Rust in Data Science! 🦀

An Outlook of Data Transformation Using Polars

Luciano Scarpulla

Intro

- Hi there! I'm Luciano (lu-cha-no)
- I'm passionate about data and programming
- I love learning new things and sharing knowledge with others
- Currently working for a consultancy company in aviation
- Actively working in open source and exploring Rust to enhance efficiency

Code for this presenetation is available on GitHub

The state of Data science in Rust

"Are we data science yet?"

arrow [crate · repo · docs]

crates.io v54.0.0 | downloads 14M | recent downloads 3.1M | Star 2.7k

Rust implementation of Apache Arrow

Last Commit: 2025-01-19 **Last Published**: 2025-01-18

---

ndarray [crate · repo · docs]

crates.io v0.16.1 | downloads 19M | recent downloads 2.8M | Star 3.7k

An n-dimensional array for general elements and for numerics. Lightweight array views and slicing; views support chunking and splitting.

Last Commit: 2024-12-20 **Last Published**: 2024-08-14

---

polars [crate · repo · docs]

crates.io v0.45.1 | downloads 2.3M | recent downloads 393k | Star 31k

DataFrame library based on Apache Arrow

Last Commit: 2025-01-19 **Last Published**: 2024-12-09

---

sprs [crate · repo · docs]

crates.io v0.11.3 | downloads 1.9M | recent downloads 204k | Star 440

A sparse matrix library

https://www.arewelearningyet.com/data-structures/

The state of Data science in Rust

- Currently there are not many crates for data science in Rust
- Arrow - The universal columnar format and multi-language toolbox for fast data interchange and in-memory analytics
- **Polars** - Dataframes powered by a multithreaded, vectorized query engine, written in Rust

Some exciting features about Polars

Polars is one of the fastest data science tools that is written in Rust with its own compute and buffer implementations, while maintaining compatibility with Apache Arrow. It is more memory efficient than Pandas.

- Native columnar data storage
- Lazy API
- Multi-threaded out of the box
- Cheap Copy-on-Write
- Query plan optimization
  - Predicate pushdown
  - Projection pushdown

Dataframes and Column oriented storage

A Dataframe in polars is a collection of Series (columns) of equal length. Each column has a name and a single data type.

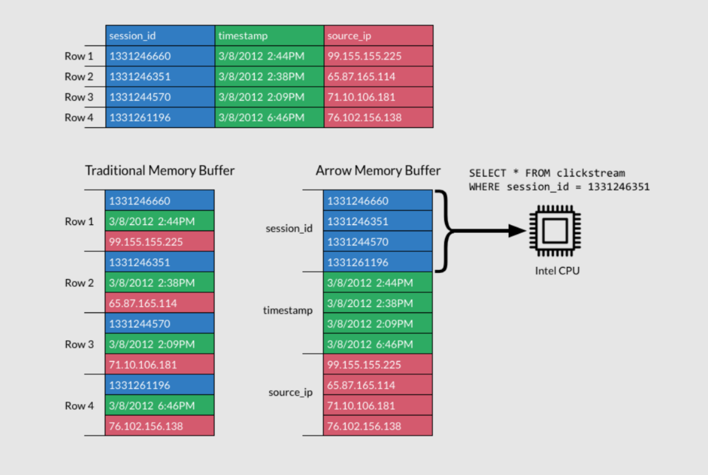
timestamp --- date	id --- i64	name --- str	age --- i32
2023-01-01	1	John Doe	25
2023-01-02	2	Jane Doe	30
2023-01-03	3	Bob Smith	45

Column oriented storage

Columnar storage is efficient because:

- Better compression - Similar values stored together compress better
- Cache efficiency - Accessing columns keeps data local in memory
- Query optimization - Only reading needed columns reduces I/O
- SIMD operations - Vector operations can process entire columns at once
- Memory efficiency - No need to read unused columns

Column oriented storage



Lazy vs Eager

- **Eager Evaluation**

```
  ───▶ Load Data ───▶ Filter ───▶ GroupBy ───▶
Sort ───▶ Result
```

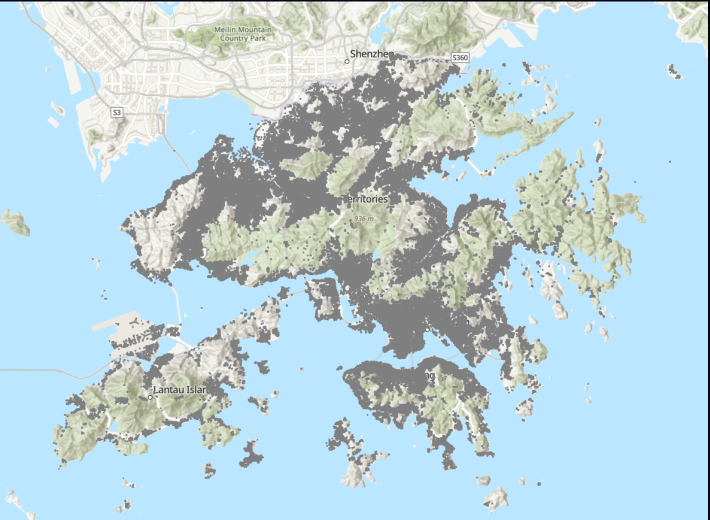
Executed immediately step by step

- **Lazy Evaluation**

```
  Load Data ─┐
  Filter    ─┤
  GroupBy   ─┼───▶ Optimize ───▶ Execute ───▶
Result
  Sort      ─┘
```

Plan built first, then optimized & executed

Show time!: Buildings in Hong Kong



```
{
"type": "FeatureCollection",
"name": "Buildings_in_Hong_Kong",
"crs": { "type": "name", "properties": { "name
": "urn:ogc:def:crs:OGC:1.3:CRS84" } },
"features": [
{ "type": "Feature",
  "properties": {
    "OBJECTID": 1, "BUILDINGSTRUCTUREID":
5243561, "BUILDINGCSUID": "0162608928T20071224
", "BUILDINGSTRUCTURETYPE": "T", "CATEGORY": "5
", "STATUS": "A", "STATUSDATE": null, "
OFFICIALBUILDINGNAMEEN": null, "
OFFICIALBUILDINGNAMETC": null, "
NUMABOVEGROUNDSTOREYS": null, "
NUMBASEMENTSTOREYS": null, "TOPHEIGHT": null, "
BASEHEIGHT": null, "GROSSFLOORAREA": null, "
RECORDCREATIONDATE": "2007-12-24T00:00:00Z", "
RECORDUPDATEDDATE": "2008-01-16T00:00:00Z", "
SHAPE__Area": 16.947265625, "SHAPE__Length":
18.893572763678002 },
  "geometry": { "type": "Polygon", "coordinates
": ... } },
...]
```

This file has 400K features.

Eager API

```rust
use polars::prelude::*;
use polars_demo::{load_data, unnest_df};

fn main() -> Result<(), Box<dyn
std::error::Error>> {
    let path = std::env::temp_dir().join("
hk_buildings.json");
    load_data(&path)?;
    let file = std::fs::File::open(path)?;
    let mut df = JsonReader::new(file).finish
()?.select(["features"])?;
    df = unnest_df(&df)?;
    println!("{:?}", df);
    println!("{:?}", df.column("
RECORDCREATIONDATE")?);
}
```

Eager API

Let's find out how many building records were created over the years!

Eager API

We can create or modify existing columns!

```
df.with_column(
    df.column("RECORDCREATIONDATE")?
        .str()?
        .as_datetime(
            Some("%FT%H:%M:%SZ"),
            TimeUnit::Nanoseconds,
            true,
            false,
            None,
            &ambiguous,
        )?
        .year()
        .with_name("creation_year".into()),
)?;

println!("{:?}", df.column("creation_year")?);
```

Eager API

```
println!(
    "{:?}",
    df.group_by(["creation_year"])?
        .select(["OBJECTID"])
        .count()?
        .sort(
            ["OBJECTID_count"],
            SortMultipleOptions::default().
with_order_descending(true)
        )
);
```

Eager API

creation_year --- i32	OBJECTID_count --- u32
2005	204182
2008	74603
2007	27891
2009	10094
2006	10088
…	…
2015	6010
2016	4900
2020	4804
2004	1567
2022	1164

How can we make this faster (Lazy API)

The Lazy API in Polars can provide significant performance improvements over eager evaluation by:

- Building a query plan first instead of executing operations immediately
- Allowing the query optimizer to analyze and improve the entire plan
- Pushing predicates down to minimize data read
- Minimizing intermediate allocations
- Parallelizing operations when possible
- Only loading required columns from disk
- Combining multiple operations into optimized steps

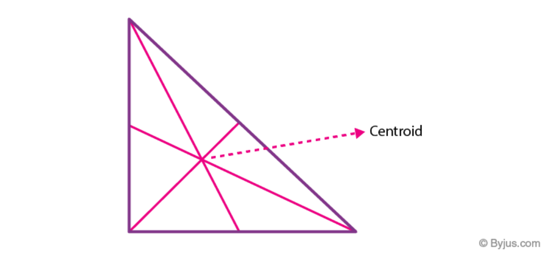
How can we make this faster (Lazy API)

```
lf = lf.with_column(
    col("RECORDCREATIONDATE")
        .str()
        .strptime(

DataType::Datetime(TimeUnit::Milliseconds,
None),
            StrptimeOptions {
                format: Some("%FT%H:%M:%SZ".
into()),
                strict: false,
                exact: true,
                cache: false,
            },
            lit("raise"),
        )
        .dt()
        .year()
        .alias("creation_year"),
)
.group_by(["creation_year"])
.agg([col("OBJECTID").count()])
.sort(
```


processing step-by-step. plan before executing, rather than

Centroid



We can add native Rust function to our Polars code!

```rust
use geo::{Centroid, Polygon};

fn calculate_centroid(coords: Vec<Vec<f64>>) ->
Option<(f64, f64)> {
    let polygon = Polygon::new(
        geo::LineString::from(
            coords
                .into_iter()
                .map(|coord| (coord[0], coord[1
]))
                .collect::<Vec<_>>(),
        ),
        vec![],
    );

    polygon
        .centroid()
        .map(|centroid| (centroid.x(),
centroid.y()))
}
```

Why Polars in Rust for Production?

- **Performance & Resources**
  - Blazing fast query execution
  - Memory efficient with zero-copy operations
  - Native multi-threading support
  - Perfect for resource-constrained environments
- **Production Ready**
  - Strong type system prevents runtime errors
  - Excellent error handling with Result type
  - Cross-platform compatibility
- **Backend Integration**

```rust
// Example with Actix-web
use actix_web::{get, web, App, HttpServer,
Result};
use polars::prelude::*;

#[get("/buildings/stats")]
async fn building_stats() ->
Result<web::Json<Value>> {
    let lf = LazyFrame::scan_parquet(
        "data.parquet",
        ScanArgsParquet::default()
    )?
```

Why Polars in Rust for Production?

- **Versatile Data Pipeline Building**
  - Read/Write multiple formats (Parquet, CSV, JSON)
  - Easy integration with Arrow ecosystem
  - Stream processing capabilities
  - Excellent for ETL workflows
- **Developer Experience**
  - Familiar DataFrame API
  - Great documentation and growing community
  - IDE support with type hints
  - Easy to test and benchmark
- **Real-world Use Cases**
  - Data APIs and Microservices
  - Real-time analytics
  - ETL Pipelines
  - Machine Learning Feature Engineering
  - IoT Data Processing

Summary 🚀

- We explored Polars - a powerful DataFrame library for Rust
- Learned about key features:
  - Column-oriented storage
  - Lazy evaluation
  - Native multi-threading
  - Production-ready capabilities
- Demonstrated real-world examples
  - Data loading and transformation
  - Geographic calculations
  - API integration
- The Rust data science ecosystem is growing!