ULTIMATE

Python Polars for Data Analytics

Transform Your Large-Scale Data into High-Performance Analytics with Python Polars

Sunny Khilare

First Published: March 2026

Published by: Orange Education Pvt Ltd, AVA®

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,
N1 7AA, United Kingdom

ISBN (PBK): 978-93-49887-35-0

ISBN (E-BOOK): 978-93-49887-33-6

Scan the QR code to explore our entire catalogue

www.orangeava.com

Table of Contents

CHAPTER 1

Introduction to Polars

"Every company has big data in its future and every company will eventually be in the data business."

Thomas H. Davenport

Introduction

Data is a byproduct of human actions, interactions and reactions. The way we experience the world can be expressed through the data we generate. With the advent of the internet, there has been a tremendous increase in data volumes over the years. Data professionals use this data to make data-driven decisions and enhance their products, strategy, and operations.

De facto industry-leading libraries such as Pandas have helped professionals analyze the data. With an increase in the data volume, the limitations of such libraries are becoming apparent. This is where Polars comes in. It is a lightning-fast dataframe library which can handle larger datasets and demanding workflows. Polars is written in Rust and optimized for parallelism. It can handle large amounts of data compared to other traditional dataframe libraries. It can be used to create flexible, scalable and elegant data solutions. Polars is slowly becoming a top choice for professionals who want to take their workflows to the next level.

This book will help you master Polars with ease. Hence, whether you are a data analyst, data scientist, or data engineer, you will learn how to use Polars for data wrangling with large datasets and build scalable workflows.

Structure

In this chapter, we will discuss the following topics:

- Introduction to Polars
- Key Features of Polars

    - Apache Arrow Backend
    - Apache Arrow Data Types
    - Parallelization
    - Query Optimization
- Comparison with Other Dataframe Libraries
- Installation of Polars in Python Environment
    - Our First Dataframe with Polars

Introduction to Polars

Polars started as an open-source project in 2020. The vision for it was to make it an easy-to-use dataframe library which does not compromise performance. Developed using Rust, Polars currently supports four languages, which are Python, Rust, R and Node.js. Polars provides seamless integration with Python, which allows users to work with data from various sources such as CSV, Parquet, and JSON. Whether you are performing simple analysis or building intricate data pipelines using the cloud, Polars is designed to handle large-scale data processing with speed, flexibility, and ease. Polars shines in single-node operations, while existing libraries such as Pandas are single-threaded and utilize only RAM and do not incorporate any DBMS concepts in their operation. Polars overcomes these inefficiencies with parallelism, cache optimization, and query engine.

Key Features of Polars

Now, we will cover the important features of Polars that set it apart from other dataframe libraries.

Apache Arrow Backend

Apache Arrow is an open-source, cross-language development platform designed to accelerate data processing and analytics. It provides a standardized, columnar memory format that optimizes the way data is stored and accessed in memory. This columnar structure is ideal for analytical workloads, as it allows for efficient vectorized execution and better cache locality. This makes data processing faster and more efficient. Arrow is

engineered to handle large-scale data with high performance, which is crucial for modern big data applications.

One of Arrow's key features is its ability to facilitate fast, zero-copy data interchange between different programming languages and systems (Refer to *Figure 1.1*). By providing a common memory layout, Arrow allows data to be shared between frameworks, such as Python, R, Java, and C++, without the need for serialization or deserialization. This eliminates the overhead associated with traditional data transfer methods and results in substantial performance improvements when you are moving data across different systems or languages.

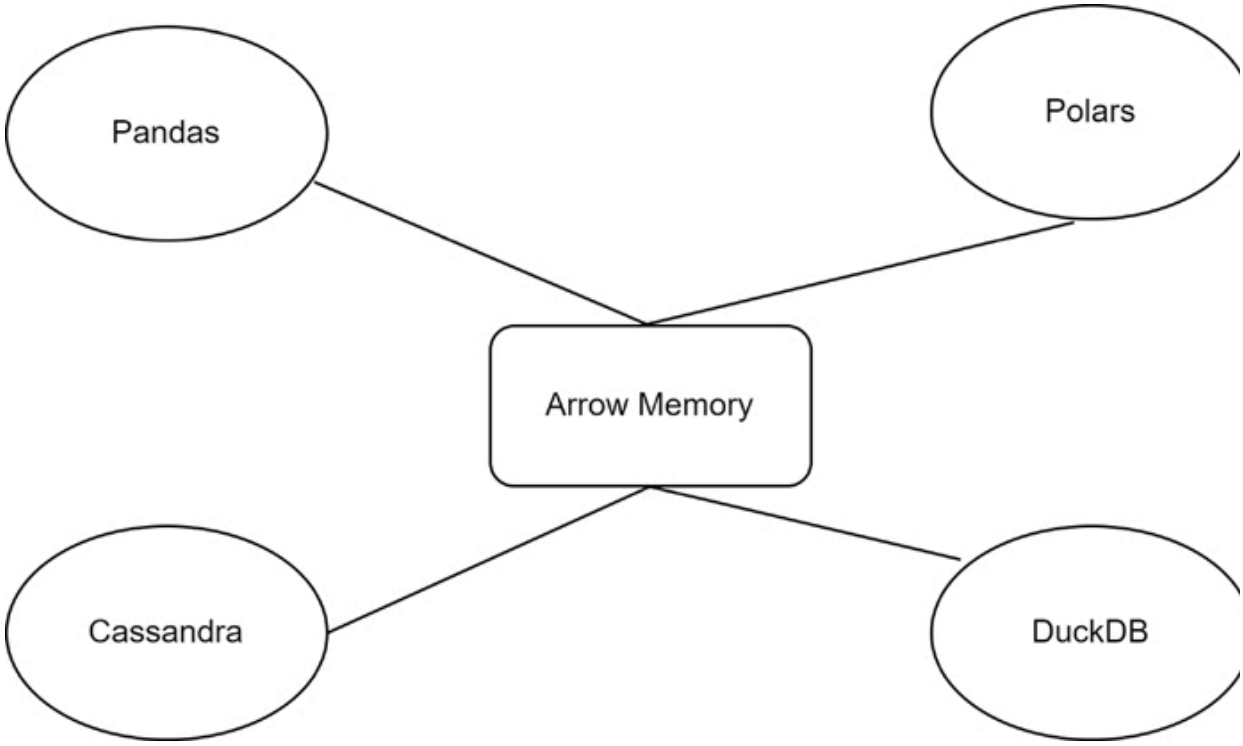

Figure 1.1: *Apache Arrow Zero Copy*

Instead of creating a copy of data and then converting it for a target representation system utilizing Apache Arrow backend can freely transfer data between them.

Arrow's columnar memory format serves as the foundation for several data processing libraries and tools, including Apache Parquet and Polars (Refer *Figure 1.2*). It plays a crucial role in enabling interoperability between various data tools and frameworks, which makes it an essential part of the modern data ecosystem. By providing a unified, efficient memory

representation, Apache Arrow not only speeds up data processing but also simplifies data handling, which empowers data engineers and analysts to work more effectively across multiple platforms.

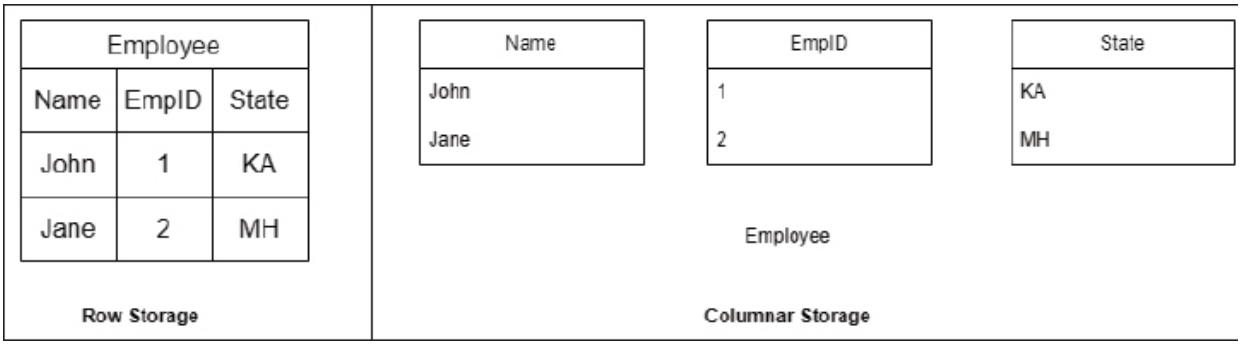

Figure 1.2: *Apache Arrow Columnar Storage*

Columnar storage is better than raw storage for analytical queries because it allows for faster access to specific columns and reduces the need to read unnecessary data. It offers higher compression rates due to the similarity of data within columns, which saves storage space. It excels in read-heavy workloads, especially when only a few columns are queried. Columnar databases enable faster aggregations and calculations by optimizing the data retrieval process. This makes columnar storage ideal for OLAP systems and big data analytics.

Apache Arrow leverages Single Instruction, Multiple Data (SIMD) to optimize data processing, particularly for vectorized operations on large datasets. Its columnar storage format is well-suited for SIMD, as it allows entire columns to be processed simultaneously, rather than individual elements (Refer *Figure 1.3*). This increases performance by enabling faster operations such as filtering and aggregation. Additionally, the columnar layout improves cache utilization and instruction parallelism, which makes Arrow highly efficient for analytical workloads. By taking advantage of SIMD, Arrow accelerates data manipulation, particularly on modern CPUs that are designed to handle vectorized operations.

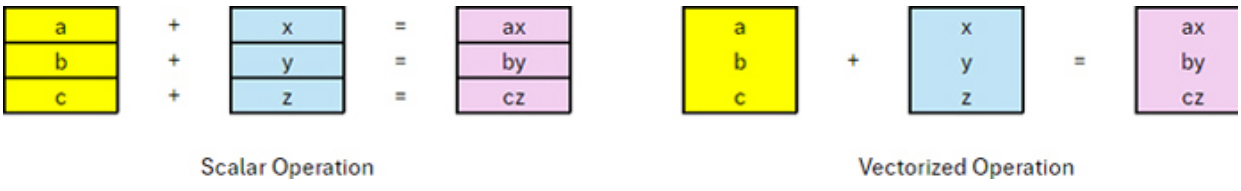

Figure 1.3: *Apache Arrow Vectorized Operations*

Instead of adding elements one by one to each other in a scalar fashion, we add two columns using a single instruction. This results in a vectorized operation.

Apache Arrow Data Types

Apache Arrow provides a rich set of data types tailored for high-performance, in-memory columnar data processing. Its primitive types include integers, floating-point numbers, and booleans, which align closely with traditional data structures while offering precise control over bit widths (for example, int32, float64). Arrow further supports advanced scalar types such as decimal for fixed-point precision, which is often required in financial applications. Its string and binary types enable user to handle text and raw data efficiently and support both variable-length and large-scale storage for datasets with significant size variations. These data types are optimized for analytical tasks and leverage a columnar layout for improved CPU and memory performance.

Numeric Array

In Apache Arrow, numeric arrays are designed with a focus on both performance and clarity, while representing data. Each numeric array contains the actual data values (such as numbers stored as f32 for 32-bit floating-point or u64 for 64-bit unsigned integers) in a data buffer. Alongside this, Arrow always includes a validity buffer, which is a small array of bits. Each bit in the validity buffer indicates whether a corresponding value in the data buffer is valid or missing. This design keeps the memory overhead minimal because only a single bit is used for each value to track its validity.

This approach offers a clear advantage over libraries such as Pandas. In Pandas, missing data is often represented by special values such as NaN for floats, which can be confusing because NaN could mean either "missing data" or "not a number." Arrow, by contrast, makes a clear distinction between valid but undefined values (such as NaN) and genuinely missing data, which ensures more accurate data handling and less ambiguity.

Arrow String Array

The memory layout of an Arrow LargeString array is optimized for performance and efficiency. All the string data is stored in a single continuous block of memory (called the data buffer), which makes it faster to access and process strings because they are stored sequentially. To know where each string starts and ends, Arrow uses an offset array that keeps track of the positions of each string in the data buffer. Additionally, a null-bit buffer is included to indicate if any strings are missing, which saves memory by using only one bit per value.

In contrast, Pandas handles strings as Python objects, which means that each string will be stored separately in memory. This contributes to more memory overhead because each object includes additional metadata about its type. While accessing strings in Pandas, the program jumps around in memory to fetch each string, which can slow things down because it leads to cache misses (when data is not readily available in fast memory).

While Arrow is faster for reading strings due to its sequential layout, it has a drawback. If you filter or select values from an Arrow string array, the entire string data might need to be copied. This is slower and uses more memory. In Pandas, only pointers (references to the string data) are copied, which make such operations faster and less memory-intensive. However, the trade-off is that Pandas' scattered memory layout is less efficient for tasks such as sequential reading.

Parallelization

With CPUs no longer getting significantly faster due to physical limits (end of Moore's Law), modern computers now rely on multiple cores to handle tasks in parallel. Polars is designed to make the best use of this parallelism, as it splits tasks across multiple cores to speed up data processing. Some tasks are naturally easy to run in parallel because they do not need communication or sharing of data between threads. Polars takes advantage of these tasks, such as aggregating columns in a DataFrame or applying functions in group-by operations. Each thread can process its part independently, which makes these operations highly efficient.

Many data operations such as grouping or joining rely on creating hash tables. But splitting these tasks across threads is tricky, if each thread builds its own hash table, the results will need extra work to combine, which can cause delays (synchronization phase).

Sharing one hash table with all threads causes a bottleneck because threads wait for access (locking), which makes parallelism ineffective.

Polars avoid these problems with a lock-free hashing algorithm. Instead of threads waiting on each other:

- Each thread calculates hashes for its data and decides which thread should handle each key. This is based on a simple rule: hash value % thread count.
- This guarantees each thread has unique keys in its hash table.
- Once all threads finish, the results are combined quickly, which makes hashing both parallel and efficient.

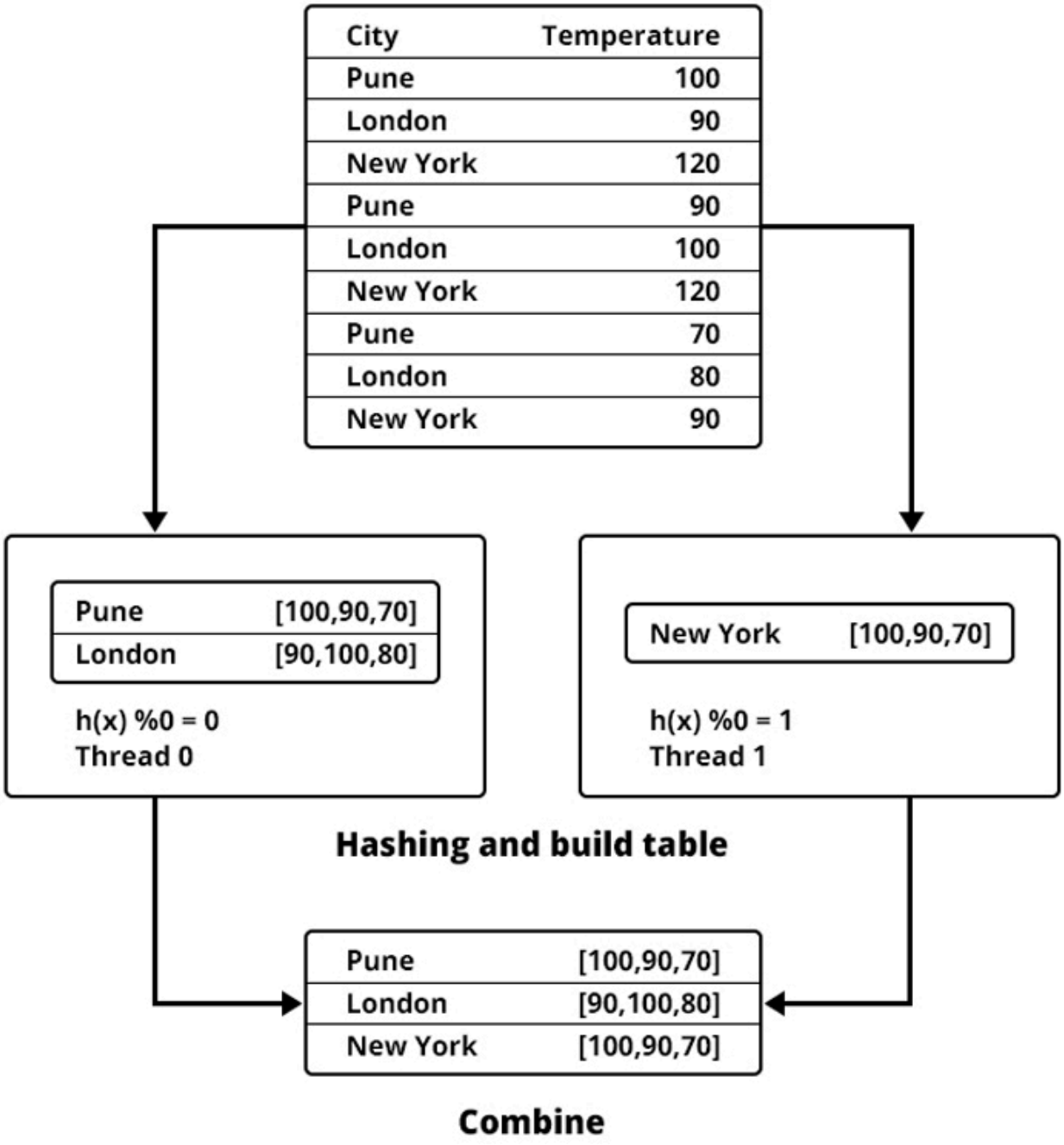

Figure 1.4: *Lock Free Hashing*

The expression hash_value % thread_number is commonly used in hash-based partitioning to distribute data or tasks across multiple threads in a balanced manner. Here is a detailed explanation:

- **Hash Value:** A hash function takes an input (such as a data element or key) and produces a numeric value (the "hash value") that uniquely

represents the input. The hash value is deterministic as the same input yields the same hash, but it appears as random for varied inputs.

- **Modulo Operation (%):** The modulo operation computes the remainder when the hash value is divided by the total number of threads (thread_number). This ensures that the result is always in the range [0, thread_number - 1].
- **Thread Assignment:** The result of hash_value % thread_number determines the thread responsible for processing that particular data element. Each result corresponds to a thread ID (for example, 0, 1, 2, …, thread_number - 1).

Key Benefits:

- **Even Distribution**: After assuming that the hash function distributes values uniformly, the modulo operation ensures that data or tasks are evenly spread across all threads.
- **Deterministic Mapping***:* The same input will always map to the same thread, which will bring about consistency in task allocation.

This innovative approach ensures maximum use of parallelism, while avoiding the delays of synchronization and locking.

Query Optimization

Query optimization in Polars is driven by its lazy evaluation model, which allows the framework to analyze the entire query plan before execution and apply optimizations for efficiency. When using lazy frames (LazyFrame), Polars constructs a logical query plan that describes the data transformations. Before executing the plan, Polars applies optimizations such as predicate pushdown (filtering data early in the pipeline), projection pushdown (selecting only necessary columns), and join simplifications (optimizing the way datasets are merged). Additionally, Polars benefits from its columnar memory model and uses efficient Rust-based execution to process data in parallel, while leveraging Single Instruction, Multiple Data (SIMD) operations for faster computation. These optimizations enable Polars to handle large datasets with high performance while minimizing resource usage.

Comparison with Other Dataframe Libraries

Polars is a high-performance DataFrame library designed to handle large datasets efficiently, while leveraging modern hardware features such as multi-threading and Single Instruction Multiple Data (SIMD). While Pandas, Dask, and PySpark are widely used, Polars stands out with its focus on performance, simplicity, and a unique query engine inspired by SQL. The following is a comparison of Polars with these libraries (Refer *Table 1.1*):

Polars versus Pandas

Performance

Polars is significantly faster than Pandas for large datasets because of their columnar memory layout, lazy evaluation, and multi-threaded execution. Pandas processes data row by row, which is less efficient for analytical workloads.

Scalability

Pandas struggles with datasets larger than memory. Polars, on the other hand, uses memory-efficient techniques and can handle out-of-core computations with tools such as Arrow or by chunking data.

API and Features

Both libraries have intuitive APIs, but Polars introduces additional features such as lazy frames for query optimization and parallel execution.

Polars versus Dask

Parallelism

Dask provides parallelism by splitting data into smaller Pandas-like DataFrames, while Polars natively supports multi-threading and SIMD for faster computation. Polars' built-in parallelism is easier to use compared to Dask's chunk-based setup.

Ease of Use

Polars is simpler to work with as it does not require manual tuning of partitions or task graphs such as Dask.

Scalability

Dask is designed for distributed computing and scales well across clusters. Polars is more suited for single-machine environments but is optimized to handle large datasets efficiently within that constraint.

Polars versus PySpark

Performance

For single-machine workloads, Polars outperforms PySpark due to its optimized execution model and lack of overhead from distributed systems. However, PySpark excels in distributed environments and scales effortlessly across multiple nodes.

Ease of Use

PySpark has a steeper learning curve, requiring knowledge of Spark internals. Polars, with its Pythonic API, is more beginner-friendly and integrates seamlessly into Python workflows.

Deployment

PySpark is ideal for big data and distributed systems. Polars shines in scenarios where performance on a single machine is critical without the complexity of a distributed setup.

Metric	Polars	Pandas	Dask	PySpark
Execution Model	Multi-threaded, SIMD	Single-threaded	Parallel (Chunked Pandas)	Distributed
Performance	Fastest for single machine	Slower for large data	Good for medium workloads	Best for distributed systems
Scalability	Single machine, large data	Limited by memory	Scales across a cluster	Designed for distributed
Optimizations	Lazy evaluation, Arrow-based	None	Task graph optimizations	Optimized for distributed ops
Best Use Case	Single-machine, large data	Small to medium data	Medium to large-distributed	Large-scale distributed

Table 1.1: *Polars versus Incumbent Dataframe Libraries*

Polars versus Pandas

To compare the execution times of Polars and Pandas. We will be creating a dataframe with 10 million rows using Pandas and Polars, with two columns: id (containing random integers between 1 and 100) and 9 value columns (containing random float values).

- Perform a groupby operation on the id column and calculate the following aggregations for each of the 9 value columns:
    - Sum
    - Mean
    - Max
    - Min
- Measuring the execution time of these operations using timeit to compare the performance between Pandas and Polars.

Pandas: A straightforward, single-threaded approach for performing the groupby aggregation, which is suitable for small-to-medium datasets, but is potentially slower with larger datasets like 10 million rows.

Polars: It uses LazyFrames for deferred execution, which optimizes the computation by deferring the operation and optimizing the execution plan. This makes Polars much faster than Pandas for large datasets, as it can leverage multi-threading and efficiently manage memory.

Now let us perform a simple comparison of Pandas and Polars. We will generate 10 million rows of data, with random integers between 1 and 100 for the ids column and random float values for 9 values columns using numpy. A seed is set for reproducibility of the random values, as shown here:

```
import numpy as np
# Set random seed for reproducibility
np.random.seed(42)
# Generate 10 million rows
ids = np.random.randint(1, 101, size=10_000_000)  # Random
integers between 1 and 100
values = np.random.rand(10_000_000, 9)  # Random float values,
9 value columns
```

Now, we will create a Pandas DataFrame with 9 value columns and an id column, and then perform a groupby operation on the id column. We will also calculate the sum, mean, max, and min for each value column.

```
import pandas as pd
import timeit
# Create the Pandas DataFrame
df_pandas = pd.DataFrame(values, columns=[f'value_{i}' for i in
range(1, 10)])
df_pandas['id'] = ids
# Measure the execution time for groupby operation
pandas_time = timeit.timeit(lambda: df_pandas.groupby('id')
[['value_1', 'value_2', 'value_3', 'value_4', 'value_5',
'value_6', 'value_7', 'value_8', 'value_9']].agg(['sum',
'mean', 'max', 'min']), number=5)
print(f"Pandas execution time: {pandas_time} seconds")
```

Now, we will perform the same operation using Polars Lazyframe, as shown here:

```
import polars as pl
import timeit
# Create the Polars LazyFrame
df_polars_lazy = pl.LazyFrame({f'value_{i}': values[:, i-1] for
i in range(1, 10)}).with_columns(pl.lit(ids).alias('id'))
# Measure the execution time for groupby operation
polars_lazy_time = timeit.timeit(lambda:
df_polars_lazy.group_by('id')
.agg([pl.col(f'value_{i}').sum().alias(f'sum_value_{i}') for i
in range(1, 10)] +
[pl.col(f'value_{i}').mean().alias(f'mean_value_{i}') for i in
range(1, 10)] +
[pl.col(f'value_{i}').max().alias(f'max_value_{i}') for i in
range(1, 10)] +
[pl.col(f'value_{i}').min().alias(f'min_value_{i}') for i in
range(1, 10)]).collect(), number=5)
print(f"Polars LazyFrame execution time: {polars_lazy_time}
seconds")
```

The execution times for the groupby operation with aggregation on a dataset of 10 million rows show a clear performance advantage which Polars has over Pandas. Pandas took 7.76 seconds, while Polars, using a LazyFrame, completed the same task in 4.06 seconds. This significant difference in execution time demonstrates the efficiency of Polars, especially with large

datasets. Polars' ability to optimize computations using lazy execution and leverage multi-threading allows it to outperform Pandas, which is single-threaded and may struggle with large data sizes. The results underline the benefits of using Polars for data processing tasks that involve larger volumes of data.

The preceding example presents a simple case of benchmarking for groupby operations between Pandas and Polars. However, in real-world scenarios, benchmarking data processing frameworks involves considering various factors to get accurate, reliable, and meaningful results. Some of the key considerations include:

- **Data Size:** Benchmarking should be done with different dataset sizes (small, medium, large) to understand how frameworks scale.
- **Memory Usage:** Monitor memory consumption alongside execution time, as some libraries may perform faster, but use more memory.
- **Data Complexity:** Complex data types (for example, nested structures, string manipulations) can impact performance differently across libraries.
- **Concurrency/Parallelism:** For multi-threaded or distributed frameworks, measure performance with varying numbers of threads or worker nodes.
- **Hardware/Environment**: Benchmark results can vary based on the machine's CPU, memory, and storage configuration, as well as the software environment.
- **Data Preprocessing**: Preprocessing steps (such as cleaning, transformation, or feature engineering) should be considered, as they can affect the overall performance.
- **I/O Operations:** Consider disk read/write operations, if working with large datasets stored on disk (rather than in memory).
- **Latencies:** Measure latency for individual operations to identify bottlenecks that can affect the overall performance.
- **Algorithmic Optimizations**: Some frameworks may have built-in optimizations, while others may require manual fine-tuning (for example, indexing, caching).

- **Real-World Workflows**: Benchmark real-world tasks (for example, joins and complex aggregations) rather than isolated operations to capture performance under practical workloads.

Considering all these factors ensures that the benchmark results accurately reflect the performance of data processing frameworks in a production environment. We will cover these in depth in the upcoming chapters.

Installation of Polars in Python Environment

Now, we will cover how to install Python and Polars on your machine. If you would like to follow along with the book then you can install all the packages from the requirements file provided in the repository.

1. **Install Python**

   Step 1: Download and install Python from the official Python website.

   Visit https://www.python.org/downloads/.

   Choose the appropriate installer based on your operating system (Windows, macOS, Linux).

   Windows: Make sure to check the box "Add Python to PATH" during installation for easier access via the command line.

   macOS/Linux: Python should already be installed on your system by default, but if not, you can install it using Homebrew (for macOS) or the package manager (for Linux, sudo apt; install python3 for Ubuntu).

   Step 2: Verify the Python installation as follows:

   ```
   python --version
   Or for Python 3:
   python3 -version
   ```

2. **Install Polars**

   Once Python is installed, follow these steps to install Polars:

   Step 1: Open your terminal/command prompt.

   Step 2: Install Polars using pip (Python's package installer):

   ```
   pip install polars
   ```

   Or for Python 3 specifically, use:

   ```
   pip3 install polars
   ```

Step 3: Verify that Polars is installed correctly by running the following:

```
import polars as pl
print(pl.__version__)
```

This will print the version of Polars, while confirming the installation.

Note: *You can also install Polars with all optional dependencies via the following command:*

```
pip install 'polars[all]'
```

If you are expecting to encounter a row count > 4.2 billion, then you can consider using the following command for installation:

```
pip install polars-u64-idx
```

Our First Dataframe with Polars

The code reads a CSV file directly into a DataFrame using the pl.read_csv() function. This loads the data immediately and allows you to work with it without the need for lazy evaluation. The first 5 rows of the DataFrame are printed using df.head(), which provides a quick preview of the data. This approach is simple and efficient for cases where you want to perform immediate operations on the dataset.

```
import polars as pl

# Step 1: Define the path to the CSV file
csv_file_path = r"C:\Polars-
book\Chapter1\\"+'large_dataframe.csv'  # Replace with your
actual file path

# Step 2: Read the CSV file using Polars
# Polars reads the CSV file and loads it into a DataFrame by
default
df = pl.read_csv(csv_file_path)

# Step 3: Print the first 5 rows (head) of the DataFrame
print(df.head())
```

Output:

shape: (5, 10)

col_1	col_2	col_3	col_4	…	col_7	col_8	col_9	col_10
f64	f64	f64	f64		f64	f64	f64	f64
-1.182901	-2.467077	0.869938	0.022906	…	0.304478	0.012752	0.804861	1.718607
-0.170019	0.365224	-0.086186	0.81293	…	-1.029503	0.54984	0.62414	1.206564
-0.142156	0.006468	-0.70849	-0.458606	…	2.396807	1.330939	-1.342845	0.874591
0.11241	-0.058192	0.675886	0.739299	…	-0.852386	-0.242498	0.634894	0.693562
-0.274246	-0.679662	0.596456	0.46892	…	-1.822351	0.969387	-0.800207	-0.787063

Figure 1.5: *Preview of Dataframe Using df.head()*

Conclusion

Polars is a high-performance dataframe library designed for modern data processing needs. It outperforms traditional libraries such as Pandas for large-scale data operations due to its parallel execution, efficient memory usage, and innovative features such as lazy evaluation. With its simple API, high-performance backend, and ability to handle large datasets efficiently, Polars is well-positioned for data-intensive tasks in both academic and industry settings. In the next chapter, we will explore the foundational data types in Polars, including Series and DataFrames for efficient data handling. We will cover optimized numeric and temporal types for high-performance calculations and precise date/time management. Additionally, we will discuss nested and string data types for handling complex and textual data efficiently.

Key Takeaways

- **Polars**: It is an open-source dataframe library designed for high performance, especially with large datasets. It started in 2020 and was developed using Rust, with Python being one of its primary interfaces. Polars focuses on fast, parallelized execution with memory-efficient data processing, which makes it well-suited for both single-node and large-scale data tasks.
- **Performance and Scalability**: Pandas is a widely used dataframe library but is single-threaded and less optimized for large datasets. Polars improves upon this by utilizing parallelism, SIMD (Single

Instruction Multiple Data), and Apache Arrow backend for better memory utilization and faster execution.

While Pandas struggle with memory-intensive operations, Polars efficiently handles large datasets, often outperforming Pandas in operations such as groupby, aggregation, and complex transformations.

- **Apache Arrow Integration**: Polars leverage the Apache Arrow columnar memory format, thereby optimizing both storage and computation. Arrow's memory layout is designed for high-performance analytical queries, with zero-copy data exchange and efficient storage. This enables faster access to specific columns and supports vectorized execution for more efficient data manipulation, especially in large-scale datasets.
- **Lazy Evaluation:** Polars introduces LazyFrames, allowing deferred execution. This enables query optimization, parallel execution, and efficient computation, which helps in minimizing unnecessary operations.
- **Parallelization**: Polars fully exploit multi-core CPUs, distributing tasks across threads to process data in parallel. It avoids common issues such as synchronization and locking, which results in maximum performance.
- **Comparison with Other Libraries**:
    - **Pandas**: Best for small to medium datasets, but slower and memory-intensive for larger datasets. It does not utilize parallelism or advanced optimizations.
    - **Dask:** Focuses on parallelism and scalability but requires manual tuning of partitions and task graphs. It is ideal for larger distributed workloads, but can be more complex than Polars.
    - **PySpark**: Suited for distributed computing and large-scale operations across clusters. However, it has more overhead for single-node tasks compared to Polars, which excels in local data processing with minimal overhead.

Polars can be installed easily with pip, which makes it accessible for Python users. You can install the library with pip install Polars, and verify the installation by importing the library and checking the version. Additionally,

you can install Polars with all the optional dependencies for enhanced functionality and better handling of large row counts.

Exercises

1. Benchmarking Polars versus Dask: In this chapter, we have performed a simple benchmarking comparison between Polars and Pandas. Now, perform a similar comparison between Polars and Dask. Measure the execution time for similar operations (for example, creating a DataFrame, groupby aggregation, and so on.), and note down your observations on:
    - Execution Time
    - Memory Usage (If Possible)
    - Ease of Use
2. Working with JSON Data:
    - Create a DataFrame from the JSON file provided in this chapter's repository.
    - After loading the data into a Polars DataFrame, try to export it to CSV format and confirm the process by checking the file.
    - Make sure to check for any challenges or nuances you encounter, while converting from JSON to CSV.