

DuckDB & Polars

Fast single-machine analytics — SQL and DataFrame engines that handle data too large for pandas and too small to need a cluster.

Last reviewed 29 Sep 2026 · Lab-tested with DuckDB 1.5.5 ·

Prerequisites: [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) · [Python for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/)

Related: [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) · [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) · [Apache Iceberg](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) · [Cost Optimization](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Practice: [Lab 01 — SQL Analytics](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/01-sql-analytics) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/01-sql-analytics)

Overview

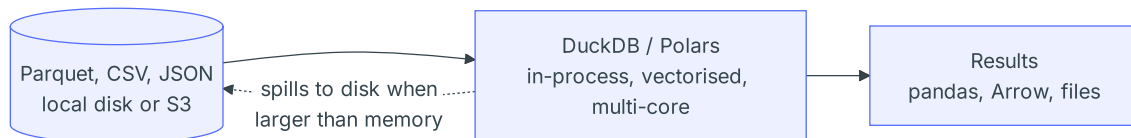
Challenge: Many datasets are too large or slow for pandas — which is single-threaded and loads everything into memory — yet far too small to justify a Spark cluster's cost and operational overhead. Tens or hundreds of gigabytes of Parquet on a laptop or a single cloud VM falls into this gap.

Solution: Two modern engines fill it. **DuckDB** is an in-process analytical SQL database (think "SQLite for analytics"); **Polars** is a DataFrame library with a lazy query optimizer. Both are columnar, multi-threaded, vectorized, read Parquet and object storage directly, and can process data larger than memory.

Data size	Typical tool
< 1 GB	pandas, DuckDB, Polars — any works

1 GB – ~1 TB DuckDB or Polars on one machine (spill to disk when needed)
> 1 TB, or many concurrent users Spark, a warehouse, or a lakehouse engine

Relevance to data engineering: these engines make local development, CI tests, small and medium pipelines, and data exploration dramatically faster and cheaper — often replacing a cluster job with a single container.



On this page

Basic - [When to Use Which](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#when-to-use-which) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#when-to-use-which) - [DuckDB Basics](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#duckdb-basics) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#duckdb-basics) - [Polars Basics](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#polars-basics) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#polars-basics)

Intermediate - [Reading and Writing Files](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#reading-and-writing-files) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#reading-and-writing-files) - [Object Storage and Lakehouse Formats](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#object-storage-and-lakehouse-formats) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#object-storage-and-lakehouse-formats) - [Interoperability](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#interoperability) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#interoperability)

Advanced - [Larger-than-Memory Processing](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#larger-than-memory-processing) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#larger-than-memory-processing) - [Performance Tips](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#performance-tips) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#performance-tips) - [Pipelines and Testing Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#pipelines-and-testing-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#pipelines-and-testing-patterns)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/#further-reading)

When to Use Which

	pandas	Polars	DuckDB	Spark
Interface	DataFrame	DataFrame (eager + lazy)	SQL (plus relational Python API)	DataFrame + SQL
Execution	Single- threaded, eager	Multi-threaded, query-optimized	Multi-threaded, query-optimized	Distributed cluster
Larger than memory	No	Yes (streaming engine)	Yes (spills to disk)	Yes
Best at	Small data, ecosystem breadth	DataFrame pipelines, feature engineering	SQL analytics, joins, ad hoc queries over files	Terabytes+, cluster-scale jobs
Deployment	Library	Library	Library (embedded database)	Cluster or managed service

Rule of thumb: use DuckDB if your team thinks in SQL, Polars if it thinks in DataFrames — and they interoperate, so mixing them is easy.

DuckDB Basics

```
import duckdb

# Query files directly – no loading step
duckdb.sql("""
    SELECT region, SUM(amount) AS revenue
    FROM 'data/orders/*.parquet'
    WHERE order_date >= DATE '2024-03-01'
    GROUP BY region
    ORDER BY revenue DESC
""").show()

# Persistent database file (tables survive restarts)
con = duckdb.connect("analytics.duckdb")
```

```

con.sql("CREATE TABLE IF NOT EXISTS orders AS SELECT * FROM
'data/orders/*.parquet'")

# Results as pandas, Polars, or Arrow
df_pandas = con.sql("SELECT * FROM orders LIMIT 1000").df()
df_polars = con.sql("SELECT * FROM orders LIMIT 1000").pl()
table      = con.sql("SELECT * FROM orders LIMIT
1000").arrow()

# Parameterized queries
con.execute("SELECT * FROM orders WHERE customer_id = ?",
[42]).fetchall()

```

Handy SQL features: `SUMMARIZE tbl` (profile every column) · `DESCRIBE` · `SELECT * EXCLUDE (col)` · `GROUP BY ALL` · `QUALIFY` · `PIVOT / UNPIVOT` · list and struct types · `read_json` with schema inference

Polars Basics

```

import polars as pl

orders = pl.read_parquet("data/orders/*.parquet") # eager

result = (
    orders
    .filter(pl.col("status") == "shipped")
    .with_columns(
        (pl.col("amount") *
pl.col("fx_rate")).alias("amount_usd"),
        pl.col("order_ts").dt.date().alias("order_date"),
    )
    .group_by("region", "order_date")
    .agg(
        pl.col("amount_usd").sum().alias("revenue"),
        pl.col("order_id").n_unique().alias("orders"),
    )
    .sort("revenue", descending=True)
)

```

```
# Conditional logic and window functions
orders = orders.with_columns(
    pl.when(pl.col("amount") >
1000).then(pl.lit("large")).otherwise(pl.lit("standard")).a
lias("size"),

    pl.col("amount").sum().over("customer_id").alias("customer_
total"),

    pl.col("order_ts").rank("ordinal").over("customer_id").alia
s("order_seq"),
)

# Joins
enriched = orders.join(customers, on="customer_id",
how="left")
```

Lazy mode (`scan_*` + `.collect()`) lets Polars optimize the whole query — pushing filters and column selection into the file scan:

```
revenue = (
    pl.scan_parquet("data/orders/*.parquet")
nothing is read yet
    .filter(pl.col("order_date") >= pl.date(2024, 3, 1))
    .group_by("region")
    .agg(pl.col("amount").sum())
    .collect()
optimize, then execute
)
print(pl.scan_parquet("data/orders/*.parquet").filter(pl.co
l("region") == "EU").explain())
```

Reading and Writing Files

```
-- DuckDB
SELECT * FROM read_csv('raw/customers*.csv', header =
true);
SELECT * FROM read_json('raw/events/*.json.gz');
SELECT * FROM read_parquet('lake/orders/*.parquet',
hive_partitioning = true);
```

```
-- Write partitioned Parquet (for local paths, the parent
directory must already exist)
COPY (SELECT *, CAST(order_ts AS DATE) AS order_date FROM
orders)
TO 'lake/orders' (FORMAT parquet, PARTITION_BY
(order_date), COMPRESSION zstd);
```

```
# Polars
df = pl.read_csv("raw/customers.csv", try_parse_dates=True)
lf = pl.scan_parquet("lake/orders/**/*.parquet",
hive_partitioning=True)
df.write_parquet("out/customers.parquet",
compression="zstd")
lf.sink_parquet("out/orders_clean.parquet") # write
without materializing in memory
```

Object Storage and Lakehouse Formats

```
-- DuckDB: read directly from S3 / GCS / Azure
INSTALL httpfs; LOAD httpfs;
CREATE SECRET (TYPE s3, PROVIDER credential_chain); --
uses the standard AWS credential chain

SELECT COUNT(*) FROM 's3://my-lake/orders/order_date=2024-
03-15/*.parquet';

-- Open table formats
INSTALL iceberg; LOAD iceberg;
SELECT * FROM iceberg_scan('s3://my-
lake/warehouse/orders');

INSTALL delta; LOAD delta;
SELECT * FROM delta_scan('s3://my-lake/delta/orders');
```

```
# Polars: object storage via storage options (credentials
can also come from the environment)
lf = pl.scan_parquet(
```

```
"s3://my-lake/orders/order_date=2024-03-15/*.parquet",
storage_options={"aws_region": "eu-west-1"},
)
```

Filters on partition columns and Parquet statistics mean only the needed files and row groups are downloaded.

Interoperability

Both engines use **Apache Arrow** in memory, so data moves between them — and pandas — with little or no copying.

```
import duckdb, polars as pl, pandas as pd

pdf = pd.read_csv("small.csv")
plf = pl.DataFrame({"id": [1, 2], "score": [0.4, 0.9]})

# DuckDB can query pandas and Polars DataFrames by variable
name
duckdb.sql("SELECT p.*, s.score FROM pdf AS p JOIN plf AS s
USING (id)").pl()

# Polars ↔ pandas ↔ Arrow
plf.to_pandas(); pl.from_pandas(pdf); plf.to_arrow()

# Polars can run SQL too
pl.SQLContext(orders=plf).execute("SELECT id FROM orders
WHERE score > 0.5").collect()
```

Larger-than-Memory Processing

```
# DuckDB: cap memory and spill to disk for large joins,
sorts, and aggregations
con = duckdb.connect()
con.sql("SET memory_limit = '8GB'")
con.sql("SET temp_directory = '/mnt/scratch/duckdb_tmp'")
con.sql("""
    COPY (
```

```

        SELECT customer_id, SUM(amount) AS lifetime_value
        FROM 'lake/orders/**/*.*.parquet'
        GROUP BY customer_id
    ) TO 'out/ltv.parquet' (FORMAT parquet)
    """

# Polars: the streaming engine processes data in batches
(pl.scan_parquet("lake/orders/**/*.*.parquet")
 .group_by("customer_id")
 .agg(pl.col("amount").sum().alias("lifetime_value"))
 .sink_parquet("out/ltv.parquet")) # or
.collect(engine="streaming") in recent versions

```

Streaming works best for scans, filters, projections, and aggregations; some operations (certain window functions, sorts over everything) need more memory.

Performance Tips

- **Use Parquet**, not CSV, for anything read more than once; partition by common filters
- **Stay lazy** in Polars (`scan_*`) and let DuckDB read files directly — both push down filters and column selection
- **Avoid Python row loops and UDFs** (`apply` , `map_elements`) — use built-in expressions
- **Select only needed columns** early
- **Right-size files**: hundreds of MB per file, not thousands of tiny files
- **Set threads and memory** explicitly in containers (`SET threads = 4` ; Polars uses `POLARS_MAX_THREADS`) so the engine doesn't assume the host's full resources

Pipelines and Testing Patterns

A containerized batch job instead of a cluster

```

from datetime import date

import duckdb

def run(run_date: str) -> None:
    day = date.fromisoformat(run_date).isoformat()
    # validates the input before it touches SQL

```

```

output = f"s3://my-
lake/gold/orders_enriched/order_date={day}/data.parquet"

con = duckdb.connect()
con.sql("INSTALL httpfs; LOAD httpfs; CREATE SECRET
(TYPE s3, PROVIDER credential_chain);")
# Query inputs can be parameters; the COPY target must
be a literal path
con.execute(f"""
    COPY (
        SELECT o.order_id, o.customer_id, o.amount,
c.region, o.order_date
        FROM read_parquet('s3://my-
lake/silver/orders/order_date=' || $day || '/*.parquet') AS
o
        JOIN read_parquet('s3://my-
lake/silver/customers/*.parquet') AS c USING (customer_id)
    ) TO '{output}' (FORMAT parquet)
    """, {"day": day}) # overwrites the day's
output file: safe to rerun

```

Fast local and CI tests: run SQL transformation logic against small fixture files with DuckDB, so tests take seconds without a warehouse connection. Transformation frameworks such as dbt and SQLMesh have DuckDB adapters for exactly this.

Local development against production-shaped data: sample a few partitions from object storage into a local DuckDB file and iterate offline.

Common Pitfalls

Pitfall	Symptom	Fix
Treating DuckDB as a shared server database	Lock errors with multiple writers	One writer process per database file; use a server database or MotherDuck for concurrent access
Using Polars eagerly on big files	Out-of-memory errors	<code>scan_*</code> + lazy queries + <code>sink_*</code> or streaming collect
Row-wise Python functions	10–100× slower than expected	Native expressions or SQL functions

Pitfall	Symptom	Fix
Reading thousands of tiny files	Slow scans dominated by file overhead	Compact into larger Parquet files
CSV type inference surprises	IDs lose leading zeros; wrong date parsing	Declare column types explicitly for production reads
No memory limit in containers	The container is OOM-killed	<code>SET memory_limit</code> , a temp directory, <code>POLARS_MAX_THREADS</code>
Choosing a single-node engine for ever-growing data	Jobs slow down as volume grows	Plan the migration path (same SQL on a warehouse or Spark) before you hit the limit
Mixing API versions from old tutorials	Deprecation warnings or errors (e.g. <code>groupby</code> vs <code>group_by</code>)	Pin library versions and follow current docs

Cheat Sheet

Task	DuckDB	Polars
Read Parquet	<code>SELECT * FROM 'path/*.parquet'</code>	<code>pl.scan_parquet("path/*.parquet")</code>
Filter	<code>WHERE status = 'shipped'</code>	<code>.filter(pl.col("status") == "shipped")</code>
New column	<code>SELECT *, amount * 1.2 AS gross</code>	<code>.with_columns((pl.col("amount") * 1.2).alias("gross"))</code>
Aggregate	<code>GROUP BY region + SUM(amount)</code>	<code>.group_by("region").agg(pl.col("amount").sum())</code>
Window	<code>SUM(amount) OVER (PARTITION BY customer_id)</code>	<code>pl.col("amount").sum().over("customer_id")</code>
Latest per key	<code>QUALIFY ROW_NUMBER() OVER (PARTITION BY id ORDER BY ts DESC) = 1</code>	<code>.sort("ts", descending=True).unique("id", keep="first")</code>
Join	<code>JOIN c USING (customer_id)</code>	<code>.join(c, on="customer_id", how="left")</code>

Task	DuckDB	Polars
Write Parquet	<code>COPY (...) TO 'out.parquet' (FORMAT parquet)</code>	<code>.sink_parquet("out.parquet") / .write_parquet(...)</code>
Profile data	<code>SUMMARIZE tbl</code>	<code>df.describe()</code>
To pandas	<code>.df()</code>	<code>.to_pandas()</code>

Interview Questions

Q: Why are DuckDB and Polars so much faster than pandas? A: They use columnar, vectorized execution — processing batches of values with CPU-efficient operations — across all cores, while pandas executes most operations on a single thread. They also have query optimizers: filters and column selection are pushed into file scans, joins are planned, and unnecessary work is removed. Finally, they can stream or spill to disk, so they aren't limited by memory the way pandas is.

Q: When would you choose DuckDB or Polars over Spark? A: When the data fits on one machine — up to hundreds of gigabytes, sometimes more with spilling — and the job doesn't need a cluster's parallelism. A single process avoids cluster startup time, scheduling overhead, and cost, and is simpler to test and deploy (a container in an orchestrator). Spark remains the right choice for terabyte-scale data, very large shuffles, or when the platform is already built around it.

Q: What is lazy evaluation in Polars and why does it matter? A: With `scan_*` functions and `LazyFrame`, operations build a query plan instead of executing immediately. On `collect()` or `sink_*`, Polars optimizes the whole plan — predicate and projection pushdown into the file scan, combining operations, choosing join strategies — and can execute it in a streaming fashion. That reads less data, uses less memory, and runs faster than executing each step eagerly.

Q: How can DuckDB help in a data engineering workflow even if production runs on a warehouse? A: It's an excellent local and CI engine: SQL logic can be tested against small fixture files in seconds, without warehouse credentials or cost. It's also useful for inspecting and profiling files in object storage, validating data before loading, and running small scheduled jobs in a container instead of on a warehouse or cluster.

Further Reading

- [DuckDB documentation](https://duckdb.org/docs/) (https://duckdb.org/docs/)

- [Polars user guide](https://docs.pola.rs/) (https://docs.pola.rs/)
 - [DuckDB: friendlier SQL](https://duckdb.org/docs/sql/dialect/friendly_sql) (https://duckdb.org/docs/sql/dialect/friendly_sql) — extensions to standard SQL
 - [Apache Arrow](https://arrow.apache.org/docs/) (https://arrow.apache.org/docs/) — the shared in-memory format
 - [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) — for data beyond a single machine
-

Previous: [Prefect](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/prefect-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/prefect-reference/) • **Next:** [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026