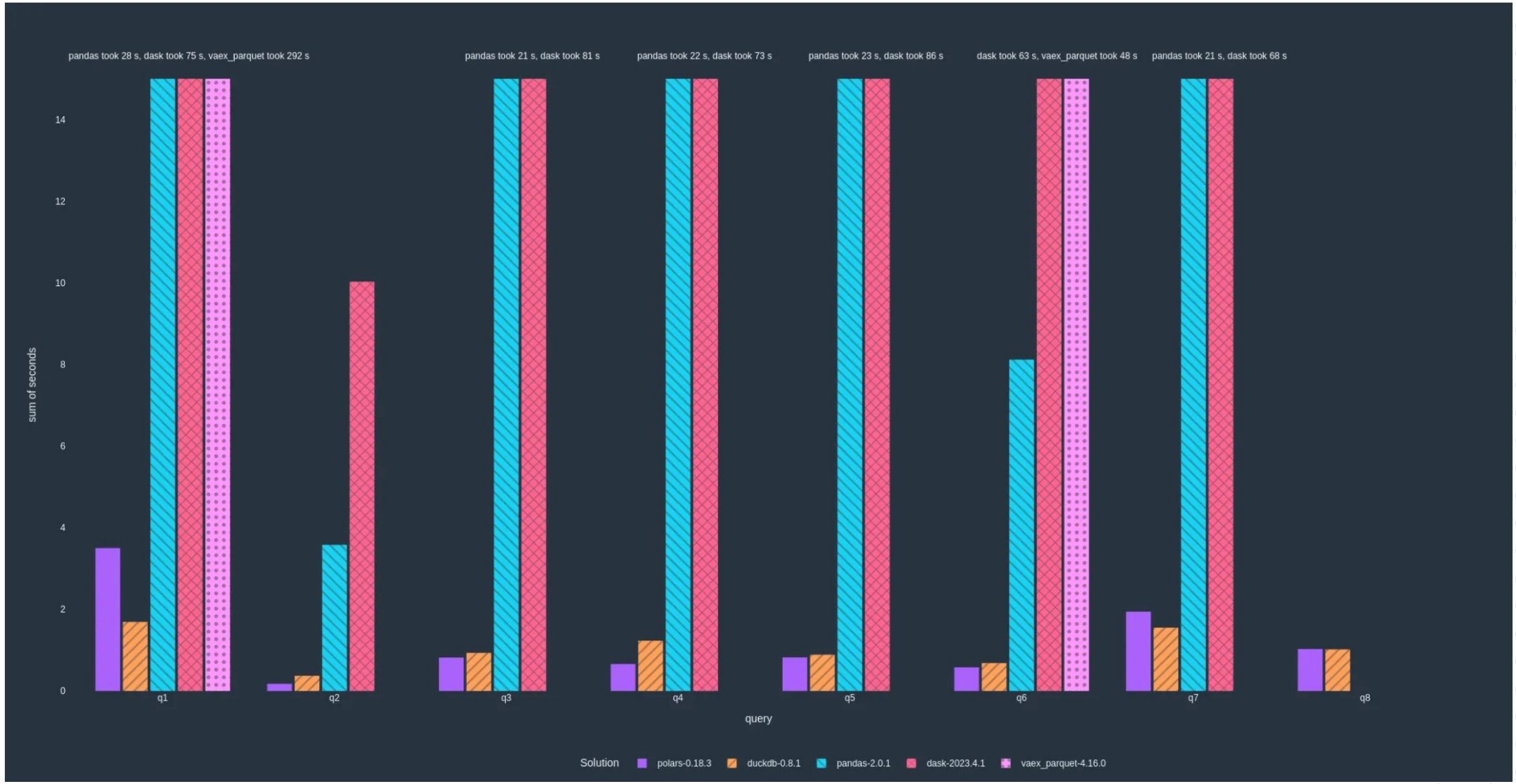
How Polars handles larger than RAM data

- Polars also offers a Spark-like Lazy API for even better performance. The scan_csv command unlike the read_csv command lazily loads the file instead of reading the entire file into memory at once:

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

df = q.collect()
```

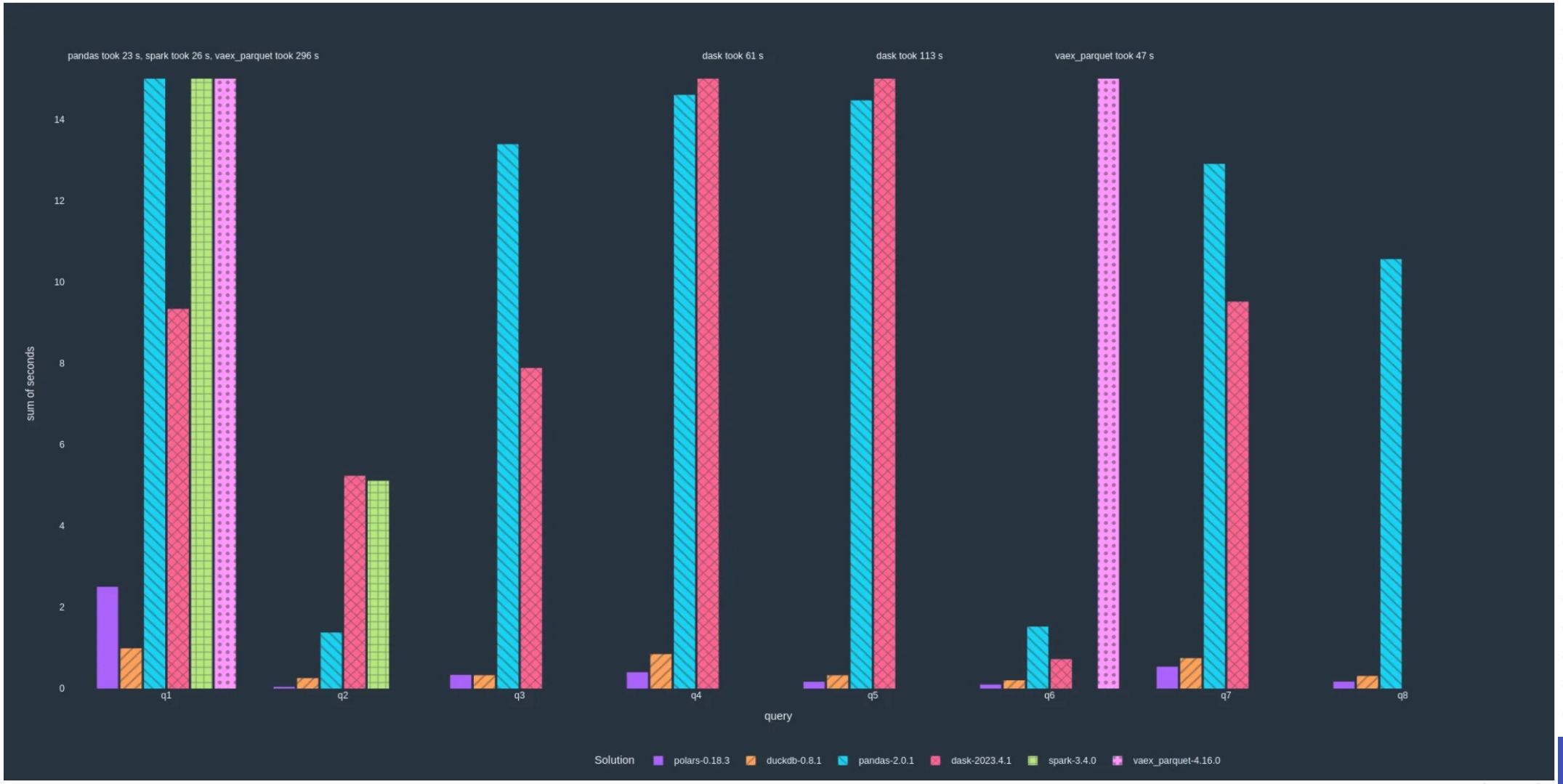
- In the above example, Polars applies predicate pushdown by filtering down to only the records where the sepal_length is greater than 5. It applies projection pushdown by only selecting the columns that are needed. As a result, only a subset of the data is loaded into memory.

Results of reading parquet (lower is better)



https://pola.rs/posts/benchmarks/

Results of in-memory data (lower is better)



Pearson

https://pola.rs/posts/benchmarks/

Polars vs Pandas speed comparison

- Polars is fast for big data based on Rust advantage
- Finding sum of 10 million records:

```
import pandas as pd
import numpy as np


# create random DataFrame with 10 million rows and 2 columns
pd_df = pd.DataFrame(np.random.rand(10000000, 2), columns=['a',
'b'])


# compute sum of each column using Pandas
%timeit pd_df.sum()
```

```
472 ms ± 6.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```


Polars vs Pandas speed comparison

```
import polars as p
import numpy as np


# create random DataFrame with 10 million rows and 2 columns
pl_df = pl.DataFrame({
    'a': np.random.rand(10000000),
    'b': np.random.rand(10000000)})


# compute sum of each column using Polar
%timeit pl_df.sum()
```

```
14.3 ms ± 2.27 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Pearson

Polars vs Pandas speed comparison

- It takes pandas *472 ms* to sum 10mil observations; that's *± 6.5 ms per loop* that is only two features. The real data often contain hundreds of features called Big Data. The time will increase exponentially once the *sum* function is replaced with a complex function or the datatype to string/object type
- It takes polars 14.3 ms to sum 10mil observations; that's ± 2.27 ms per loop!
- Polars is **34 times** faster than pandas for the sum function
- Sigmoid function applied to the observations:

```
from math import exp as
def sig(z):
  return (1/(1+e(-1*z)))e
```

Pearson

Polars vs Pandas speed comparison

- ***Pandas***

```
%timeit pd_df.apply(lambda a: sig(a[0]+a[1]/100), axis=1)
```

```
1min 17s ± 239 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- ***Polars***

```
%timeit pl_df.apply(lambda a: sig(a[0]+a[1]/100))
```

```
7.83 s ± 651 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- Polars is 10 times faster than Pandas for apply function
- Where it takes Polars 0.13min to run the sigma function on 10 million observations Pandas take 1.5min to do that same. Polars ran the ‘apply()’ function 10 times faster Pandas.

Performance comparison of Polars with Pandas

- Following operations performed on a large dataset with 1 million rows:
- ***Import speed***

```
In [18]: %timeit _ = pd.read_csv('large_dataset.csv')
         632 ms ± 18.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [19]: %timeit _ = pl.read_csv('large_dataset.csv')
         72.8 ms ± 1.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

- Polars import is 90 times faster than pandas import which is useful when handling large datasets. We can also use lazy load for even faster load and lazy evaluations:

```
In [145]: %timeit pl.scan_csv('large_dataset.csv')
          135 µs ± 1.17 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```


Poll Question

- In which scenario polars is faster than pandas? (single selection)
  - Data visualization
  - Machine Learning
  - Handling large datasets
  - Cloud computing

Performance comparison of Polars with Pandas

- ***Apply Function:***

```
In [92]: # with pandas
         %timeit df.apply(lambda c: np.sqrt(sum(c)), axis=1)

         6.77 s ± 152 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [102]: #with polars
          %timeit df.apply(lambda c: np.sqrt(sum(c)))

          1.7 s ± 27.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- A similar query can be further made faster by using the expression system provided by Polars

```
In [148]: # with polars
          %timeit df.select([np.sqrt(pl.all().sum())])

          382 µs ± 2.56 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Pearson

Performance comparison of Polars with Pandas

- ***Using groupby and aggregation:***

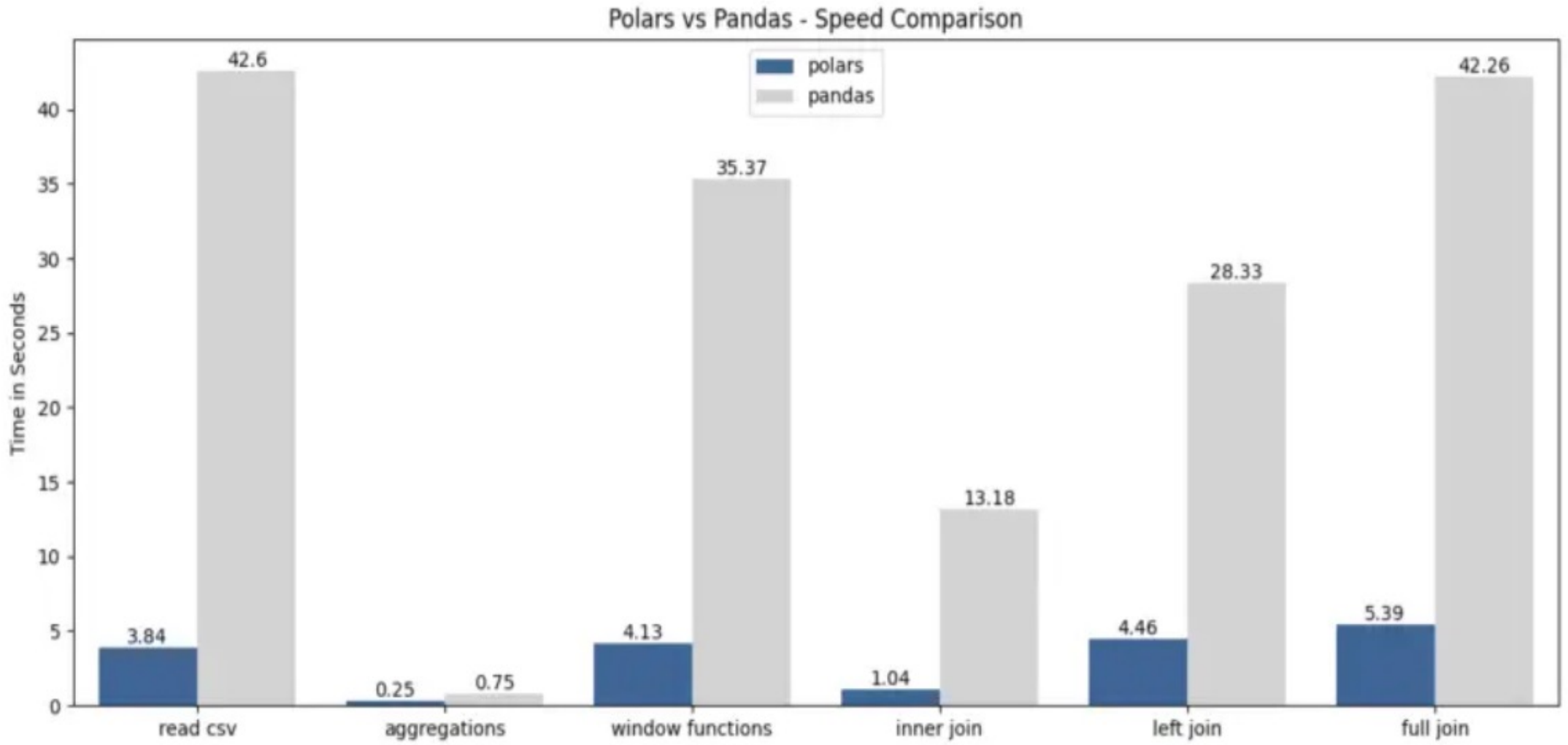
```
In [143]: # with pandas
          %timeit df.groupby(['col1']).agg(pd.isnull).sum()

          7.55 s ± 184 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [139]: #with polars
          %timeit df.groupby(['col1']).agg(pl.all().is_nan()).sum()

          13.3 ms ± 48.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

- Polars groupby uses the lazy evaluation system that pushes the query into the query engine, optimizes it, and caches the intermediate results. Finally, the query is eagerly evaluated and then the result is produced.
- This is fast compared to the very slow pandas groupby aggregation method. Under the hood, the query/statement translate from **[df.query([‘col1’]).agg([pl.all().is_nan()).sum()]** to

[df.lazy().groupby([‘col1’]).agg(pl.all().is_nan()).collect()]

Pearson

Performance comparison of Polars with Pandas



Pearson

Polars Optimizations

- If you use Polars' lazy API, Polars will run several optimizations. Some of them are executed up front, others are determined just in time as the materialized data comes in

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 group by strategy.	0/n times; dependent on query

Polars syntax & API

	Pandas	Polars
Filtering	filtered_df = df[df['column'] > 10]	filtered_df = df.filter(pl.col('column') > 10)
Renaming Columns	df = df.rename(columns= {'old_name': 'new_name'})	df = df.alias({'old_name': 'new_name'})
Grouping	grouped_df = df.groupby('column').agg({'column2': 'sum'}) grouped_df = df.groupby('column') ['column2'].sum().reset_index()	grouped_df = df.groupby('column').agg({'column2': pl.sum('column2')})
Sorting	sorted_df = df.sort_values(by='column', ascending=False)	sorted_df = df.sort('column', reverse=True)
Chaining	df = df[df['age'] > 18].groupby('gender').mean()	df = df.filter(pl.col('age') > 18) & df.groupby('gender').mean()
Column Selection	df['age']	pl.col('age')
Casting	df['age'] = df['age'].astype(float)	df = df.with_column(df['age'].cast(pl.Float64))
Rolling Average	df['rolling_average'] = df['value'].rolling(window=3).mean()	df = df.with_column(df['value'].rolling(3).mean().alias('rolling_average'))

https://www.einblick.ai/tools/what-is-polars-python-compare-pandas/

Pearson

Polars syntax & API

```pandas
# Pandas
import pandas as pd

# read csv file
df = pd.read_csv("file.csv")

# get the shape of a dataframe
df.shape

# drop columns
df.drop(columns=["col1", "col2"])


# write to a csv file
df.to_csv("file.csv")
```

```polars
# Polars
import polars as pl


df = pl.read_csv("file.csv")


df.shape


df.drop(columns=["col1", "col2"])


df.write_csv("file.csv")
```

Polars syntax & API

```python
# Pandas
df["new_col"] = "pandas"
```

```python
# Polars
df.with_column(pl.lit("polars")
  .alias("new_col"))
```

```python
# filtering rows
df.loc[df.flower.isin(["rose", "tulip"])]
```

```python
df.filter(pl.col("flower")
  .is_in(["rose", "tulip"]))
```

```python
# select subset of columns
df[["first_name", "last_name"]]
```

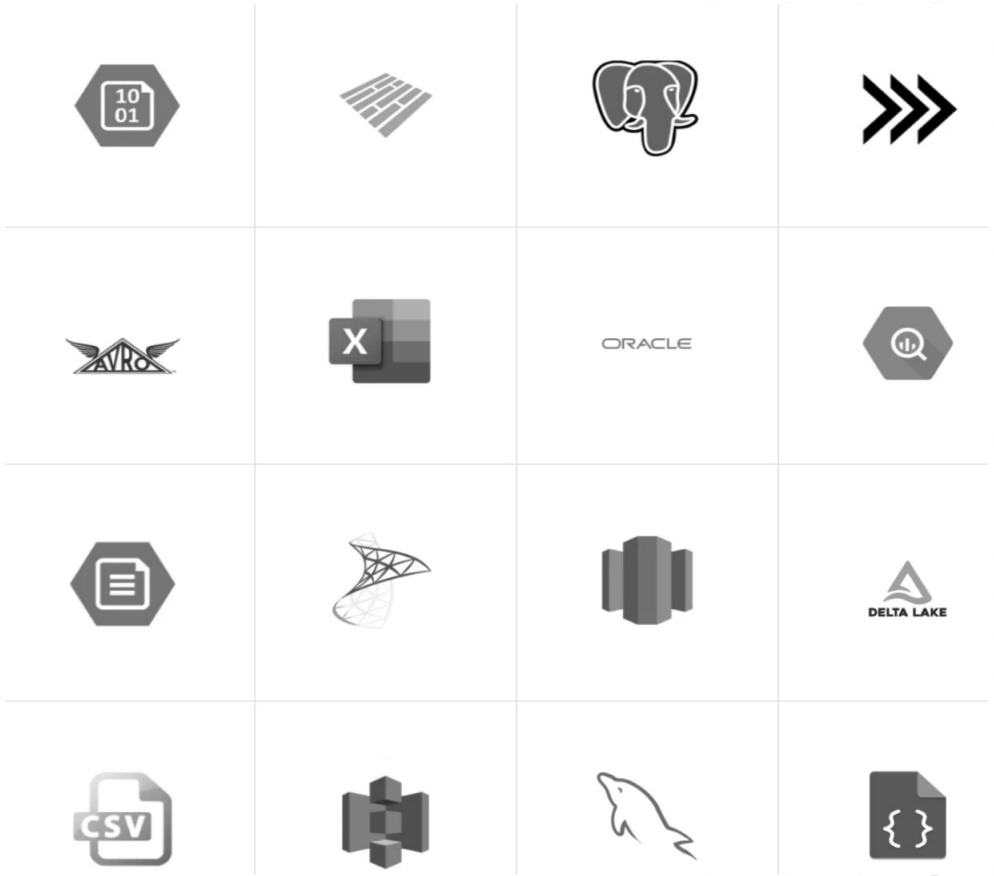
```python
df.select(["first_name", "last_name"])
```

```python
# conditional filtering
df["adult"] = df["age"].apply(lambda x:
     False if x < 18 else True)
```

```python
df.with_columns(
    pl.when(pl.col("age") < 18)
      .then(pl.lit(False))
      .otherwise(pl.lit(True))
      .alias("adult"))
```

Polars data support

- Polars support reading and writing to all common data formats that we see in Pandas too
- This allows easy integration of Polars into the existing data stack:
  - Text: csv & json
  - Binary: parquet, delta lake, avro & excel
  - IPC: feather, arrow
  - Databases: mysQL, postgres, sql server, sqlite, redshift & oracle
  - Cloud Storage: s3, azure blob & azure File



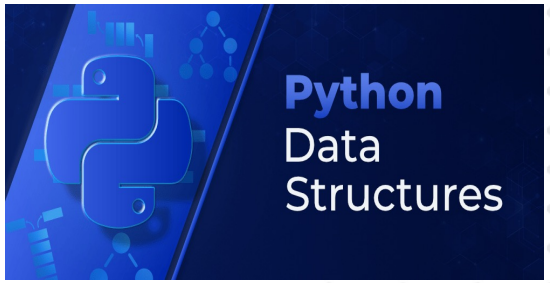
Pearson

Polars Data Structures

- The core base data structures of Polars are series & dataframe
- Series is a 1-dimensional data structure with all elements having similar datatype:

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

```
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   │
└─────────┴─────────────────────┴───────┘
```



- Dataframe is a 2-dimensional data structure and a collection of series. Group By, Join, pivot are some of the operations that can be performed

Polar Contexts

- Polars developed its own Domain Specific Language(DSL) for data transformation
- The two core components of DSL are ***Contexts & Expressions***
- Context refers to the context in which an expression is evaluated
- There are 3 main contexts:
  - Selection df.select(...), df.with_columns(...)
  - Filtering df.filter()
  - Group by / Aggregation df.group_by(...).agg(...)

Pearson

Polars

Polar Contexts

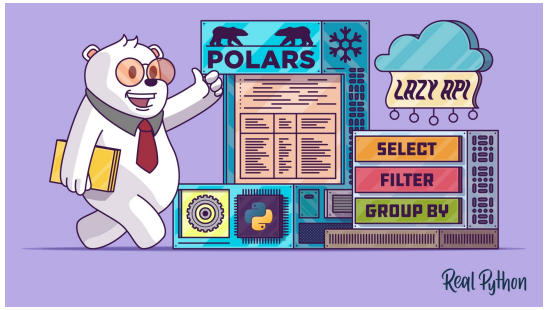
```
df = pl.DataFrame(
    {
        "nrs": [1, 2, 3, None, 5],
        "names": ["foo", "ham", "spam", "egg", None],
        "random": np.random.rand(5),
        "groups": ["A", "A", "B", "C", "B"],
    }
)
print(df)
```

```
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

Selection:

The selection context applies expression on columns. A select may produce new columns that are aggregations, combinations of expressions, or literals



Pearson

Polars

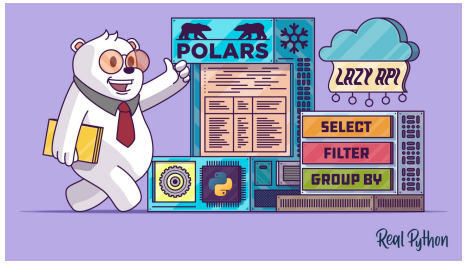
Polar Selection Context

```
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)
```

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

The selection context is very powerful and allows to evaluate arbitrary expressions independent of (and in parallel to) each other.



Pearson

Polars

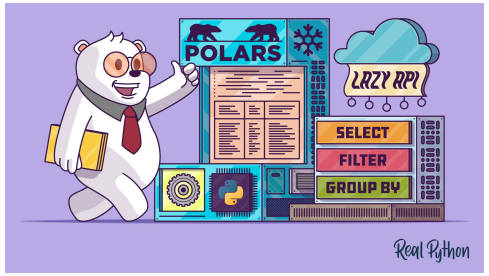
Polar Selection Context

- Similar to the select statement, the **with_columns** statement also enters into the selection context. The main difference between **with_columns** and **select** is that with_columns retains the original columns and adds new ones, whereas select drops the original columns.

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



Pearson

Polars

Polars Filtering Context

- The filtering context filters a DataFrame based on one or more expressions that evaluate to the Boolean data type

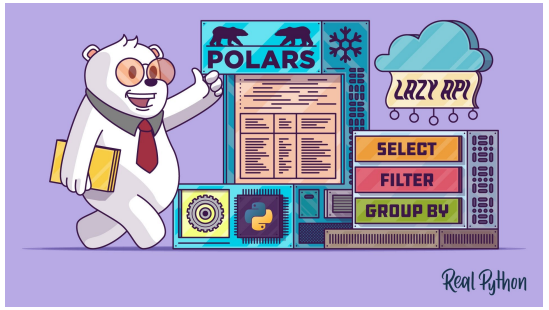
```
out = df.filter(pl.col("nrs") > 2)
print(out)
```

```
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

```
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



Pearson

Polars

Polars Group by / Aggregation Context

- In the group_by context, expressions work on groups and thus may yield results of any length (a group may have many members)

```
out = df.group_by("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().name.suffix("_sum"),
    pl.col("names").reverse().alias("reversed names"),
)
print(out)
```

```
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

```
shape: (3, 5)
```

groups	nrs	count	random_sum	reversed names
str	i64	u32	f64	list[str]
A	3	2	0.894213	["ham", "foo"]
C	0	1	0.533739	["egg"]
B	8	2	0.263315	[null, "spam"]

Polars 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

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

- 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.
- All expressions run in parallel separately and within an expression there may be more parallelization going on.

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

Polars Lazy/Eager API

- Polars support two modes of operations: ***lazy & 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 execution to last minute can have significant performance advantages that is why the Lazy API is preferred in most cases

```
df = pl.read_csv("docs/data/iris.csv")
df_small = df.filter(pl.col("sepal_length") > 5)
df_agg = df_small.group_by("species").agg(pl.col("sepal_width").mean())
print(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 which can be wasteful as we may load extra unused data

Pearson

Polars

Polars Lazy/Eager API

- In lazy API we wait on execution until all steps are defined. The query planner performs these optimizations:
  - ***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)

```
q = (
    pl.scan_csv("docs/data/iris.csv")
    .filter(pl.col("sepal_length") > 5)
    .group_by("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.

Poll Question

- Which operation mode that polars utilizes defer execution until needed? (single selection)

  - Fast
  - Lazy
  - Eager
  - Slow

When to use Lazy vs Eager API

- 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
- In many cases the eager API is 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
- Polars ***lazy API*** one of the most powerful features. Lazy API allows to specify a sequence of operations without immediately running them. Instead, these operations are saved as a computational graph and run when necessary.
- This allows Polars to optimize queries before execution, catch schema errors before the data is processed, and perform memory-efficient queries on datasets that don’t fit into memory

Pearson

Polars Streaming mode

- ***Lazy API*** allows queries to be executed in a ***streaming mode***
- This means that Instead of processing data all-at-once, Polars can execute a query in batches allowing to ***process datasets that are larger-than-memory***
- Streaming mode is still in development stages & not all lazy operations support it. Streaming is supported in:
  - filter, slice, head, tail, with_columns, select
  - group_by, join, unique, sort
  - explode, melt, scan_csv, scan_parquet, scan_ipc

```
q1 = (
    pl.scan_csv("docs/data/iris.csv")
    .filter(pl.col("sepal_length") > 5)
    .group_by("species")
    .agg(pl.col("sepal_width").mean())
)
df = q1.collect(streaming=True)
```

Pearson

Polars Streaming mode

- Polars can handle ***larger-than-memory datasets*** with its streaming mode. In this mode Polars processes all data in ***batches*** rather than all at once.
- It is worth keeping streaming switched on when working with larger datasets - particularly if you are building pipelines that you want to be ready to larger datasets in the future.
- **Simple example of streaming and non-streaming:**
- To work in streaming mode use ***streaming=True*** to collect when we evaluate a query.
- Let's create DataFrame with 1m rows, 100 floating point columns and integer ID column.
- We then do a groupby on the id column and take the mean of the remaining columns.
- We execute the query in streaming mode with the streaming=True argument

*If we compare this query with streaming=True and streaming=False (the default): **75 ms** for streaming and **120 ms** for non-streaming. For Pandas this takes about **330 ms** for comparison.*

```
(
    df
    .lazy()
    .groupby('id')
    .agg(
        pl.all().mean()
    )
    .collect(
        streaming=True
    )
)
```

```
import polars as pl
import numpy as np

N = 1_000_000
K = 100

df = (
    pl.DataFrame(
        np.random.standard_normal((N,K))
    )
    # Add an ID column
    .hstack(
        pl.DataFrame(
            np.random.randint(0,9,(N,1)
            )
        )
        .rename(
            {'column_0':'id'}
            )
        )
)
```

Pearson

Pandas 2.0 vs Polars

- Pandas 2.0 includes range of new features towards making pandas faster & memory efficient to handle complex tasks
- Pandas 2.0 includes the more robust memory management ***PyArrow***
- Users can choose btw the traditional NumPy backend or the brand-new PyArrow backend
- PyArrow is a Python interface for handling large datasets using arrow memory structures
- Another features include ***copy-on-write*** to improve performance by saving memory resources.
- This keeps a lazy copy of Pandas objects, such as series & dataframe rather than cached copied of objects. These methods ***return views when Copy-on-Write is enabled***, which provides significant performance improvement compared to the regular execution

```
pd.read_csv(my_file, engine='pyarrow')
```

```
pd.set_option("mode.copy_on_write", True)

pd.options.mode.copy_on_write = True
```

Pandas 2.0 vs Polars performance comparison

Test	Pandas 2.0	Polars	Winner	Syntax Similarity
Import Test	Reading all rows and then filtering unwanted rows using .query() method	Directly reads only the rows of interest using filter() method	Polars (better performance)	Similar, but with differences due to underlying implementation (.query() vs filter())
Group By Test	Uses .groupby() for aggregating operations such as mean and median of sales by office and month	Same as pandas, but with better performance	Polars (better performance)	Similar
Rolling Statistics Test	Uses rolling() method combined with other methods, such as mean(), for rolling mean calculations	Uses rolling_mean() method for rolling mean calculations	Polars (better performance)	Similar, but with slight differences (.rolling().mean() vs rolling_mean())
Sampling Test	Uses numpy for bootstrap sampling operations	Uses built-in sample() method for bootstrap sampling operations	Polars (nearly x5 times faster)	Different, pandas relies on numpy while polars has built-in methods
Compound Manipulations Test	Performs join operation with another dataset and then sorts and selects data	Performs same tasks but more efficiently	Polars (significantly faster)	Similar, but with slight differences (merge().query().sort_values().head() vs join().filter().sort().head().select())

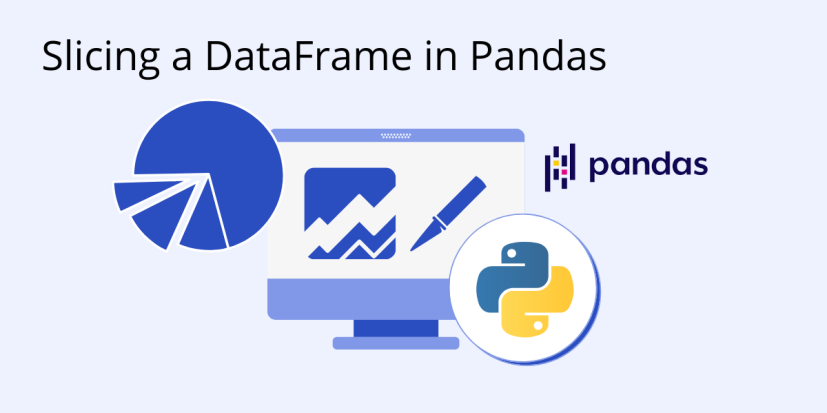
https://www.datacamp.com/tutorial/high-performance-data-manipulation-in-python-pandas2-vs-polars

What's new in Pandas 2.2

- Pandas 2.2.0 released on Jan 19,2024
- **Improved PyArrow Support:**
  - Lists & structs were NumPy object dtype before
  - Now Arrow type backend provides much easier way for lists & structs
  - This is a series that contains a dictionary in every row.

Previously, this was only possible with numpy object dtype and

accessing elements from these rows required iterating over them.




```
import pyarrow as pa

series = pd.Series(
    [
        {"project": "pandas", "version": "2.2.0"},
        {"project": "numpy", "version": "1.25.2"},
        {"project": "pyarrow", "version": "13.0.0"},
    ],
    dtype=pd.ArrowDtype(
        pa.struct([
            ("project", pa.string()),
            ("version", pa.string()),
        ])
    ),
)
```

Pearson

What's new in Pandas 2.2

- The ***struct*** accessor now enables direct access to certain attributes:

```
import pyarrow as pa

series = pd.Series(
    [
        {"project": "pandas", "version": "2.2.0"},
        {"project": "numpy", "version": "1.25.2"},
        {"project": "pyarrow", "version": "13.0.0"},
    ],
    dtype=pd.ArrowDtype(
        pa.struct([
            ("project", pa.string()),
            ("version", pa.string()),
        ])
    ),
)
```

```
series.struct.field("project")

0      pandas
1       numpy
2    pyarrow
Name: project, dtype: string[pyarrow]
```

Pearson

What's new in Pandas 2.2

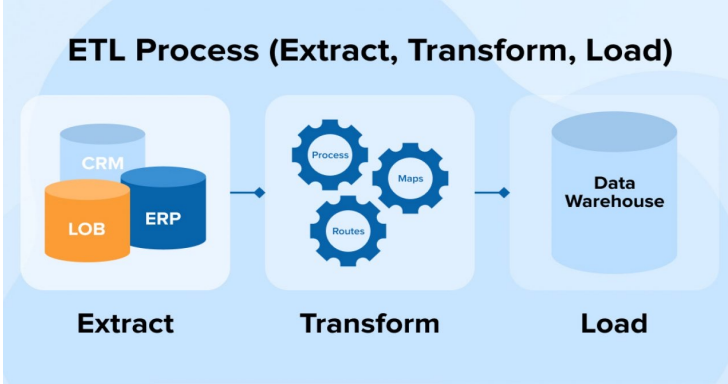
- **Integrating the Apache ADBC Driver:**
  - Pandas used sqlalchemy to read data from sql database but was very slow
  - Alchemy reads data row-wise while pandas has columnar layout which makes reading & moving data into a dataframe slower than necessary
  - The ADBC driver from the Apache Arrow project enables users to read data in a columnar layout, which brings huge performance improvements
  - The ADBC Driver currently supports ***postgres and sqlite.***
- **case_when:**
  - Pandas 2.2 adds a new ***case_when*** method, that is defined on a Series. It operates similarly to what sql does

```
df = pd.DataFrame(dict(a=[1, 2, 3], b=[4, 5, 6]))

default=pd.Series('default', index=df.index)
default.case_when(
    caselist=[
        (df.a == 1, 'first'),
        (df.a.gt(1) & df.b.eq(5), 'second'),
    ],
)
```

Pearson

Polars for ETL

- ***Extract, transform, and load (ETL)*** is a process of combining data from multiple sources into a large, central repository called a data warehouse. ETL uses a set of business rules to clean and organize raw data and prepare it for storage, data analytics, and machine learning (ML).
- Organizations today have both structured and unstructured data from various sources including:
  - Customer data from online payment and customer relationship management (CRM) systems
  - Inventory and operations data from vendor systems
  - Sensor data from Internet of Things (IoT) devices
  - Marketing data from social media and customer feedback
  - Employee data from internal human resources systems



- *Polars speed, simplicity, fast data ingestion & no memory constraints make it an ideal choice for ETL*

Pearson

When should we stick with Pandas

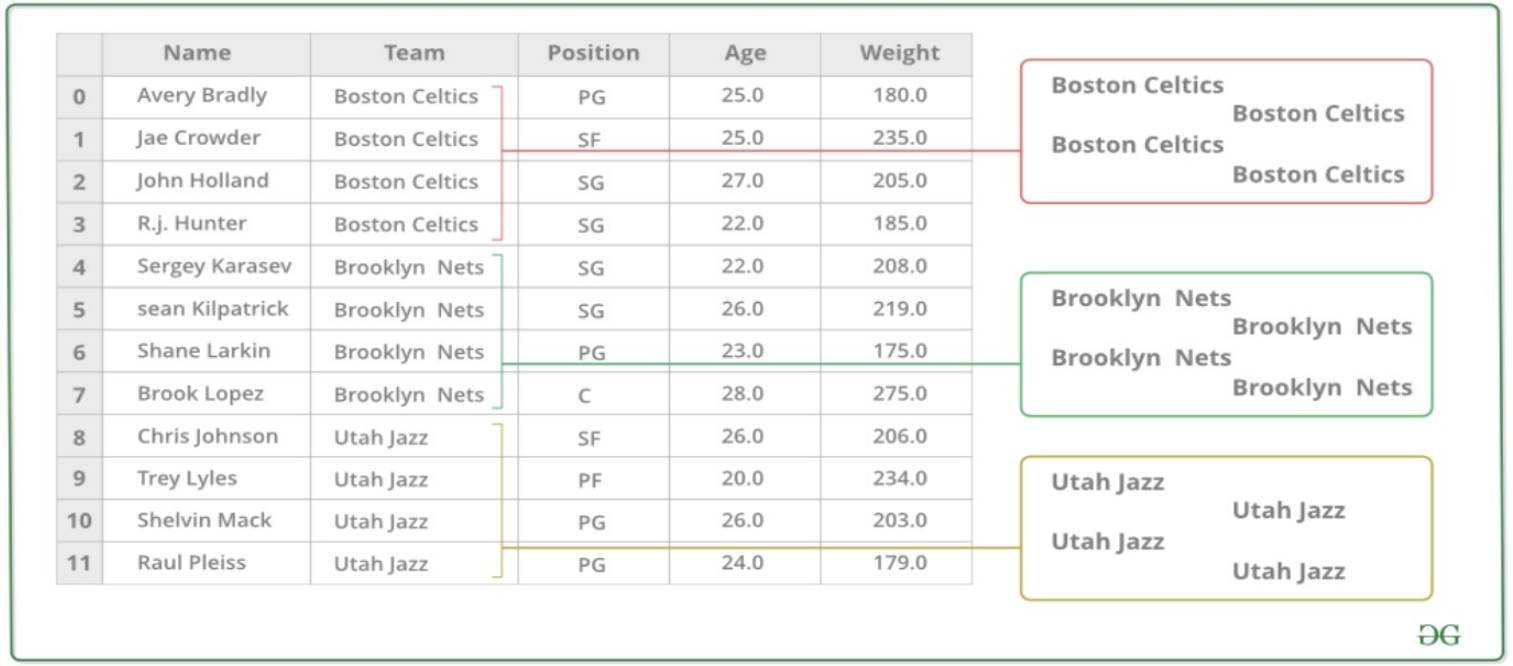
- Pandas is good for small to medium scale projects and is still the best package in many scenarios like data exploration & machine learning because:
- ***Interoperability*****:** Polars has extensive interoperability with packages using Arrow but not compatible with python data visualization & machine learning libraries like scikit-learn & pytorch. Plotly does support polars
- ***Tooling*****:** Polars proves to be faster for data manipulation. However, pandas is still the best for data exploration and machine learning tasks. As the Python ecosystem evolves, the compatibility gap between Polars and other libraries may narrow, making Polars an even more compelling option in the future

How can we improve data mining performance

- **Reducing copying with copying-on-write:**
  - In traditional data manipulation tools, every data operation leads to creation of a new copy of the entire dataframe. This behavior can be problematic for memory resources when dealing with large datasets.
  - ***Copy-on-write*** is most common technique to solve this issue. It is a resource-management technique that allows to efficiently implement a "duplicate" or "copy" operation on modifiable resources. That means that if a dataset is duplicated but not (entirely) modified, it is not necessary to create a new resource. Instead, the memory resources can be shared between copy and original.

How can we improve data mining performance

- **Parallel Computation:**
  - Pandas is threaded and does not have parallel computing techniques. Parallel computing is done by either multi-core machines or distributed computing (i.e., a group of computers working together in a cluster).
  - ***Example***: To calculate mean age of basketball players grouped by teams, we could use parallel computing to assign calculation of mean of each group to one core or machine instead of calculating them one by one, thereby increasing performance



https://www.datacamp.com/tutorial/high-performance-data-manipulation-in-python-pandas2-vs-polars

How can we improve data mining performance

- **Optimizing data import:**

  Reading large datasets is often time consuming in pandas and polars provides faster alternative.

- Here are some techniques that can help improve performance when reading data files:

  - **Data Chunking:**

  Instead of reading full file, we can break it into little pieces, aka chunks. This technique allows for faster data importing while reducing memory required to store it all simultaneously.

  - **Read only required columns:**

  Instead of reading whole dataset, use a preliminary filter and read only the required columns as polars does

  - **Downcasting data types.**

  Tools like pandas have a default behavior when assigning data types to imported columns. Sometimes, there may be other available types that require less memory without losing information. Downcasting is a process of changing data types into the smallest type that can handle its value

How can we improve data mining performance

- **Lazy Evaluation and Query Planning**
  - The order of operations affects performance. We can reorder some computations and get same answer more efficiently.
  - There are two strategies to achieve this: ***lazy evaluation*** and ***query planning***. The former refers to process of delaying computations until absolutely necessary. In contrast, the latter is a process of finding most efficient way of performing multiple data manipulations given a set of different queries. To do so, query planning normally involves use of a query optimizer.
- **Out-of-Core Processing**

Pandas store entire datasets in memory which is not efficient when working with large datasets. Out-of-core processing refers to external memory (e.g., an external disk) that stores data that couldn't fit in core memory. It processes externally stored data in chunks and gives the same results.

Pearson

What are Some High-Performance pandas Alternatives?

- Besides Polars some high performance pandas alternatives are:

  - **Koalas**. A pandas API built on top of PySpark. If you use Spark, you should consider this tool.
  - **Vaex**. A pandas API for out-of-memory computation for analyzing big data at billion rows per second.
  - **Modin**. A pandas API for parallel programming, based on Dask or Ray frameworks for big data projects. If you use Dask or Ray, Modin is a great resource.
  - **cuDF**. Part of the RAPIDS project, cuDF is a pandas-like API for GPU computation that relies on NVIDIA GPUs or other parts of RAPIDS to perform high-speed data manipulation.

Poll Question

- How does polars process larger than memory datasets? (multiple selection)
  - Non-streaming mode
  - Streaming mode
  - Dynamic threshold
  - Batches of data
  - Multi-processing
  - Multi-threading

Why choose Polars

- Though Pandas is most widely used it has limitations when dealing with large datasets. The core mission of **Polars** is to offer a lightning-fast dataframe library that addresses the problems of pandas.
- Written in ***Rusk*** (a powerful programming language that provides C/C++ performance and allows to fully control performance critical parts in a query engine), polars integrates state-of-the-art methodologies to increase performance, designed to:
  - Enhance parallel computing using all available cores & optimize queries to reduce unneeded work/memory allocations using lazy evaluation and query planning.
  - Improve storage performance by using ***copy-on-write*** semantics and ***Apache Arrow***.
  - Handle datasets much larger using its out-of-core data capabilities through its streaming API.
- An important feature of polars is that it has a pretty similar syntax to pandas and shares the same building blocks, such as series and dataframes.

Polars code execution

```
q1 = (
    pl.scan_csv("docs/data/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
)
```

- In the code above on the Reddit CSV, 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:**

```
q4 = (
    pl.scan_csv(f"docs/data/reddit.csv")
    .with_columns(pl.col("name").str.to_uppercase())
    .filter(pl.col("comment_karma") > 0)
    .collect()
)
```

Pearson

Polars code execution

shape: (14_029, 6)

id	name	created_utc	updated_on	comment_karma
---	---	---	---	---
i64	str	i64	i64	i64
6	TAOJIANLONG_JASONBROKEN	1397113510	1536527864	4
17	SSAIG_JASONBROKEN	1397113544	1536527864	1
19	FDBVFDSSDGFDS_JASONBROKEN	1397113552	1536527864	3
37	IHATEWHOWEARE_JASONBROKEN	1397113636	1536527864	61
…	…	…	…	…
1229384	DSFOX	1163177415	1536497412	44411
1229459	NEOCARTY	1163177859	1536533090	40
1229587	TEHSMA	1163178847	1536497412	14794
1229621	JEREMYLOW	1163179075	1536497412	411

- 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 data as one batch. This means that all the data has to fit into available memory at the point of peak memory usage in query.

Pearson

Polars code execution

- **Execution on larger-than-memory data:**
  - If data requires more memory than available Polars processes the data in ***batches*** using ***streaming mode***. To use streaming mode simply pass the streaming=True argument to 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)
)
```

- **Execution on a partial dataset:**
  - While writing, optimizing or checking query on a large dataset, querying all available data may lead to a slow development process. We can execute query with ***.fetch method***. The .fetch method takes a parameter ***n_rows*** and tries to 'fetch' that number of rows at the data source. Here we "fetch" 100 rows from the source file and apply the predicates.

Polars code execution

```
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	l
i64	str	i64	i64	i64	i
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	1
77768	TINHEADNED	1139665457	1536497404	4434	1

Pearson

Polars Query 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 group by strategy.	0/n times; dependent on query

https://docs.pola.rs/user-guide/lazy/optimizations/

How Polars will make your life easier

01
Easy to use

Write your queries the way they were intended. Polars will determine the most efficient way to execute them using its query optimizer.

02
Embarrassingly parallel

Complete your queries faster! Polars fully utilizes the power of your machine by dividing the workload among the available CPU cores without any additional configuration or serialization overhead.

03
Apache Arrow

Polars utilizes the Apache Arrow memory model allowing you to easily integrate with existing tools in the data landscape. It supports zero-copy data sharing for efficient collaboration.

04
Close to the metal

Polars is written from the ground up, designed close to the machine and without external dependencies. This allows for full control of the ecosystem (API, memory & execution).

05
Written in Rust

The core of Polars is written in Rust, one of the fastest growing programming languages in the world. Rust allows for high performance with fine-grained control over memory.

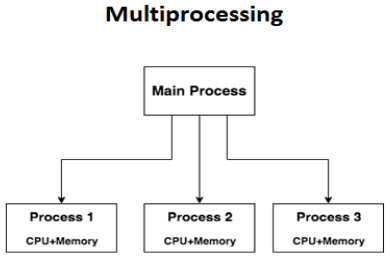
06
Out of core

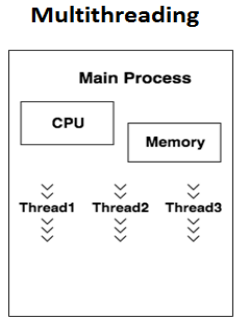
Want to process large data sets that are bigger than your memory? Our streaming API allows you to process your results efficiently, eliminating the need to keep all data in memory.

Multithreading & Parallel Computing

- Multithreading refers to the concurrent execution of more than one sequential set (thread) of instructions.
- Multithreaded programming is programming multiple, concurrent execution threads. These threads could run on a single processor. Or there could be multiple threads running on multiple processor cores.
- Parallel programming is a process of using a set of resources to solve a problem in less time by dividing the work.
- ***Multiprocessing implements parallelism. Multiprocessing allocates separate memory and resources for each program — or process. But multithreading shares the same memory and resources for threads belonging to the same process.***
- Polars uses 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 combined at the end. Another example is aggregating a value within groups using group_by().agg(<expr>), each group can be evaluated separately.

BASIS FOR COMPARISON	MULTIPROCESSING	MULTITHREADING
Basic	Multiprocessing adds CPUs to increase computing power.	Multithreading creates multiple threads of a single process to increase computing power.
Execution	Multiple processes are executed concurrently.	Multiple threads of a single process are executed concurrently.





Pearson

Polars in Big Data Analysis, Machine Learning & Data Science

- ***Polars & Big Data Analysis:*** In big data analysis, processing large datasets efficiently is crucial. Polars, with its fast execution speed and memory efficiency, is an excellent tool for such tasks. Its ability to handle large datasets and perform complex operations efficiently makes it a preferred choice for big data analysis.
- ***Polars in Machine Learning:*** Machine learning involves working with large datasets and requires efficient data manipulation for feature extraction, data cleaning, and preprocessing. Polars' efficient dataframe manipulation capabilities can significantly speed up these processes, making it a useful tool for machine learning practitioners.
- ***Polars for Data Science:*** Data science involves extracting insights from data, which often requires efficient data manipulation. Polars, with its wide range of features for dataframe manipulation, can help data scientists clean, transform, and analyze their data more efficiently.

Pearson

https://ioflood.com/blog/polars/

Conclusion

- Use pandas and polars together to make data processing tasks more efficient, swap out the parts that are slow with polars and leave the rest of the code in pandas. It's easy to switch between pandas and polars
- You can convert a pandas dataframe to polars: df_pl = pl.from_pandas(df_pd) and vice versa: df_pd = pl.to_pandas(). This facilitates refactoring existing code and to use third party libraries that only support Pandas.
- Polars is a powerful library for handling large datasets, complex data types and offers easy integration with other python libraries such as numpy and pyarrow
- Polars' optimized back end, familiar yet efficient syntax, lazy API, and integration with the python ecosystem make the library stand out among the crowd.
- Polars mission is to provide seamless data processing experience on any scale

Task Completed

When working with big datasets, performance improvement is crucial, Polars takes charge

Pearson

References

- https://pandas.pydata.org/docs/dev/getting_started/overview.html
- https://realpython.com/polars-python/
- https://docs.pola.rs/user-guide/
- https://www.einblick.ai/tools/what-is-polars-python-compare-pandas/
- https://blog.jetbrains.com/dataspell/2023/08/polars-vs-pandas-what-s-the-difference/
- https://towardsdatascience.com/whats-new-in-pandas-2-2-e3afe6f341f5
- https://www.linkedin.com/pulse/polars-vs-pandas-which-one-faster-benchmarking-analysis-saleem/
- https://www.rust-lang.org/
- https://rotational.io/blog/introduction-to-polars/
- https://www.rhosignal.com/posts/polars-dont-fear-streaming/
- https://www.perforce.com/blog/qac/multithreading-parallel-programming-c-cpp
- https://ioflood.com/blog/polars/