DuckDB

DuckDB runs SQL right on your DataFrames and files with no server, here from a first query to joins, exports, and EXPLAIN.

Connect & Query

One import, one connection, SQL straight away.

PURPOSE	COMMAND	RESULT
Run SQL with zero setup.	`duckdb.sql("SELECT 42 AS answer")`	
Open an in-memory connection.	`con = duckdb.connect()`	
Open or create a file database.	`con = duckdb.connect("my.duckdb")`	
Run SQL on that connection.	`con.sql("SELECT * FROM range(3)")`	
Execute a statement (DDL, params).	`con.execute("INSERT INTO t VALUES (?)", [1])`	
Close when done.	`con.close()`	

import duckdb, then duckdb.sql(...) runs against a default in-memory db; connect() for your own, connect("f.duckdb") to persist

Query Files Directly

Point SQL at Parquet, CSV, or a glob of files with no load step.

PURPOSE	COMMAND	RESULT
Read one Parquet file in SQL.	`duckdb.sql("SELECT * FROM 'sales.parquet'")`	
Read a folder of files (glob).	`FROM 'data/*.parquet'`	
CSV with auto type detection.	`FROM read_csv_auto('data.csv')`	
Read CSV via the Python helper.	`duckdb.read_csv("data.csv")`	
Read CSV from a URL.	`FROM 'https://…/d.csv'`	
Peek at the first rows.	`FROM 'sales.parquet' LIMIT 5`	

a path inside the SQL reads the file directly; a glob auto-unions a folder, read_csv_auto sniffs types, https:// streams remotely

SQL on a DataFrame

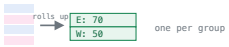
Query a pandas or Polars DataFrame by its Python variable name, no copy.

PURPOSE	COMMAND	RESULT
Query a pandas frame by name.	`duckdb.sql("SELECT * FROM sales_df")`	
Query a Polars frame by name.	`duckdb.sql("SELECT region FROM pl_df")`	
Aggregate the frame in SQL.	`SELECT region, sum(amt) FROM sales_df GROUP BY region`	
Register a frame under a name.	`con.register("orders", sales_df)`	
Send the result back to pandas.	`duckdb.sql("SELECT * FROM sales_df").df()`	
Mix a frame and a file.	`SELECT * FROM sales_df JOIN 'ref.parquet' USING (id)`	

replacement scans put a DataFrame's Python variable name straight in the FROM clause; .df() / .pl() hand the answer back

Filter, Aggregate, Window

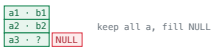
WHERE filters rows, GROUP BY rolls up, OVER computes without collapsing.

PURPOSE	COMMAND	RESULT
Filter rows with WHERE.	`SELECT * FROM t WHERE amt > 100`	
Group and aggregate.	`region, sum(amt) AS total ... GROUP BY region`	
Order and limit (top N).	`ORDER BY amt DESC LIMIT 10`	
Running total with a window.	`sum(amt) OVER (ORDER BY ts) AS running`	
Partitioned window (per group).	`rank() OVER (PARTITION BY region ORDER BY amt DESC)`	
Condense the whole pipeline.	`WHERE amt > 0 GROUP BY region HAVING avg > 50`	

GROUP BY collapses many rows into one per group; a window OVER (PARTITION BY ... ORDER BY ...) computes without collapsing

Joins

INNER keeps matches, LEFT keeps all left rows, ASOF matches the nearest earlier time.

PURPOSE	COMMAND	RESULT
Inner join on a key.	`FROM a JOIN b ON a.id = b.id`	
Left join (keep all left rows).	`FROM a LEFT JOIN b ON a.id = b.id`	
Join on a shared column name.	`FROM a JOIN b USING (id)`	
ASOF join (nearest earlier row).	`ASOF JOIN prices p ON t.sym=p.sym AND t.ts>=p.ts`	
Stack rows from two sources.	`SELECT * FROM a UNION ALL SELECT * FROM b`	
Anti join (rows with no match).	`WHERE id NOT IN (SELECT id FROM b)`	

ASOF JOIN matches each left row to the nearest earlier right row (t.ts >= p.ts), the standard tool for prices-per-trade

Move Results Out

A Relation is lazy. One fetch method turns it into pandas, Polars, Arrow, or Python.

PURPOSE	COMMAND	RESULT
To a pandas DataFrame.	`duckdb.sql("...").df()`	
To a Polars DataFrame.	`duckdb.sql("...").pl()`	
To an Arrow table.	`duckdb.sql("...").arrow()`	
To plain Python objects.	`duckdb.sql("...").fetchall()`	
To NumPy column arrays.	`duckdb.sql("...").fetchnumpy()`	
Lazy until you fetch.	`rel = duckdb.sql("SELECT ...")`	

nothing runs until you fetch or print; .df() / .pl() / .arrow() / .fetchall() / .fetchnumpy() pick the output format you want

Persist & Export

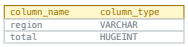
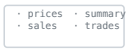
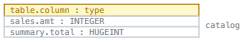
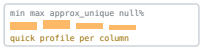
Materialize a query into a stored table, or write it straight to a file with COPY.

PURPOSE	COMMAND	RESULT
Create a table from a query.	`CREATE TABLE summary AS SELECT ... GROUP BY region`	
Replace an existing table.	`CREATE OR REPLACE TABLE summary AS ...`	
Append more rows.	`INSERT INTO summary SELECT * FROM more`	
Write a query to Parquet.	`COPY (SELECT * FROM t) TO 'out.parquet'`	
Write to CSV with a header.	`COPY t TO 'out.csv' (HEADER, DELIMITER ',')`	
Write via the Relation helper.	`duckdb.sql("...").write_parquet("out.parquet")`	

CREATE [OR REPLACE] TABLE ... AS SELECT freezes a query into a stored table; COPY (q) TO 'file' writes straight to disk

Inspect & EXPLAIN

See the schema, list what exists, and read how a query will run.

PURPOSE	COMMAND	RESULT
Describe a table or query.	`DESCRIBE summary`	
List tables in the database.	`SHOW TABLES`	
Show all columns across tables.	`DESCRIBE or duckdb_columns()`	
Summarize a dataset's stats.	`SUMMARIZE t`	
Read the query plan.	`EXPLAIN SELECT region, sum(amt) ... GROUP BY`	
Read the plan with real timings.	`EXPLAIN ANALYZE SELECT ...`	

DESCRIBE / SHOW TABLES / SUMMARIZE inspect schema and stats; EXPLAIN prints the operator tree, EXPLAIN ANALYZE adds timings