Polars: Working with Big Data

On just your old laptop

Pranjal Rawat

April 19, 2025

Georgetown University

Installation, Setup & Why Polars?

1. Install & Setup

Install Polars

```
pip install polars pyarrow # pyarrow recommended
```

Import & Define Paths

```
import polars as pl
from pathlib import Path

# Folder with your CSVs/subfolders
DATA_ROOT = Path("./your_big_data_folder")
# Finds all CSVs inside DATA_ROOT, recursively
CSV_PATTERN = str(DATA_ROOT / "**/*.csv")

print(f"Targeting: {CSV_PATTERN}")
```

Why Polars? (The Gist)

Compared to libraries like Pandas:

- *Lazy Execution:* Plans first, computes later (avoids loading all data).
- *Multi-threaded:* Uses all CPU cores automatically (faster).
- *Fast Rust Core:* Memory-efficient foundation.
- *Smart Optimizer:* Reduces unnecessary work.

Key takeaway: Handles bigger data, often much faster, using **less RAM**.

Basic Lazy Syntax

2. Scan, Don't Read!

Fundamental Step: Use `pl.scan_csv()` to create a `LazyFrame`.

- Reads *metadata* (headers, paths), *not* the actual data rows.
- Creates a lightweight *plan*. Uses almost no memory.

Creating the LazyFrame

```
lf = pl.scan_csv(
    CSV_PATTERN,
    has_header=True,
    separator=",",
    # --- Specify dtypes here! (See Optimizations) ---
    # dtypes=dtype_hints,
    low_memory=True # Can help parser
)
print(f"Schema Polars will use:\n{lf.schema}")
```

Building the Plan: Filtering & Selecting

Chain operations onto the `LazyFrame` (`lf`). This just adds steps to the plan.

Filtering Rows

```
# Plan to keep rows where value > 100.0
lf_filtered = lf.filter(pl.col("value") > 100.0)
```

Selecting Columns

```
# Plan to select specific columns after filtering
lf_selected = lf_filtered.select(
    ["id", "timestamp", "category", "value"]
)
```

Still no major computation or memory usage!

Building the Plan: Transforming Columns

Add or modify columns using `with_columns`. Still lazy!

Adding/Modifying Columns

```
# Plan to add a calculated column \& modify category
lf_with_cols = lf_selected.with_columns([
    (pl.col("value") * 1.1).alias("value_plus_10_percent"),
    pl.col("category").str.replace("Type_", "T-").alias("
     category_short")
])
```

Conditional Logic (like CASE WHEN)

```
# Plan to create a 'value_tier' column
lf_with_tier = lf_with_cols.with_columns(
    pl.when(pl.col("value") > 1000)
      .then(pl.lit("High"))
      .when(pl.col("value") > 100)
      .then(pl.lit("Medium"))
      .otherwise(pl.lit("Low"))
      .alias("value_tier")
)
```

Building the Plan: Aggregating & Sorting

Group data and calculate summary stats, then sort.

Grouping and Aggregating

```
# Plan to calculate sum/mean value per category
lf_agg = lf.group_by("category").agg([
    pl.sum("value").alias("total_value"),
    pl.mean("value").alias("average_value"),
    pl.count().alias("num_records")
])
```

Sorting Results

```
# Plan to sort the aggregated results
lf_sorted_agg = lf_agg.sort("total_value", descending=True)
```

Getting Results (Execution)

3. Getting Results: The Execution Choice

You've built the plan (`LazyFrame`). Now, execute it.

Your choice depends on the expected size of the FINAL result.

Main Execution Strategies

- `.collect()`: Result fits easily in RAM.
- `.collect(streaming=True)`: Result *might* fit RAM; reduces peak memory *during* compute.
- `.sink_*()`: Result too big for RAM; writes directly to disk.

Execution: `.collect()`

- Use Case: Small final results (summaries, small samples).
- *Action:* Executes plan → returns standard DataFrame in RAM.

Example: Small Summary

```
# Assumes lf_sorted_agg produces a small table
summary_df = lf_sorted_agg.collect()
print(summary_df)
```

Execution: `.collect(streaming=True)`

- Use Case: Larger results where computation might spike memory, but the final DataFrame *should* fit in RAM.
- *Action:* Executes plan in stages, reducing peak memory *during computation*.
- *Caveat:* Needs enough RAM for the final complete DataFrame.

Example: Larger Filtered Data

```
# lf_with_tier might produce many rows, but maybe it fits
try:
    transformed_data = lf_with_tier.collect(streaming=True)
    print(f"Collected {len(transformed_data)} rows.")
except Exception as e:
    print(f"Streaming collect failed (result too big?): {e}")
```

Execution: `.sink_parquet()` / `.sink_csv()`

- Use Case: HUGE results that definitely won't fit in RAM.
- *Action:* Executes plan → writes output *directly* to file.
- Most memory-safe. Parquet is faster/smaller than CSV output.

Example: Saving Huge Filtered Output

```
# Save large filtered data directly to disk
(lf_filtered # Plan from earlier filter slide
 .sink_parquet(
     "filtered_output.parquet",
     compression="zstd" # Good compression
 )
)
print("Filtered data saved to disk.")
```

Key Optimizations

4. Key Optimizations

Make Polars work even better:

1. Specify Data Types
2. Filter & Select Early
3. Convert to Parquet (Why?)

Optimization 1: Specify Data Types

Benefit: Less memory usage & faster operations.

- Use `dtypes` in `scan_csv`.
- `pl.Categorical`: For repeating strings (categories, codes). Massive memory saver. Stores each unique string only once.
- Smaller Numerics: `pl.Float32`, `pl.Int32`, `pl.Int16`, etc. Use the smallest type that holds your data range without losing needed precision.

Example with `dtypes`

```
dtype_hints = {
    "user_id": pl.UInt32,
    "product_category": pl.Categorical,
    "rating": pl.Float32, # Less precision than Float64
    "order_date": pl.Date
}
lf = pl.scan_csv(CSV_PATTERN, dtypes=dtype_hints)
```

Optimization 2: Filter & Select Early

Benefit: Polars' optimizer avoids reading unnecessary data *from disk*.

- Place `.filter(...)` and `.select(...)` early in your chain.
- Less data read from disk = Less I/O = Faster execution.
- Less data processed later = Less memory and CPU needed.

Good Practice

```
lf.filter(...).select(...).with_columns(...).group_by(...)
```

Why Consider Parquet Format?

Parquet is designed for efficient analytical querying:

- Columnar Storage: Reads only the columns specified in `.select()`. CSV requires reading entire rows even for one column. Huge I/O saving.
- Efficient Compression: Files are typically much smaller than equivalent CSVs, saving disk space and speeding up reads.
- Schema & Statistics: Stores data types and statistics (min/max) within the file. Polars uses statistics to skip reading irrelevant data chunks (*predicate pushdown*) based on `.filter()` conditions.

Optimization 3: Convert to Parquet

If you analyze this dataset often:

- Perform a **one-time conversion** from CSV to Parquet using the lazy `scan_csv -> sink_parquet` method.
- Use `pl.scan_parquet()` for all future analyses – it leverages Parquet's benefits and is significantly faster than re-scanning CSVs.
- Consider partitioning the Parquet dataset during conversion (e.g., by year, month, category) if you frequently filter on those columns.

Summary

Polars Big Data Mantra

Scan Lazy

Operate Lazy

Optimize Types

Collect/Sink Smartly

Appendix: More Syntax Examples

More Syntax Examples

A quick reference for other common Polars operations. These work on `LazyFrames` too.

Reading/Writing Other Formats

Scanning/Reading (Lazy Preferred)

```
lf_parquet = pl.scan_parquet("path.parquet")
# lf_ipc = pl.scan_ipc("path.arrow") # Feather format

# Eager read (loads all into RAM)
# df_small = pl.read_parquet("small.parquet")
# df_json = pl.read_json("config.json")
```

Writing/Sinking (Lazy Preferred for Large)

```
# From LazyFrame
# lf.sink_parquet("output.parquet", ...)
# lf.sink_ipc("output.arrow", ...)

# From Eager DataFrame (df must fit RAM)
# df.write_parquet("small_output.parquet")
# df.write_json("small_output.json")
```

Inspecting DataFrames

Schema and Shape

```
# Get schema (column names and types)
print(lf.schema) # Infers schema from source

# Collect schema without full data compute
lazy_schema = lf.collect_schema()

# Get shape (rows, columns) - requires compute!
# Use fetch() for estimation or collect()
# estimated_rows = lf.fetch(1).height
# full_shape = lf.collect().shape # Computes fully!
```

Viewing Head/Tail (Use `fetch/limit`)

```
# View first N rows (triggers compute for N rows)
print(lf.fetch(5)) # Recommended for LazyFrames
# Or: print(lf.limit(5).collect())

# Tail is hard for LazyFrames (needs full scan)
# print(df.tail(5)) # Eager only easily
```

More Column Selections

Using Expressions & Aliases

```
lf_renamed = lf.select([
    pl.col("old_name").alias("new_name"),
    (pl.col("value") * 100).alias("value_pct"),
    pl.col("timestamp") # Keep original
])
```

Using Regex & Selectors

```
# Select columns starting with 'sensor_'
lf_sensor_cols = lf.select(pl.col("^sensor_.*$"))

# Selectors (requires import polars.selectors as cs)
# Need eager frame or collect() for full function
# print(df.select(cs.string() | cs.numeric()))
```

More Row Filtering

OR Conditions

```
lf_or = lf.filter(
    (pl.col("status") == "CANCELLED") | (pl.col("value") == 0)
)
```

Null Checks & Membership

```
lf_not_null = lf.filter(pl.col("optional_field").is_not_null())

codes = ["X", "Y", "Z"]
lf_in_list = lf.filter(pl.col("code").is_in(codes))
lf_not_in = lf.filter(~pl.col("code").is_in(codes)) # Negate
```

String & Datetime Ops

String Manipulations (.str)

```
lf_strings = lf.with_columns([
    pl.col("name").str.to_uppercase().alias("upper_name"),
    pl.col("notes").str.contains("urgent", literal=True).alias("
     is_urgent"),
    pl.col("product_id").str.slice(0, 4).alias("product_group")
])
```

Date/Time Manipulations (.dt)

```
lf_datetime = lf.with_columns([
    pl.col("timestamp").dt.date().alias("date_only"),
    pl.col("timestamp").dt.time().alias("time_only"),
    pl.col("timestamp").dt.strftime("%Y-%m").alias("year_month")
])
```

Advanced Aggregations

More Aggregation Functions

Combine these within .agg([...])

```
# Examples used inside .agg()
pl.median("value").alias("median_val")
pl.std("value").alias("std_dev_val")
pl.n_unique("user_id").alias("distinct_users")
pl.first("timestamp").alias("first_event")
pl.last("value").alias("last_value")
pl.quantile("value", 0.95).alias("p95_value")
# Conditional aggregation
pl.sum("value").filter(pl.col("type") == "A").alias("sum_A")
pl.col("value").filter(pl.col("type") == "B").mean().alias("mean_B")
```

Window Functions (`.over()`)

Apply calculations over groups without collapsing rows. Often needs sorting first.

Window Function Examples

```
# Assume lf_sorted = lf.sort(["group_id", "timestamp"])

lf_window = lf_sorted.with_columns([
    # Sum of 'value' for each 'group_id'
    pl.sum("value").over("group_id").alias("group_total"),

    # Rank within each 'group_id' based on 'value'
    pl.col("value").rank(method='dense').over("group_id").alias("
     rank_in_group"),

    # Get previous value within 'group_id'
    pl.col("value").shift(1).over("group_id").alias("previous_value"
    ),

    # Calculate difference from previous value within 'group_id'
    pl.col("value").diff().over("group_id").alias("change_from_prev"
    )
])
```

Joining DataFrames

Join `LazyFrame` + `LazyFrame` for best memory use. If one is eager (`df`), use `df.lazy()`.

Common Join Types

```
# Assume lf_left and lf_right are LazyFrames
lf_inner = lf_left.join(lf_right, on="key", how="inner")
lf_left_join = lf_left.join(lf_right, on="key", how="left")

# Join on different key names
lf_diff_keys = lf_left.join(lf_right,
                            left_on="left_key",
                            right_on="right_key",
                            how="inner")

# Other `how` options: "outer", "anti", "semi", "cross"
```

Concatenating (Stacking)

Stacking Rows (`pl.concat`)
DataFrames must have compatible schemas (same column names/types or use `how='diagonal'`).

```
# Assume lf1, lf2 are LazyFrames with same schema
lf_stacked = pl.concat([lf1, lf2], how="vertical")

# Stacking many LazyFrames from a list
# list_of_lfs = [pl.scan_csv(f) for f in list_of_files]
# lf_all_stacked = pl.concat(list_of_lfs, how="vertical")
```

Handling Missing Data

Dropping Nulls

```
# Drop rows with any null value
lf_no_nulls = lf.drop_nulls()

# Drop rows with nulls in specific columns
lf_subset_no_nulls = lf.drop_nulls(subset=["colA", "colB"])
```

Filling Nulls

```
# Fill with a literal value
lf_fill_lit = lf.with_columns(
    pl.col("value").fill_null(0).alias("value_no_null"),
    pl.col("category").fill_null("MISSING").alias("cat_no_null")
)

# Fill with mean/median (needs pre-calculation or window fn)
# mean_val = lf.select(pl.mean("value")).collect().item() # Computes
    !
# lf_fill_mean = lf.fill_null(mean_val) # Use eager value

# Fill using forward/backward strategy (often for timeseries)
# lf_sorted = lf.sort("time") # Need sorted data
# lf_ffill = lf_sorted.with_columns(pl.col("value").forward_fill())
```

Sampling & Other Ops

Sampling (Requires Computation)

Sampling a `LazyFrame` directly usually triggers `collect`. Sample *after* filtering/limiting if possible.

```
# Collect first, then sample (if result fits RAM)
# eager_df = lf.collect()
# sample_n = eager_df.sample(n=100)
# sample_frac = eager_df.sample(frac=0.01)

# Efficient sampling of a filtered subset
sample_of_filtered = (
    lf
    .filter(pl.col("value") > 500)
    .fetch(10000) # Get first 10k matching rows (less memory)
    .sample(n=100) # Sample from the fetched eager frame
)
```

Dropping Columns

```
lf_less_cols = lf.drop(["col_to_remove1", "notes"])
```

Appendix: Under the Hood

Under the Hood: Why Polars is Fast

Polars achieves its speed through several core design choices:

- Columnar Memory (Apache Arrow): Data for each column is stored together in memory. This is very CPU cache-friendly and allows for fast, vectorized operations (SIMD) on entire columns at once. Enables zero-copy data sharing with other Arrow-compatible tools.
- Native Rust Engine & Multi-threading: Core operations run in fast, compiled Rust code, bypassing Python's GIL. Polars automatically uses all available CPU cores for parallel execution of many tasks (filters, aggregations, etc.).

Under the Hood: Why Polars is Fast

- Query Optimizer: In lazy mode, Polars analyzes your entire sequence of operations and optimizes it *before* execution (like a database). It performs optimizations like predicate pushdown (filtering at the source) and projection pushdown (reading only needed columns).
- Streaming Capability: For operations that allow it (like scanning files or some aggregations), Polars can process data in chunks without loading everything into RAM, enabling work on datasets larger than available memory.

Polars vs. Pandas: Key Differences

While both offer DataFrames, their approach differs significantly:

- *Index:* Polars is index-free. Rows are identified by position. Pandas relies heavily on row indexes, affecting operations and alignment.
- *Memory:* Polars uses Apache Arrow (columnar). Pandas primarily uses NumPy arrays (can be less efficient for mixed types or strings).
- *Parallelism:* Polars is multi-threaded by default. Pandas is mostly single-threaded.

Polars vs. Pandas: Key Differences

While both offer DataFrames, their approach differs significantly:

- *Evaluation:* Polars defaults to/excels with lazy evaluation (optimizes pipelines). Pandas is primarily *eager* (computes each step immediately).
- *Typing:* Polars is strict about data types (predictable). Pandas can be more flexible but sometimes changes types unexpectedly (e.g., int to float with nulls).
- *API Focus:* Polars uses a powerful Expression API for transformations within its engine. Pandas often requires vectorization or sometimes Python-level loops/apply.

Polars vs. Spark: Different Scales

Use Polars for large data on your laptop/server. Use Spark when you truly need a multi-machine cluster for massive scale.

- *Scope:* Polars = Single Node. Optimized for data that fits (or can be streamed) on one powerful machine. Spark = Distributed Cluster. Designed for datasets far exceeding single-node capacity.
- *Engine:* Polars runs natively (Rust). Spark runs on the JVM (Java/Scala), involving more overhead (startup, GC, network shuffle). On data that fits one node, Polars is typically much faster and lighter.

Polars vs. Spark: Different Scales

- *API Feel:* Polars feels more column-centric (operations on column expressions). Spark (PySpark) feels slightly more row/SQL-centric, constrained by distributed concepts.
- *Laziness:* Both are lazy and have query optimizers. Polars optimizes for single-node parallelism and memory; Spark optimizes for distributed execution (data partitioning, shuffle).

Thinking in Polars: Idiomatic Usage

Get the most out of Polars by adopting its style:

- Embrace Laziness: Use `scan_*` and chain operations before `.collect()` or `.sink_*()`. Let the optimizer work!
- Use the Expression API: Prefer `pl.col()`, `pl.when()`, `.over()`, etc. Avoid Python loops or Pandas-style `.apply()`. Describe *what* you want, let Polars handle *how*.
- Think Columnar: Formulate operations on entire columns. Filter with boolean expressions, transform with column algebra.
- Leverage Built-ins: Use `.group_by().agg()` and window functions (`.over()`) directly for group-wise logic instead of manual merges.
- Be Type-Aware: Specify efficient `dtypes` (`Categorical`, smaller numerics). Use Polars' strict typing for predictable results.