ChatGPT

Polars (Python) Reference for DataFrames (>= 0.20)

Frame Modes: Eager vs. Lazy vs. Streaming

Polars DataFrame (eager) executes operations immediately and stores results in memory, similar to pandas. In contrast, a **LazyFrame** represents a deferred computation: it **builds a query plan without executing it until explicitly collected** [1]. This lazy evaluation allows Polars to optimize queries (e.g. filter and projection pushdown) and run computations in parallel [2] [3]. Use `df.lazy()` to get a LazyFrame from an existing DataFrame [4].

Polars also supports a **streaming execution engine** for LazyFrames. In streaming mode, Polars processes data in **batches** so that datasets larger than RAM can be handled by *streaming through data instead of loading everything in memory* [5]. To execute a lazy query in streaming mode, call `.collect(engine="streaming")` on the LazyFrame [6] (in Polars >=0.20, `engine="streaming"` replaces older `streaming=True`). When streaming, Polars will execute the plan pipeline-wise (indicated by "PIPELINE" in `LazyFrame.explain(streaming=True)` output) and chunk data into batches (e.g. *STREAMING CHUNK SIZE* of 12,500 rows by default, computed from number of threads and columns) [7] [8]. Streaming is **transparent** – if an operation isn't implemented in streaming, Polars automatically falls back to the in-memory engine for that part of the plan [3].

Summary: Use eager `pl.DataFrame` for quick interactions or when results fit in memory. Use `pl.LazyFrame` for complex pipelines and let Polars optimize and parallelize the work. Use **streaming** lazy execution for very large data or out-of-core processing, with the caveat that not all operations stream (e.g. certain joins or sorts may buffer data). You can inspect a lazy plan via `LazyFrame.explain()` or visualize it via `LazyFrame.show_graph(...)` to see how Polars will execute it [3]. Lazy execution enables advanced optimizations described below.

Expression API and Deferred Execution Semantics

Polars provides a **column expression API** for transforming data. Instead of applying Python functions row-by-row, you build vectorized expressions using Polars internals (which are implemented in Rust and parallelized). For example, `pl.col("x") + 5` creates an expression that will add 5 to column **x**. These expressions are *deferred*: they don't run until we tell a DataFrame or LazyFrame to execute them (in a `select`, `with_columns`, `filter`, etc.).

Expressions can be used in: - `DataFrame.select(exprs…)` – to create a new DataFrame with columns defined by the given expressions [9]. - `DataFrame.with_columns(exprs…)` – to add or replace columns in an existing DataFrame using expressions. - `DataFrame.filter(condition_expr)` – to filter rows by an expression (Boolean mask).

Using these methods, Polars will **apply the whole expression pipeline in optimized Rust code, columnar batch-wise**, avoiding Python loops. All basic vectorized operations (arithmetic, comparisons, string ops,

etc.) are available as `pl.Expr` methods or via namespace (e.g. `pl.col("text").str.contains("foo")`). Multiple expressions can be combined; Polars runs them in parallel when possible [2]. Critically, in a LazyFrame context, these expression plans are not executed until `.collect()` is called – allowing the query optimizer to reorder or simplify them for efficiency (see **Query Optimizations** below).

Deferred Execution: In a LazyFrame, building an expression (like `.filter(pl.col("y") > 0).select([...])`) just appends to the query plan. When `.collect()` is invoked, Polars optimizes the plan then executes it in Rust. This yields performance similar to SQL engines. Even in eager mode, methods like `df.select(exprs)` are optimized internally – for instance, `df.select(pl.col("a")+pl.col("b"))` will execute in Rust without Python overhead.

Query Optimizations: Predicate Pushdown and Projection Pruning

Polars automatically optimizes lazy query plans via its **query optimizer**: - **Predicate Pushdown:** Polars will push filter conditions (`filter` / `where` expressions) down to the earliest point possible, often to the **scan** of input data. For example, given `pl.scan_csv("data.csv").filter(pl.col("country") == "US").select("sales")`, Polars will apply the filter as it reads the CSV so that non-US rows are skipped immediately [10] [11]. This reduces IO and memory usage. - **Projection Pruning:** Polars detects which columns are actually needed to satisfy the expressions, and *only reads those columns* from source files or dataframes [10]. In the above example, only the `"country"` and `"sales"` fields would be read from CSV. Unused columns are pruned out of the plan. - **Optimization of expression tree:** Polars can simplify or reorder operations. For instance, redundant operations may be eliminated (common subexpressions are only computed once), and certain linear operations can be fused. Polars will also determine the optimal execution order for joins and aggregations (e.g. small tables might be buffered on the side of a join). - **Aggregate pushdown & partitioning:** In some group aggregations, Polars might automatically partition data by group key (on multiple threads) to aggregate in parallel. (By default Polars chooses partitioning based on group cardinality, which can be tuned or disabled via env vars like `POLARS_NO_PARTITION` [12].) - **Deferred materialization:** In lazy plans Polars keeps data in its efficient **Arrow-based columnar format** throughout, avoiding materializing Python objects unless absolutely necessary.

These optimizations happen behind the scenes when using LazyFrame. They allow Polars to approach query performance similar to SQL engines or Spark, all within a single process. You can use `LazyFrame.explain()` to see the optimized logical plan and `LazyFrame.show_graph(plan_stage="optimized")` for a visual graph. For debugging performance, `LazyFrame.show_graph(plan_stage="physical", engine="streaming")` can illustrate which parts of a plan run in streaming vs in-memory mode [3]. A `POLARS_VERBOSE` environment variable can also be set to get optimizer logs printed to stderr [13].

Note: In eager mode (direct DataFrame methods), Polars applies some of these optimizations too (like projection pushdown in `read_parquet` by only reading needed columns, or using vectorized Rust kernels for operations). But the full power of the optimizer (and ability to reorder operations) comes with lazy queries.

Selecting and Modifying Data

Polars DataFrames support SQL-like selection, filtering, and column transformations:

- **Select Columns:** `df.select(exprs)` evaluates the given expressions and returns a new DataFrame. You can select existing columns by name, or compute new ones. For example:

  ```
  df.select(
      pl.col("total"),
      (pl.col("price") * pl.col("quantity")).alias("amount")
  )
  ```

  This will produce a DataFrame with the original *total* column and a new *amount* column computed by multiplying price and quantity. In lazy mode, `LazyFrame.select` similarly produces a new LazyFrame with those projections (executed on collect). Using `pl.all()` is a convenient wildcard to select all columns, optionally with some transformation.

- **Add/Replace Columns:** `df.with_columns(exprs)` adds new columns or overwrites existing ones, returning a modified DataFrame. For example:

  ```
  df = df.with_columns([
      pl.col("amount") - pl.col("discount").alias("net_amount"),
      pl.when(pl.col("quantity") >
  100).then(True).otherwise(False).alias("bulk")
  ])
  ```

  This adds a *net_amount* column and a boolean *bulk* flag column based on an expression (if quantity > 100) [14] [15]. In lazy mode, `with_columns` just updates the plan (no computation yet).

- **Filter Rows:** `df.filter(condition_expr)` filters rows by the boolean expression (the Polars equivalent of SQL WHERE). For instance: `df.filter(pl.col("category") == "A")`. Filters can be combined with bitwise operators: `df.filter((pl.col("x") > 0) & (pl.col("y") < 5))`. Under the hood, Polars applies the filter in optimized Rust, often with predicate pushdown if it's a LazyFrame scan.

All these methods accept **Polars expressions** which are evaluated efficiently. This contrasts with pandas where one might use Python loops or apply; in Polars you almost always prefer an expression over a Python UDF for performance [16] [17].

Example – Eager vs Lazy Select:

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

```
# Eager select
out = df.select(pl.col("a") * pl.col("b"))
print(out)
# shape: (3, 1)
# ┌───────┐
# │ a * b │
# │ ---   │
# │ i64   │
# ╞═══════╡
# │ 4     │
# │ 10    │
# │ 18    │
# └───────┘

# Lazy select (no output until collect)
lazy_res = df.lazy().select(pl.col("a") * pl.col("b"))
print(type(lazy_res))  # <class 'polars.internals.lazyframe.LazyFrame'>
df2 = lazy_res.collect()  # triggers execution
```

Above, the `.select` with expressions ensures the multiplication runs in Rust. The lazy example defers it until `.collect()`. Both yield the same result, but the lazy version allows further chaining and optimization before computing.

Aggregations and GroupBy

Polars supports powerful aggregation operations, either on the whole DataFrame or grouped by keys:

- **Simple Aggregations:** DataFrame methods like `df.sum()`, `df.mean()`, `df.min()`, etc. compute column-wise aggregates across the entire DataFrame (returning a single-row DataFrame of results) [18] [19]. For example, `df.sum()` returns the sum of each numeric column [20] [21]. There are also horizontal aggregations like `df.sum_horizontal()` to sum across columns for each row [22].

- **GroupBy Aggregation:** Using `df.group_by("key").agg([...])` (or `groupby` alias), you can perform aggregations per group. For example:

  ```
  df.group_by("department").agg([
      pl.sum("sales").alias("total_sales"),
      pl.avg("sales"),
      pl.count()           # number of rows in group
  ])
  ```

  This produces one row per unique *department*, with aggregated statistics. The list passed to `agg` can contain expressions (like `pl.sum("sales")`) or any of Polars' aggregation expressions (sum,

mean, min, max, count, etc.) [23]. Polars group-by is implemented in Rust with parallel aggregation: different groups can be processed on different threads. It may partition large group operations for efficiency as noted earlier.

You can also aggregate multiple columns at once using wildcards or suffixes, e.g. `df.group_by("id").agg(pl.all().sum().suffix("_sum"))` will sum all columns per id (non-numeric columns will be summed as lists by default). Grouping on multiple keys is supported: `df.group_by(["country","year"]).agg([...])`.

- **Rolling and Dynamic GroupBy:** For time-series data, Polars has specialized grouping:
- `df.group_by_dynamic("ts", every="1d", period="7d", by="id").agg(...)` groups by fixed windows of time (like resampling) [24].
- `df.group_by_rolling("ts", period="30m", offset="15m", by="sensor").agg(...)` for rolling time windows. These create time-windowed groups before aggregation.

These are advanced and useful for downsampling or windowed stats in time series, but for brevity, refer to Polars docs for details on the parameters.

- **Window/Aggregate as Expression:** You can also perform aggregations **without group_by** by using the `.over()` **expression** (see next section on Window Functions). E.g. `pl.sum("sales").over("department")` produces an expression giving each row the sum of sales in that department [25] [26]. This is effectively a *window function* that returns a value per row (contrasted with group_by which collapses groups).

Joining Aggregated Results: Often you may join summary stats back to the original data. With Polars, a convenient way is using `.over()` as mentioned (no explicit join needed). But you can also do `df.join(df.group_by("k").agg([...]), on="k")` to attach group results.

Aggregation Output and Nulls: By default, grouping on an empty DataFrame returns no rows. If a group has no values (e.g. all nulls), an aggregation like sum will produce null (Polars generally propagates nulls in aggregations – use `pl.sum("col").fill_null(0)` if needed to treat null as zero).

Example: Grouping and aggregating:

```
df = pl.DataFrame({
    "team": ["A","A","B","B","B"],
    "points": [10, 15, 7, 8, 10]
})
# Total and average points per team:
stats = df.group_by("team").agg([
    pl.sum("points").alias("total_points"),
    pl.mean("points").alias("avg_points")
])
print(stats)
# shape: (2, 3)
# ┌──────┬──────────────┬────────────┐
```

```
# ┌──────┬──────────────┬────────────┐
# │ team ┆ total_points ┆ avg_points │
# │ ---  ┆ ---          ┆ ---        │
# │ str  ┆ i64          ┆ f64        │
# ╞══════╪══════════════╪════════════╡
# │ A    ┆ 25           ┆ 12.5       │
# │ B    ┆ 25           ┆ 8.3333     │
# └──────┴──────────────┴────────────┘
```

This shows basic grouping. We could also get the same *total_points* for each original row by `df.select(pl.col("team"), pl.sum("points").over("team"))`.

Window Functions (Analytics)

Polars supports **window functions** via the `.over()` expression, similar to SQL window functions. Using `Expr.over(partition_keys)` lets you compute an aggregation or expression *over groups defined by partition keys*, but without collapsing the result – the output has the same number of rows as the original (each row gets the aggregate value of its partition) [27] [28]. This is useful for computing group-based metrics like rankings, running totals, or percentages.

Example: Suppose we want to rank each item within its category by price:

```
df = pl.DataFrame({
    "item": ["X", "Y", "Z", "W"],
    "category": ["C1","C1","C2","C2"],
    "price": [5, 10, 7, 3]
})
out = df.select([
    "item", "category", "price",
    pl.col("price").rank("dense",
descending=True).over("category").alias("price_rank")
])
print(out)
# shape: (4, 4)
# ┌──────┬──────────┬───────┬────────────┐
# │ item ┆ category ┆ price ┆ price_rank │
# │ ---  ┆ ---      ┆ ---   ┆ ---        │
# │ str  ┆ str      ┆ i64   ┆ u32        │
# ╞══════╪══════════╪═══════╪════════════╡
# │ X    ┆ C1       ┆ 5     ┆ 2          │
# │ Y    ┆ C1       ┆ 10    ┆ 1          │
# │ Z    ┆ C2       ┆ 7     ┆ 1          │
# │ W    ┆ C2       ┆ 3     ┆ 2          │
# └──────┴──────────┴───────┴────────────┘
```

Here, `.rank("dense", descending=True).over("category")` computes a rank per category [29]. Each row's *price_rank* is the rank of that item's price among items of the same category. The output keeps all original rows with the new *price_rank* column.

Window expressions can use any aggregation or expression that makes sense over a group: - **Aggregates:** `pl.mean("col").over("group")`, `pl.max(...).over(...)`, etc. give each row the group's mean, max, etc. - **Row numbering:** Polars provides `pl.col(...).rank()` as above, and also `.cumcount().over("group")` to number rows within group (cumulative count). - **Shifts and diffs:** You can do lag/lead within group by combining `.over` with `pl.col("x").shift(1)` or similar.

By default, `.over("key")` requires that the result of the inner expression has the same length as the group (most built-in aggregations do produce a single value per group, which Polars then broadcasts to each row of the group). If the inner expression yields multiple values per group, Polars needs a mapping strategy – by default it's `"group_to_rows"` which aligns each output value to the corresponding original row [28] [30]. An alternative is `mapping_strategy="explode"`, which instead expands the DataFrame (like a join with the aggregated list) [31]. This is an advanced feature – typically, using standard aggregations yields one value per group, so `"group_to_rows"` just broadcasts it back.

Order in Windows: If you need ordering within the window (like a running total or window frame), Polars does not yet support explicit *frame boundaries* in `.over`. However, there is `Expr.sort_by()` which can sort values by another column within the group *before* an aggregation. For example, to get a cumulative sum of sales ordered by date within each store: `pl.col("sales").cumsum().over("store").sort_by("date")` (note: ensure the DataFrame is sorted by date or include date in the over partition for deterministic results). This is somewhat limited compared to SQL window frames. Development is ongoing to allow sliding window frames (see Polars issue on sliding `over()` windows [32]). For strictly time-based rolling calculations, use the rolling group-by or `pl.Series.rolling_*` methods.

Rolling Window (Moving) Functions: Aside from group-based windows, Polars offers rolling operations for time series or sequence data: - Per Series: `Series.rolling_min(window_size)`, `rolling_max`, `rolling_mean`, etc., and exponential rolling (`ewm_mean`, etc.) for moving averages [33] [34]. - As expressions: `pl.col("y").rolling("ts", period="1h").mean()` to take rolling 1-hour windows on a time column, or `pl.col("val").rolling(window_size=3).sum()`, etc. There is also `group_by_rolling` as mentioned for multiple series grouped by key with time windows.

In summary, Polars covers window functions both in the SQL-like analytic sense (`over`) and the moving-window sense (rolling aggregations). These are all executed in optimized Rust and can be combined with lazy execution for efficiency.

Join Operations

Joining data is a strength of Polars – it supports many join types and executes them in parallel. The primary API is `DataFrame.join(other_df, left_on=..., right_on=..., how="...")`. You can join on one or multiple keys (by providing a list to `left_on` / `right_on`), or even join on expressions.

Equi-Joins: The standard case where keys must exactly match. - Supported `how` options: `"inner"`, `"left"`, `"outer"` (full), `"right"`, plus semi/anti joins: - **Inner join:** only matching rows from both sides are kept [35]. - **Left join:** all rows from left, with matching right data or nulls if no match [36]. - **Full (outer) join:** all rows from both sides, nulls where no match [37]. - **Semi join:** keeps rows from left that have at least one match on right (like a filtering of left by membership in right) [38]. - **Anti join:** keeps left rows that **do not** have a match in right [38]. - **Cross join:** use `how="cross"` to get the Cartesian product of two frames [39]. (No keys needed; `len(result)=len(left)*len(right)`.)

Example inner join:

```
df1.join(df2, on="id", how="inner")
```

If column names differ: `df1.join(df2, left_on="id1", right_on="id2")`.

Polars join implementation is heavily optimized in Rust. By default, it builds a hash table on the smaller input for speed (and may parallelize by partitioning keys). For very large joins, ensure you have enough memory or use streaming if possible (see non-equi join for streaming case). You can force a join to *rechunk* data into one contiguous chunk before joining (to possibly improve performance) via env var `POLARS_NO_CHUNKED_JOIN=0` (actually Polars will rechunk by default unless you set `POLARS_NO_CHUNKED_JOIN` to force skip) [40].

As-of Joins: An *as-of join* matches on *nearest keys* rather than exact. Typically used for time-series (e.g. find latest smaller timestamp in right for each left timestamp). Use `DataFrame.join_asof(other, left_on="time", right_on="time", by="id", strategy="backward/forward")`. By default it finds the nearest earlier or equal key on the right (backward as-of). You can specify `strategy="forward"` to find the next later key. You can also set a `tolerance`. For example:

```
df_trades.join_asof(df_quotes, on="time", by="stock", strategy="backward",
tolerance=datetime.timedelta(minutes=1))
```

This would attach the most recent quote within 1 minute before each trade time [41] [42]. If no quote in that window, result fields are null. As-of join requires sorted data on the key. This is great for aligning events with timestamps (Polars as-of join is analogous to pandas merge_asof, but faster).

Non-Equi Joins: Polars 0.19+ introduces **non-equi joins** via a special `join_where` API. This allows using arbitrary comparison operators in join conditions (not just equality) [43] [25]. For example:

```
result = df1.join_where(df2, pl.col("df1.val") < pl.col("df2.val"))
```

This would pair every row of df1 with every row of df2 where `df1.val < df2.val` [44] [45]. Under the hood, Polars might perform a nested loop or use an interval index depending on the predicate. Because a non-equi join can produce an arbitrary number of matches, the result can be larger than input frames (be careful – a cross-join with a filter is effectively what this does). In Polars 0.20, `LazyFrame.join()` can

accept a condition expression for non-equi (the `join_where` for lazy). The example from Polars user guide:

```
players.join_where(props_prices, pl.col("cash") > pl.col("cost"))
```

pairs each player with all properties cheaper than their cash [26] [45] (yielding a "wish list" of what they can buy). Non-equi joins are currently executed akin to a cross join + filter internally [46], but Polars 1.x will introduce more optimized algorithms for some cases (e.g. sorted merge-join for inequalities).

Join Performance and Memory: By default Polars joins in-memory. It may allocate temporary structures (hash tables, etc.) and if the join is huge, memory can spike. In Polars <=0.20, joins are not yet "streaming". However, you can sometimes simulate a streaming join by sorting data and using as-of join or chunked processing. Polars developers are actively working on streaming and distributed join capabilities (Polars Cloud uses a novel approach for distributed joins [47]). For now, ensure you have enough memory for large joins, or break the problem into chunks (e.g. process each key range separately).

Cross Joins: As noted, `how="cross"` gives the Cartesian product. This can blow up quickly in size, but can be useful for e.g. attaching every combination of two small dimensions.

Null equality: By default, join keys with null do **not** match each other (to mimic SQL null != null). In Polars >=1.0, a parameter `null_equal=True` can be set on join to treat nulls as equal for join matching [48]. Otherwise, any row with null in a join key will only match with null on the other side if `null_equal` is True.

Join summary: Polars covers the standard range of joins and some advanced ones (as-of, non-equi). It's generally faster than Pandas and competitive with SQL engines for join operations due to optimized algorithms and parallelization. Use the appropriate join type for your task, and consider key cardinalities (for big data, joining on high-cardinality keys may be expensive). Always prefer joining on columns rather than computing a cross product manually in Python.

Complex Data Types and Null Handling

Polars is built on Apache Arrow and supports a rich set of data types beyond simple numbers and strings. Notable types:

- **Numeric Types:** Integers (signed/unsigned 8,16,32,64-bit), Floats (32,64-bit), booleans – straightforward. Be mindful of overflow (Polars will not automatically upcast types, so summing a `UInt8` column will stay `UInt8` and could overflow without warning).

- **UTF-8 Strings:** Represented as `Utf8` type (Arrow variable-length UTF8). These are stored efficiently and can be processed with vectorized string operations (`.str` namespace methods on expressions/Series). **Nulls** in string columns are separate from empty string `""` – null means missing. Polars uses Arrow's bitmask for nulls, so null handling is efficient. Most ops propagate null (e.g. concatenating a null yields null). Use `fill_null` or `fill_nan` to replace nulls [49] [50].

- **Categorical (Dictionary-encoded) Type:** Polars has a `Categorical` dtype which stores strings as integer category codes (with an internal mapping). This is like R/Pandas factor or Arrow dictionary array. It's **opt-in**: Polars will not auto-convert strings to categorical unless you explicitly cast or specify `dtype=pl.Categorical`. One can cast: `df.with_columns(pl.col("col").cast(pl.Categorical))`. Categorical is efficient for repeated string values and allows fast comparisons and groupings. However, by default each `Categorical` column has its own mapping (so two categorical columns with the same string values might use different codes) [51] [52]. This means comparing or joining different categorical columns requires either merging categories or enabling a global string cache.

- **Enum (Global Categorical):** New in Polars ≥0.20 is an `Enum` dtype [53]. Enums are like categorical but with a fixed universe of possible values defined upfront. For example: `dtype = pl.Enum(["Red","Green","Blue"])` then `pl.Series(["Red","Blue"], dtype=dtype)`. Enums guarantee the same encoding across all uses of that enum type (so different columns or frames with the same Enum type share codes) [54]. They don't allow any values outside the predefined set (an error is thrown if you try to insert a disallowed value) [55]. Use Enum when you have a known finite set of categories globally – it avoids category merging overhead and is safe for joins without a global string cache.

- **Global String Cache:** Polars offers a mechanism to make normal Categoricals behave globally consistent. By calling `pl.enable_string_cache()` (or using a context `with pl.StringCache():`), any subsequent categorical creations will use a **global mapping** [56] [57]. This allows, for instance, joining two categorical columns from different DataFrames (created under the same cache) by comparing their codes directly [58] [59]. Without it, Polars would have to map categories between columns, which is expensive [60]. The string cache is disabled by default because it introduces a small overhead on category creation and retains a global mapping in memory [61]. Best practice: if you plan to join or concat multiple categorical columns with overlapping values, either enable the string cache during their creation or use the new Enum type for a more explicit approach. Always disable the cache after use (`pl.disable_string_cache()` or exit the context) to avoid unintended memory use.

- **Structs:** Polars can have struct-type columns, essentially nested named fields within a row. A struct column is analogous to a table within a cell, or a JSON object in one cell. You can create a struct by combining columns, e.g. `df.select(pl.struct(["lat","lon"]).alias("coords"))` makes a single *coords* column of type struct with fields *lat* and *lon*. Access struct fields with `.struct.field("fieldname")` in expressions [62] or by `df["coords"].struct.fields` etc. Structs are useful for hierarchical data or when you want to group related columns without fully flattening (e.g. Polars might return a struct for grouped aggregation of multiple fields). They are also used in Polars for methods like `DataFrame.to_struct()` which creates a single series of struct for each row [63]. You can explode a struct into separate columns with `df.unnest("struct_col")` [64]. Under the hood, struct is implemented as multiple parallel arrays, so accessing one field is efficient (no actual nested loop needed).

- **List (Array) Columns:** A List type column holds a variable-length list of values for each row (possibly empty or null). This is equivalent to an array type in Arrow. For example, reading JSON may produce list columns, or you can intentionally collect values into a list per group: `df.group_by("id").agg(pl.col("val").list())` yields a list of all *val* in each group. You can

then manipulate these lists: Polars supports exploding lists (turning each element into a row – `df.explode("list_col")`), slicing lists, or applying expressions to list elements via `list.eval` (e.g. `pl.col("list_col").list.eval(pl.element().mean())` to compute mean of each sublist) [65]. Many functions exist for list: e.g. `.arr.contains(x)`, `.arr.sum()` on Series to operate within each list [66]. Lists are useful for representing one-to-many relationships or aggregated data.

- **Binary (Bytes) Type:** Polars can store binary data (arbitrary byte sequences) in columns of type `Binary` (Arrow's large binary). This is for cases like images, cryptographic hashes, etc. You don't get text-specific operations on binary (though you can use `.bin.contains()` for substring search in bytes, etc. as shown in API docs [67]). Binary columns display as hex by default. They can be created by reading a binary file or explicitly converting (e.g. `pl.Series([b'\x00\xFF', ...], dtype=pl.Binary)`).

- **Object (Python object) Type:** Polars has an `Object` dtype which can hold arbitrary Python objects. **This is intended for niche use** – it's basically a pointer to a Python object for each cell, similar to pandas `object` dtype. Operations on these are not vectorized and mostly unsupported in Polars (you typically can only carry them around or use `apply` which will see Python objects). By default, Polars does not allow creating Object series unless you enable an unsafe flag `POLARS_ALLOW_EXTENSION=1` [68]. This type is included primarily to support the DataFrame interchange protocol or for users to store custom data without Polars touching it. In general, avoid Object type if possible – use proper numeric or list types for performance.

- **Dates, Datetimes, and Time/Durations:** Polars has rich temporal types: `Date` (daily precision), `Datetime` (timestamp with optional timezone), `Time` (time of day), and `Duration` (time interval). These are stored as 32 or 64-bit integers (e.g. Unix epoch milliseconds or nanoseconds) under the hood, but Polars displays and operates on them with calendar awareness. Use `.dt` namespace for date/time methods (extracting year, month, day, etc., or rounding, offsetting, etc.). Timezone-aware datetimes are supported (stored as UTC with tz info tracked). Arithmetic between datetime and duration yields another datetime, etc. (Polars uses Arrow chrono).

- **Null Handling:** Polars uses a sentinel null value (like SQL NULL) with a validity bitmap. Any column except `Boolean` and `Binary` can have nulls (and even those can, but are less common). In expressions, **nulls propagate** by default (e.g. `null + 5 = null`). Many operations have an option to ignore nulls or fill them:

  - Use `fill_null(value)` to replace nulls with a value or strategy (`pl.mean("col").fill_null(0)`).
  - Use `drop_nulls()` to remove null-containing rows.
  - Predicates treat null as False for filtering (so `df.filter(pl.col("x") > 0)` will drop rows where x is null, since the comparison yields null which is not True).
  - For reducing operations like sum/mean, Polars will by default treat all-null input as null (or return null if any null in sum unless you use `ignore_nulls=True` in the expression).

There is a special `Null` dtype if an entire Series is nulls, but usually Polars will infer a type in mixed situations. If you create a Series from all None Python values, Polars may label it as `Float64` or `Utf8`

depending on context, or `Null` type if it truly cannot infer anything. You can explicitly use `pl.Series([None, None], dtype=pl.Int32)` to get a nullable Int32 full of nulls.

Casting and Schema: You can cast between types with `pl.col(...).cast(new_type)`. If Polars cannot represent a value in the target type, it will emit a null (e.g. casting `"abc"` to Int64 yields null). Use `strict=True` in cast to instead throw an error on invalid cast. Schema inference on file reading will pick a type that can accommodate all parsed values (e.g. a CSV column that looks like all ints might be inferred Int64, but if a float appears later it may error unless you set `infer_schema_length` or specify dtype). When reading data, you can pass a `dtype` dict to `read_csv` etc. to force types.

User-Defined Functions (UDFs) and Custom Operations

While Polars' expression API covers a lot of cases, sometimes you need to apply custom Python code. Polars provides several mechanisms for this, but they should be used sparingly – they run Python code and thus forfeit many of Polars' performance advantages. Always prefer a vectorized expression if possible, as UDFs in Python will be **much slower** [69] [70].

Element-wise UDFs – `map_elements`: You can apply a Python function to each element of a column via `Expr.map_elements(func)`. For example:

```
# Pseudocode: classify values as "big" or "small"
df.select(pl.col("x").map_elements(lambda val: "big" if val > 1000 else
"small").alias("size"))
```

This will call the lambda on each value of column *x*. **Important:** This is essentially a Python loop – Polars will pass each value to the function one by one (or in small batches, but each invocation is a Python function call) [71] [70]. It's much slower than any built-in vectorized operation. Polars will attempt to infer the return type by calling your function on a dummy value, but it's best to provide `return_dtype` explicitly to avoid ambiguity [72]. By default, null values are skipped (`skip_nulls=True` means if an input is null, Polars just outputs null without calling your function) [73]. You can set `pass_name=True` to also pass the column name to your function (rarely needed) [74]. There is also a `strategy` parameter: `'thread_local'` (default) runs the UDF on a single thread (to avoid GIL issues), `'threading'` attempts to run multiple UDF instances in parallel threads (only beneficial if your function releases the GIL or is heavy) [75]. Polars will issue a **PerformanceWarning** when you use map_elements, reminding you it's inefficient [70].

Batch UDFs – `map_batches`: Rather than element-wise, you can apply a function on whole columns or multiple columns at once using `Expr.map_batches(func)`. The function you supply should accept a **Series or sequence of Series** and return either a Series or numpy array or scalar. For example:

```
# Use numpy to compute something for each column in a DataFrame
df.select(pl.all().map_batches(lambda s: s.to_numpy().argmax()))
```

In this example (from Polars docs), the lambda receives each column as a Series `s`, converts to numpy and finds the index of the max element [76]. The result is a single-row DataFrame giving the argmax of each

selected column [77]. `map_batches` can be used on an expression that yields multiple columns (like `pl.all()` yields each column). You can also use it in a group_by context (see below). Key points: - If your function returns a scalar for the whole batch, you might want `returns_scalar=True` so Polars knows not to wrap it in a list [78] [79]. - `return_dtype` is important to set if Polars can't infer it. - There is an `agg_list` flag: if `True` in a groupby, Polars will aggregate the group into a list and call your function once per group (instead of on each value) [80]. - `is_elementwise`: if you set `is_elementwise=True` and your function truly behaves like elementwise, Polars may allow it in streaming (use with care; misusing it can give wrong results in groupby) [81].

In simple terms, `map_batches` gives you a vector (or table) at a time to operate on, instead of single values. This amortizes Python overhead better than map_elements. For example, using `map_batches` to call a numpy vectorized operation can be far faster than map_elements calling a pure Python math on each element. But still, **native Polars expressions are faster** whenever available.

Row-wise UDFs – DataFrame.apply / map_rows: If your operation inherently needs to consider multiple columns at once in a Python function for each row (like a custom combination or non-vectorizable logic), you can use `df.apply(func, return_dtype=None)`. This will feed each row as a tuple of Python values to your `func` and collect the results. `DataFrame.apply` (and its equivalent method `DataFrame.map_rows`) **executes in Python for each row** [82] [83]. This is the least efficient approach (even slower than map_elements typically), because it involves potentially millions of Python function calls for a large frame. It also *materializes the whole DataFrame in memory* (because it must iterate over rows in Python) [84]. Use it only if absolutely necessary. If the function returns a tuple or list with the same length each time, Polars will turn that into multiple columns. If it returns a single value, you get a Series.

For example:

```
# Concatenate two column values and a constant per row (as a tuple)
df.apply(lambda row: (row[0] + "_" + row[1], 42), return_dtype=None)
```

This might produce two output columns (one for the string, one for the constant 42). But doing the same with an expression would be better: `df.with_columns((pl.col("col1") +"_"+pl.col("col2")).alias("concat"), pl.lit(42).alias("const"))` – no Python loops needed.

Polars explicitly warns that `map_rows/apply` are slow and that **"wherever possible you should strongly prefer the native expression API"** [84] [85]. It notes the reasons: Rust native is parallel and optimized, Python UDFs force single-threaded execution and break optimizations [84]. Indeed, using `df.apply` will prevent predicate pushdown, projection pruning, etc., because Polars treats the UDF as an opaque black box with all columns in use [86].

One limitation: `DataFrame.apply` cannot preserve custom column names if you produce a tuple – Polars will name them `column_0, column_1, ...` by default [86]. And you must often specify `return_dtype` if it cannot infer.

Group-wise UDFs: Within a grouping, Polars provides: - `GroupBy.apply(func)` – applies a function to each group *as a DataFrame* and expects a Polars DataFrame back which Polars will stack together. This is similar to Pandas groupby-apply. For instance, `df.group_by("category").apply(lambda subdf: subdf.sort("value").tail(2))` might take the last 2 rows of each category. This is powerful but internally it invokes the Python function once per group, which can be slow if many groups. If the function returns a different schema or number of rows per group, Polars has to reconcile that (which it can do since it gets actual DataFrames). Use this only when you need to do something per group that you can't express as an aggregation or `.over`. - `GroupBy.map_groups(func)` – similar to apply, likely identical in implementation for Python (the API differentiate that `apply` returns a DataFrame that is concatenated, whereas `map_groups` expects exactly the same schema for each group and then does the concatenation; in practice both give a new DataFrame). The **Polars docs caution** that using group apply will not be multi-threaded and is not as efficient as built-ins [87] (Polars favors its native per-group aggregations over Python grouping). - You can also use expression `pl.col(...).map_batches(..., agg_list=True)` inside a `.agg()` to process groups in one go (the `agg_list=True` will send the entire group Series to your function) [80]. This can sometimes be faster than group_by.apply since it stays in the lazy/expression context.

When to use these UDFs: - Use **map_elements** only for something truly element-specific that Polars lacks (e.g. a complex math function from Python's stdlib or a lookup in a Python dict). - Use **map_batches** when you can leverage a vectorized library (numpy, etc.) on chunked data for speed. - Use **apply/map_rows** as a last resort for highly irregular logic that can't be expressed with vector operations (but consider if you can reshape your data or use `.join()` or `.over()` to avoid it).

Warning about stability: The Polars team labels some UDF features as *unstable* (not in terms of crashing, but API might change). For example, `map_elements` might change its internals without a major version bump [88], and they might add new UDF mechanisms (like user-defined Rust functions via plugins, which is an advanced topic beyond this reference).

In summary: **exhaust Polars' built-in functions first** (string ops, list ops, etc.). If needed, use a UDF on batches. Only drop down to per-row Python loops if there is absolutely no other way. Remember that UDFs break Polars' optimizations: e.g. a filter after a UDF can't be pushed down; the UDF will run on all rows. So sometimes it's better to filter first in Polars, then apply UDF on a reduced DataFrame.

File I/O and Data Sources

Polars can read and write a variety of formats. It offers both **eager** `pl.read_*` functions that immediately produce a DataFrame, and **lazy** `pl.scan_*` functions that create a LazyFrame (to allow query pushdown on the source).

CSV/TSV: - `pl.read_csv("file.csv", ...)` reads a CSV into a DataFrame. It has many options (delimiter, column names, dtypes, null values, skip rows, etc.). It will try to infer dtypes by default from the first N rows (`infer_schema_length`, default 100). If your file is large, you might set a larger sample or specify dtypes to avoid incorrect inference. - `pl.scan_csv("file.csv", ...)` creates a LazyFrame for a CSV. This is great for large files – you can then apply filters/selects and Polars will only parse the necessary columns and rows. For multiple files, you can use glob patterns (e.g. `pl.scan_csv("data/part*.csv")`) – Polars will treat them as one dataset (with an implicit union) [89] [90]. - Writing CSV:

`df.write_csv("out.csv")`. Options for including header or not, etc. Polars CSV writing is quite fast (leveraging SIMD if possible). For streaming large outputs, Polars offers `LazyFrame.sink_csv("out.csv")` [91] – this will execute the lazy query and write out in one pass, without holding the whole result DataFrame in memory.

Parquet (Apache Parquet): - `pl.read_parquet("file.parquet")` reads a Parquet file (or list of files) eagerly. Polars uses its Rust native Parquet reader by default, which is super fast. It can also pushdown column selection (only loads columns needed). Parquet stores schema and data in column chunks; Polars will automatically use that to infer schema precisely (no guessing). - `pl.scan_parquet("file.parquet")` returns a LazyFrame for Parquet. This enables predicate pushdown (Polars will use Parquet metadata and statistics to skip row groups that don't satisfy filters) [92] [93]. It also can intelligently avoid reading row groups of Parquet that aren't needed. This is a key advantage: e.g., scanning a large partitioned Parquet dataset with a filter on partition column will avoid opening many files altogether. - Writing Parquet: `df.write_parquet("out.parquet", compression="zstd", statistics=True)` etc. Polars supports common compressions (snappy, gzip, zstd) and by default writes statistics (min/max per column per row group) which help predicate pushdown on read. For lazy, `LazyFrame.sink_parquet("out.parquet")` will stream the query result directly to Parquet file [94]. - For cloud sources (S3, etc.), Polars 0.19+ supports reading by providing an `fsspec` file path or using `aws://` style paths, etc., and even does **asynchronous prefetch** for Parquet. As of 0.20, native cloud readers for AWS/GCP/Azure are integrated [95].

Arrow IPC (Feather/Arrow file): - `pl.read_ipc("file.arrow")` or `.read_ipc_stream("file.arrow")` loads Arrow IPC format (also known as Feather v2) [96]. This is a binary columnar format very similar to Parquet but without compression by default (though v2 supports compression). It's good for lightning-fast reads if the data is small-medium. - `pl.scan_ipc("file.arrow")` gives a LazyFrame. Note that Arrow IPC doesn't have row group stats like Parquet, so predicate pushdown may not skip data (though column projection still applies). - Writing: `df.write_ipc("out.feather")`. `LazyFrame.sink_ipc(...)` exists as well [97]. Arrow IPC is useful for data interchange because it's zero-copy convertable to pyarrow and other Arrow consumers.

NDJSON (Newline-delimited JSON): - `pl.read_ndjson("file.ndjson")` can read a file where each line is a JSON object. This yields a DataFrame with columns for each JSON field (it will infer types, potentially creating lists or structs for nested JSON). - `pl.scan_ndjson("file.ndjson")` lazy scan is supported as well [98]. JSON is more expensive to parse, and Polars cannot push down predicates into NDJSON easily except at the line granularity (no statistics). But Polars will still do projection pruning (if you only select certain fields, it will avoid materializing others). - Writing JSON: `df.write_json("out.json")` for whole-document JSON, or `df.write_ndjson("out.ndjson")` for line-delimited. Polars writing of NDJSON will output each row as a compact JSON object.

Other Formats: - **Excel:** Polars has experimental reading for XLSX/ODS via `pl.read_excel` / `read_ods` (requires installing `xlsx2csv` or equivalent). These are not as fast and have fewer options. - **Avro:** `pl.read_avro` and `write_avro` if compiled with Avro support [99]. - **Delta Lake / Iceberg:** Polars can scan those if you have feature flags enabled (like `pl.scan_delta("table_path")` for a Delta Lake table, using DataFusion under hood). In 0.20 this is still experimental. - **Database connections:** `pl.read_database(uri, query)` can run a SQL query in a database and pull results into a Polars

DataFrame (through connectorx). Similarly `pl.read_sql(sql, connection)` via ad-hoc. For writing, `df.write_database(table_name, connection)` to push to a DB (with some limitations).

Polars aims to integrate with many data sources. The general rule: - Use **scan_** *functions for large data, to leverage lazy pushdowns (only supported by Parquet, CSV, Delta, etc. where Polars can optimize; less useful for JSON). - Use read_* **for quick loading of moderate data or if you immediately need the DataFrame. - For** streaming output**, the** sink_*** functions (on LazyFrame) allow you to execute a pipeline and directly stream-write the result to disk or network without ever creating a giant DataFrame in Python. This is memory-efficient for big data output.

Memory mapping: Polars does not memory-map files by default except perhaps in some optimized paths. PyArrow sometimes memory maps Parquet, but Polars' native reader reads into memory buffers. For large binary data, ensure you have enough memory or use streaming to process chunk by chunk.

Schema Evolution: If you read multiple files (like partitioned data) with inconsistent schema, Polars will try to merge them (e.g., missing columns get filled with nulls). You can also use `pl.scan_<format>(..., schema=...)` to define expected schema.

SQL Query Interface

Polars provides an **SQL query layer** on top of DataFrames, which is useful for those who prefer SQL syntax or integrating with SQL-based tools. This is available in two ways: - **DataFrame.sql() method:** You can call `df.sql("SELECT ... FROM self ...")` on a DataFrame. The DataFrame will be registered as table name `"self"` (or you can give a custom name via `table_name` parameter) [100] [101]. The SQL query string is then executed by Polars' SQL engine and the result is returned as a new DataFrame. - **Global SQLContext:** You can use the top-level `pl.SQLContext` to register multiple DataFrames and execute joins or queries across them. Example:

```
ctx = pl.SQLContext()
ctx.register("df1", df1)
ctx.register("df2", df2.lazy())  # can register lazy or eager frames
result = ctx.query("SELECT df1.foo, df2.bar FROM df1 JOIN df2 ON df1.id =
df2.id")
```

The `ctx.query()` returns a DataFrame (it collects the lazy result).

Polars' SQL syntax aims for compatibility with standard SQL. It supports SELECT, WHERE, GROUP BY, HAVING, JOIN, etc., using Polars under the hood: - You can use functions like `COUNT`, `MIN`, `MAX`, `SUM`, `AVG` in the SELECT. - It has some SQL-specific functions mapped to Polars expressions: e.g. `EXTRACT(year FROM date_col)` [102], `CONCAT_WS(':', col1, col2)` to concatenate with separator [103], type casts like `0::float4` to cast 0 to float (as seen in example) [102]. - Filter (WHERE) and expressions in SELECT are translated to Polars expressions. - Joins use the usual SQL JOIN syntax. - `LIMIT n` is supported (translates to `.head(n)`). - Aliasing tables: as shown, you can alias the implicit table using `table_name` in DataFrame.sql (in the example they alias "self" to "frame" and use `FROM frame` [104]).

The SQL engine in Polars is built on the lazy API. A SQL query is internally parsed to a logical plan of Polars operations. That means all the Polars optimizations (pushdown, etc.) apply to SQL queries as well [101]. The execution is lazy until the result is materialized as a DataFrame.

Stability: As of Polars 0.20, SQL support is marked **experimental** (close to stable, but API could still change a bit) [105]. Not all SQL features are present – e.g., window functions in SQL syntax may not all work (though you can always mix Polars expressions after using SQL for part of the query). The basics of SELECT/JOIN/AGG are solid.

Use cases: SQL interface is handy for quick queries or for users coming from SQL background. It also allows integration with tools expecting an SQL engine (Polars can act as a backend for something like DuckDB replacements in certain scenarios). But note that not everything in Polars has SQL equivalent yet (for complex Polars-specific operations, you'd use the regular API).

Example:

```
df = pl.DataFrame({
    "a": [1, 2, 3],
    "b": ["x", "y", "z"],
    "c": [10.0, 20.5, 30.5]
})
# Query using SQL string:
res =
df.sql("SELECT a, CONCAT(b, '-tail') AS b_mod, c * 2 AS double_c FROM self WHERE
c > 15")
print(res)
# shape: (2, 3)
# ┌─────┬────────┬──────────┐
# │ a   ┆ b_mod  ┆ double_c │
# ╞═════╪════════╪══════════╡
# │ 2   ┆ y-tail ┆ 41.0     │
# │ 3   ┆ z-tail ┆ 61.0     │
# └─────┴────────┴──────────┘
```

Here `self` refers to the DataFrame. We used a SQL CONCAT (Polars SQL uses the `CONCAT(x, y)` or the `||` operator alternatively for string concat) and an arithmetic expression. Polars SQL is case-insensitive for keywords, but case-sensitive for column names.

In summary, Polars' SQL is an added layer on the same engine. It's useful, but if you're comfortable with the Polars API, you can achieve everything (and sometimes more) without it. Keep in mind the SQLContext for multi-frame queries (or use DataFrame.join which may be more straightforward).

Interoperability (Arrow, Pandas, NumPy, etc.)

One of Polars' strengths is that it's built on the Apache Arrow columnar format. This enables zero-copy or low-copy interoperability with many other data tools.

- **Arrow Tables:** You can seamlessly convert between Polars DataFrames and PyArrow Tables:
- `df.to_arrow()` gives a `pyarrow.Table` with zero copy (the Polars data is essentially handed off as Arrow arrays) [106] . This means you can use PyArrow or send the table to other libraries without overhead.
- `pl.from_arrow(pa_table)` creates a Polars DataFrame from an Arrow table/array efficiently (zero-copy for supported types) [107] [108] . It will reuse the Arrow memory where possible. Unsupported types might be cast or cause an error.
- This also means you can use Arrow datasets with Polars – e.g., read a dataset via PyArrow and then convert to Polars, or vice versa.

- Polars supports Arrow's extension types for temporal data etc., usually mapping to its own dtypes.

- **Pandas DataFrames:** Polars can convert to/from pandas:

- `df.to_pandas()` will return a pandas DataFrame. This involves converting each column to pandas (which may be a copy; columns of primitive types can often be zero-copy by sharing the memory buffer with a suitable pandas Series if all else aligns, but currently Polars likely copies data to pandas to be safe).
- `pl.from_pandas(pd_df)` or simply `pl.DataFrame(pd_df)` will ingest a pandas DataFrame. Under the hood, Polars may use the Arrow interchange (if pandas supports the `__dataframe__` protocol) or convert column by column. Data will be copied for safety in most cases (e.g., object dtypes or categorical in pandas will be converted to Polars types).
- **Arrow Interchange Protocol:** Polars implements the new `__dataframe__` protocol on DataFrames [109] . If a library (like pandas 2.0+) implements this, Polars `from_dataframe` can import it zero-copy. `pl.from_dataframe(obj)` is a general constructor for any object implementing the exchange protocol [110] . This is future-proofing for smooth interchange without overhead.

- Note: When converting large DataFrames, memory usage will temporarily increase (because you'll have two copies). So prefer using scan (if from a file) or Arrow direct if possible.

- **NumPy:** Polars can work with NumPy arrays in a few ways:

- You can create a Series from a numpy array: `pl.Series(name, numpy_array)`. Polars will zero-copy if the dtype is compatible (e.g. a float64 numpy array becomes a Float64 Series by sharing the buffer). If not compatible (like a structured dtype), it might raise or copy.
- `df.to_numpy()` returns a 2D numpy ndarray of the DataFrame values (this will always copy, and if DataFrame has multiple dtypes, it will cast to a common type, often object, so it's not very efficient or recommended for large frames) [111] .
- `pl.from_numpy(array, schema=...)` can create a DataFrame from a 2D numpy array or from a structured array by interpreting it as columns [112] . There is also `pl.from_dict` and `pl.from_records` if you have Python data.

- In expressions, you might use numpy within `map_batches` as shown, but Polars doesn't automatically integrate with numpy beyond conversion. However, because Polars uses Arrow memory, you can get a pointer (`Series.to_numpy()` returns a view if possible) [113].

- **PyTorch / Jax / TensorFlow:** Polars can export to these:

- `df.to_numpy()` and then to torch is one way. But Polars directly offers `df.to_pytorch()` or `df.to_torch()` (Polars 0.19 added experimental torch integration) [114]. Essentially this gives you a Torch Tensor with a copy of the data. Similarly, `df.to_jax()` for JAX device array.

- These are convenience methods; performance-wise, similar to converting to numpy then to tensor.

- **SQL Databases:** While not exactly in-memory interoperability, Polars can send/receive data to DBs. We touched `read_database` and `write_database`. Polars also works nicely with DuckDB: you can do `duckdb.query(df)` thanks to Arrow conversion (DuckDB can query Arrow tables directly). Conversely, pulling a DuckDB result as Polars: `pl.from_arrow(duckdb_df.arrow())` can be done.

- **Polars <-> Other DataFrames (Ibis, Pandas API, etc.):**

- *Ibis:* Ibis is a Python dataframe-like API that can target multiple backends (SQL, Spark, etc.). Polars is one backend for Ibis (contributed recently), meaning you can write Ibis expressions and execute with Polars engine. This is advanced usage for people wanting abstraction.
- *Pandera:* Pandera is for validation; it now supports Polars schema checks as well.
- *Dask/Ray:* There's no built-in Polars distributed, but you can easily use Polars inside tasks of Ray or Dask. E.g. use Dask to partition file reading, and each partition is a Polars DataFrame (since Polars DataFrame is picklable, you can send it between processes). Just remember not to use fork (see below).

In summary, Polars plays well in the data ecosystem. The zero-copy conversion to Arrow is particularly useful: any tool that speaks Arrow (PyArrow, DuckDB, Turicreate, etc.) can send data to/from Polars without serialization overhead [115] [116]. This makes Polars a good choice for embedding in pipelines where data comes from different sources.

Configuration and Tuning

Polars is largely automatic, but there are some configuration options and environment variables that advanced users can tweak:

- **Number of Threads:** By default Polars uses all available CPU cores for parallel operations [117]. You can override this by setting the environment variable `POLARS_MAX_THREADS` **before importing polars** [118] [119]. For example, in bash: `POLARS_MAX_THREADS=4 python script.py`. Or in Python:

```
import os; os.environ["POLARS_MAX_THREADS"] = "4"; import polars as pl
```

Once Polars is loaded, the thread pool size is fixed (you can query it with `pl.thread_pool_size()` [120]). There isn't a public API to change threads at runtime (aside from env var at startup). Generally, leaving it default is fine; if you run multiple Polars queries concurrently or share CPU with other tasks, you might lower it.

- **Memory and Out-of-Memory:** Polars doesn't have a single switch for memory usage. However, for streaming, it by default uses up to 80% of RAM and will spill to disk if needed for certain operations like sorting (this is internal, you might see large temp files if a streaming group-by or sort runs out of memory). There are environment variables to control some memory aspects:

- `POLARS_FORCE_PARTITION` / `POLARS_NO_PARTITION`: to control grouping strategy as mentioned (force or prevent partitioned algorithm) [12].
- `POLARS_PARTITION_UNIQUE_COUNT`: threshold for switching groupby algorithm [121].
- `POLARS_NO_PARQUET_STATISTICS`: if set, Polars will ignore Parquet stats (disabling predicate pushdown via metadata) [122].
- `POLARS_FMT_MAX_ROWS`, `POLARS_FMT_MAX_COLS`: limits for DataFrame `print()` output (how many rows/columns to print) [123]. Also various styling toggles (`POLARS_FMT_TABLE_*`) to control table drawing characters [123] [124].

- There isn't a built-in disk spilling for non-streaming mode except what OS does for virtual memory. So handle large data with streaming or ensure enough memory.

- **String Cache:** Controlled via `pl.enable_string_cache()` / `pl.disable_string_cache()` or context. No env var for default; by default it's off. There was discussion to enable by default, but it remains opt-in due to the performance cost if not needed [125].

- **Logging and Debug:**

- `POLARS_VERBOSE` env var: if set (`=1`), Polars will print query plan optimizations and other verbose logs to stderr during execution [13].
- `POLARS_PANIC_ON_ERR`: if set, Polars will `panic!` (crash) on internal errors instead of catching them and raising Python exceptions [126]. Not generally recommended, but could help to get Rust backtraces.

- `POLARS_BACKTRACE_IN_ERR`: if set, error messages might include a Rust backtrace [126] – useful for debugging Polars internals issues.

- **Chunk Size (Streaming):** As seen earlier, `pl.Config.set_streaming_chunk_size(n)` can manually set the chunk batch size for the streaming engine [127] [128]. The default formula Polars uses is based on columns and threads [129]; you might tune it if you find performance issues (e.g., maybe larger chunks for better compression or smaller for lower memory spikes). This is an advanced lever.

- **Temporarily Toggle Settings:** There is a `pl.toggle_string_cache(True/False)` as shorthand for enable/disable. Also `pl.Config` exposes some other settings: for example, `pl.Config.set_fmt_str_len(n)` to set how many chars of string to print in one cell, `pl.Config.set_tbl_cols(n)` and `set_tbl_rows(n)` to set the max columns/rows for printing (these correspond to the FMT env vars, but can be changed at runtime via API).

- **Thread Pool Global:** Polars uses a global thread pool for CPU tasks. If you use Polars in a multi-processing scenario, each process will have its own Polars thread pool. Polars is designed not to work with fork (the thread pool doesn't survive fork safely). If you must use multiprocessing, see the next section for guidelines.

- **Installing optimizations:** If you compile Polars from source, enabling `RUSTFLAGS="-C target-cpu=native"` can give CPU-specific speedups [130]. The official wheels are compiled with quite broad SIMD feature support, so you typically get good performance out-of-box.

- **No GIL issues:** Because Polars releases the GIL on heavy computations, you can run multiple polars operations in separate threads (from Python's perspective) concurrently. However, since Polars already multithreads internally, it's usually more effective to run one big Polars job at a time.

In short, Polars defaults are sane for most use. Only tweak thread count if needed or string cache when doing multi-DataFrame categorical ops. Use the logging flags to debug performance. And note printing options to avoid overwhelming outputs.

Best Practices for Large-Scale and Distributed Processing

Polars is a single-node, multi-threaded engine – it excels on one machine with multiple cores and lots of RAM. To handle very large data or to integrate Polars in distributed workflows, consider these practices:

- **Use Lazy Execution and Streaming:** For large datasets (gigabytes to terabytes), always prefer `pl.scan_*` to create a lazy query, apply your filters/projections/aggregations, and then collect. This ensures Polars can push down operations to reduce data early [10]. If data doesn't fit memory, try `.collect(engine="streaming")` to process chunk by chunk. The streaming engine allows Polars to handle datasets larger than RAM by not materializing the whole result at once [1] [131]. For example, you could stream through a 100GB CSV to compute summary stats, only holding one group's chunk in memory at a time.

- **Limit Data in Memory:** Take advantage of projection pruning – select only needed columns. Read data in partitioned form if possible (Polars can accept a directory of files with wildcards; if those files are partitioned by some key, include that key in your filter so Polars only globs relevant files). For instance, if you have yearly Parquet files and only need 2020, do `pl.scan_parquet("data/2020/*.parquet").filter(pl.col("year")==2020)` – Polars will only open that directory (or you can programmatically select files).

- **Divide and Conquer:** If an operation cannot be done streaming (e.g. a full sort or certain joins), you might need to manually break data. For example, sorting 100 million rows might OOM. If you can sort within partitions and then merge (like external merge sort), you can do that manually: sort each chunk with Polars, write to disk, then use a streaming merge (Polars doesn't yet have an external sort API, so you might do that in Python or external tool). Similarly, for a very large groupBy, consider if you can group in chunks and combine results (though that's complex for accurate results except for distributive aggregates like sum).

- **Use Multiple Processes (carefully):** Polars cannot natively distribute across machines or processes by itself. But you can use Python multiprocessing or frameworks like Ray/Dask to launch multiple Polars tasks on partitions of data. Key point: use **spawn method** for multiprocessing (on Unix) because Polars is multithreaded and will deadlock if you fork after starting threads [132] [133]. The Polars guide explicitly says to use `multiprocessing.get_context("spawn")` [134] [135]. This will start fresh Python processes. For example, if you have 4 CSV files, you could `spawn` 4 processes each doing `pl.read_csv` and some computations, then combine results (maybe using Polars concatenation at the end). Each process will utilize multiple threads too, so be careful not to oversubscribe CPU (you might also limit `POLARS_MAX_THREADS` per process in that case).

- **Avoid data copy between processes:** If using multiprocessing, consider writing intermediate results to disk in a format like IPC or Parquet and reading them in the main process, rather than sending large DataFrames through pickling (which copies data). Polars DataFrames are picklable via Arrow, but for huge data the overhead can be high.

- **Distributed execution:** If one truly needs multi-node distribution, Polars by itself doesn't manage that. However, Polars Cloud (a hosted solution by Polars team) or using Polars via Ibis on a cluster are options. In open source, you could orchestrate with Ray: e.g., use Ray tasks to each handle a chunk with Polars, then aggregate. Or with PySpark: Polars can't directly replace Spark's execution, but you could use Spark to partition data and Polars to process each partition (not common, but possible through UDF).

- **Memory tuning:** On large scale, you might hit memory limits. Some tips:

- Turn off categorical global cache if not needed to save memory.
- Use `with_columns(pl.col(...).cast(desired_type))` to downcast data if possible (e.g., if an int64 ID only needs 32 bits, cast it).
- After heavy filtering, use `df.shrink_to_fit()` to free unused capacity [136] or in lazy, do `.collect(rechunk=True)` (default) to compact memory. Polars' **rechunking** will consolidate chunked memory into contiguous arrays [11]. Many small chunks can use more memory overhead; use `df.rechunk()` if your DataFrame ends up with many chunks after certain operations. (Note: Polars automatically rechunks after reading multiple files or concatenating, unless you set `rechunk=False` on scan to optimize skipping that cost [137]).
- If doing extremely large joins/sorts, ensure you have a large temporary disk for OS swap or consider an algorithm that avoids holding all data (not trivial; Polars might auto-spill in some cases like sort as mentioned, but don't rely on it for all operations in 0.20).

- **Monitoring:** When running Polars on large tasks, monitor CPU and memory. Polars will try to use all cores (100% CPU usage is normal during heavy compute). If you see memory climb too high, identify which operation is the culprit (maybe a huge group by or join). You might break that operation down or increase swap.

- **Parallel file reading:** Polars will parallelize reading within a single file (especially Parquet – reading multiple row groups concurrently). If you have multiple files, Polars will also distribute work across threads. So often you don't need to manually multi-process file reading; a single scan_parquet on a directory will utilize threads across files.

- **Stability on long runs:** Polars is generally stable for long-running processes. It doesn't have known memory leaks in typical use (any leaks would be considered bugs). But releasing memory to OS: Polars uses the memory allocator (jemalloc on Linux by default [138]) which might hold onto memory even after `df` is dropped (for reuse). If you need to free memory mid-process, you might force Python GC or delete objects, but ultimately the allocator might keep it reserved. Designing pipelines to complete and exit (or use multiple processes for different stages) might be necessary for extremely memory-sensitive workflows.

In summary, **scale with Polars** by leveraging lazy streaming, partitioning data, and using multiple processes only when the single-process multi-thread approach isn't enough. Many "big data" problems can be handled on one machine with Polars if you carefully filter and stream; for those that truly require cluster scale, Polars can still be a component (for per-node processing or in new distributed Polars variants), but out-of-the-box it's not a distributed engine.

Sources: The details in this reference are drawn from the Polars official user guide and API documentation [53] [26], Polars blog posts [92] [11], and the library's source and community knowledge. They reflect Polars version 0.20.x and some upcoming 1.x behaviors. For further information, consult the latest Polars documentation and community discussions.

---

[1] [3] [5] [6] [131] Streaming - Polars user guide
https://docs.pola.rs/user-guide/concepts/streaming/

[2] [12] [13] [33] [34] [40] [65] [68] [121] [122] [123] [124] [126] [130] [138] polars - Rust
https://docs.rs/polars/latest/polars/

[4] [24] [48] [49] [50] [64] [109] [136] polars.DataFrame.join — Polars documentation
https://docs.pola.rs/py-polars/html/reference/dataframe/api/polars.DataFrame.join.html

[7] [8] [89] [90] [117] [119] [127] [128] [129] Crucial parameters for streaming in Polars | Rho Signal
https://www.rhosignal.com/posts/streaming-chunk-sizes/

[9] [100] [101] [102] [103] [104] [105] polars.DataFrame.sql — Polars documentation
https://docs.pola.rs/py-polars/html/reference/dataframe/api/polars.DataFrame.sql.html

[10] [11] [53] [54] [55] [92] [93] [95] [137] Polars — Polars in Aggregate: Going from 0.19.0 to 0.20.2
https://pola.rs/posts/polars_in_aggregrate-0.20/

[14] [15] [62] [69] [70] [71] [72] [73] [74] [75] [88] polars.Expr.map_elements — Polars documentation
https://docs.pola.rs/api/python/dev/reference/expressions/api/polars.Expr.map_elements.html

[16] [17] [82] [83] [84] [85] [86] polars.DataFrame.map_rows — Polars documentation
https://docs.pola.rs/py-polars/html/reference/dataframe/api/polars.DataFrame.map_rows.html

[18] [19] [20] [21] [22] [63] [106] [114] Python API reference — Polars documentation
https://docs.pola.rs/py-polars/html/reference/

[23] [25] [26] [35] [36] [37] [38] [39] [41] [42] [43] [44] [45] Joins - Polars user guide
https://docs.pola.rs/user-guide/transformations/joins/

27 28 29 30 31 Window functions - Polars user guide
https://docs.pola.rs/user-guide/expressions/window-functions/

32 Sliding window for pl.Expr.over function · Issue #8976 · pola-rs/polars
https://github.com/pola-rs/polars/issues/8976

46 Non-equi join in polars - python - Stack Overflow
https://stackoverflow.com/questions/78922047/non-equi-join-in-polars

47 Polars — Polars Cloud; the distributed Cloud Architecture to run ...
https://pola.rs/posts/polars-cloud-what-we-are-building/

51 52 56 57 60 61 Categorical data and enums - Polars user guide
https://docs.pola.rs/user-guide/expressions/categorical-data-and-enums/

58 125 Enable string cache by default · Issue #9106 · pola-rs/polars - GitHub
https://github.com/pola-rs/polars/issues/9106

59 py-polars.enable_string_cache() does not simply turn on/off #10425
https://github.com/pola-rs/polars/issues/10425

66 67 91 94 96 97 98 99 polars.map_groups — Polars documentation
https://docs.pola.rs/api/python/version/0.19/reference/expressions/api/polars.map_groups.html

76 77 78 79 80 81 polars.Expr.map_batches — Polars documentation
https://docs.pola.rs/docs/python/dev/reference/expressions/api/polars.Expr.map_batches.html

87 为什么polars的group_by.apply似乎没有用多进程计算？ - 知乎
https://www.zhihu.com/question/665915695/answer/3613185549

107 108 113 polars.from_arrow — Polars documentation
https://docs.pola.rs/py-polars/html/reference/api/polars.from_arrow.html

110 111 112 polars.from_dataframe — Polars documentation
https://docs.pola.rs/py-polars/html/reference/api/polars.from_dataframe.html

115 Polars — DataFrames for the new era
https://pola.rs/

116 PyArrow vs Polars (vs DuckDB) for Data Pipelines.
https://www.confessionsofadataguy.com/pyarrow-vs-polars-vs-duckdb-for-data-pipelines/

118 How to limit the number of threads in Polars - Stack Overflow
https://stackoverflow.com/questions/71179317/how-to-limit-the-number-of-threads-in-polars

120 polars.thread_pool_size — Polars documentation
https://docs.pola.rs/py-polars/html/reference/api/polars.thread_pool_size.html

132 133 134 135 Multiprocessing - Polars user guide
https://docs.pola.rs/user-guide/misc/multiprocessing/