DuckDB & SQL

Lecture 21

Dr. Colin Rundel

SQL

Structured Query Language is a special purpose language for interacting with (querying and modifying) indexed tabular data.

- ANSI Standard but with dialect divergence (MySql, Postgres, SQLite, etc.)
- This functionality maps very closely (but not exactly) with the data manipulation verbs present in dplyr.
- SQL is likely to be a foundational skill if you go into industry - learn it and put it on your CV

DuckDB

> DuckDB is an open-source column-oriented relational database management system (RDBMS) originally developed by Mark Raasveldt and Hannes Mühleisen at the Centrum Wiskunde & Informatica (CWI) in the Netherlands and first released in 2019. The project has over 6 million downloads per month. It is designed to provide high performance on complex queries against large databases in embedded configuration, such as combining tables with hundreds of columns and billions of rows. Unlike other embedded databases (for example, SQLite) DuckDB is not focusing on transactional (OLTP) applications and instead is specialized for **online analytical processing (OLAP)** workloads.

From Wikipedia - DuckDB

OLTP vs OLAP / In-Process vs Server

DuckDB & DBI

DuckDB is a relational database just like SQLite and can be interacted with using DBI and the duckdb package.

```
1 library(DBI)
2 (con = dbConnect(duckdb::duckdb()))
```

```
<duckdb_connection 61b20 driver=<duckdb_driver dbdir=':memory:' read_only=FALSE bigint
```

```
1 dbWriteTable(con, "flights", nycflights13::flights)
2 dbListTables(con)
```

```
[1] "flights"
```

```r
dbGetQuery(con, "SELECT * FROM flights") |>
  as_tibble()
```

```
# A tibble: 336,776 × 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ℹ 336,766 more rows
# ℹ 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

```r
library(dplyr)
tbl(con, "flights") |>
  filter(month == 10, day == 29) |>
  count(origin, dest) |>
  arrange(desc(n))
```

```
# Source:     SQL [?? x 3]
# Database:   DuckDB 1.5.0 [root@Darwin 25.4.0:R 4.5.3/:memory:]
# Ordered by: desc(n)
   origin dest      n
   <chr>  <chr> <dbl>
 1 JFK    LAX      32
 2 LGA    ORD      30
 3 LGA    ATL      29
 4 JFK    SFO      23
 5 LGA    CLT      21
 6 EWR    ORD      18
 7 JFK    BOS      16
 8 EWR    SFO      16
 9 LGA    BOS      16
10 LGA    DCA      16
# ℹ more rows
```

DuckDB CLI

Connecting via CLI

```
1 > duckdb employees.duckdb
```

```
DuckDB v1.5.1 (Variegata)
Enter ".help" for usage hints.
employees D
```

Once connected, you can run SQL queries directly in the terminal. For example,

```
1 SELECT 'Hello, DuckDB!' AS greeting;
```

```
┌──────────────────┐
│     greeting     │
│     varchar      │
├──────────────────┤
│ Hello, DuckDB!   │
└──────────────────┘
```

Table information

Dot commands are expressions that begins with `.` and are specific to the DuckDB CLI, some examples include:

```
1 .tables
```

```
── employees ──
──── main ────
```

```
┌──────────────┐
│  employees   │
│              │
│ name   varchar │
│ email  varchar │
│ salary double  │
│ dept   varchar │
│              │
│    6 rows    │
└──────────────┘
```

```
1 .schema employees
```

```
CREATE TABLE employees("name" VARCHAR, email VARCHAR, sala
```

```
1 .version
```

```
DuckDB v1.5.1 (Variegata) 7dbb2e646f
clang-17.0.0
```

```
1 .print TESTING 1 2 3 ...
```

```
TESTING 1 2 3 ...
```

A full list of available dot commands can be found here or listed via `.help` in the CLI.

Basic SQL query

```
1 SELECT * FROM employees;
```

name varchar	email varchar	salary double	dept varchar
Alice	alice@company.com	52000.0	Accounting
Bob	bob@company.com	40000.0	Accounting
Carol	carol@company.com	30000.0	Sales
Dave	dave@company.com	33000.0	Accounting
Eve	eve@company.com	44000.0	Sales
Frank	frank@comany.com	37000.0	Sales

Output formats

The format of duckdb's output (in the CLI) is controlled via `.mode` - the default is duckbox, see possible output formats.

```
.mode csv
SELECT * FROM employees;
```

```
name,email,salary,dept
Alice,alice@company.com,52000.0,Accounting
Bob,bob@company.com,40000.0,Accounting
Carol,carol@company.com,30000.0,Sales
Dave,dave@company.com,33000.0,Accounting
Eve,eve@company.com,44000.0,Sales
Frank,frank@comany.com,37000.0,Sales
```

```
.mode markdown
SELECT * FROM employees;
```

```
| name  |       email       | salary  |    dept    |
|-------|-------------------|--------:|------------|
| Alice | alice@company.com | 52000.0 | Accounting |
| Bob   | bob@company.com   | 40000.0 | Accounting |
| Carol | carol@company.com | 30000.0 | Sales      |
| Dave  | dave@company.com  | 33000.0 | Accounting |
| Eve   | eve@company.com   | 44000.0 | Sales      |
| Frank | frank@comany.com  | 37000.0 | Sales      |
```

```
.mode json
SELECT * FROM employees;
```

```
[{"name":"Alice","email":"alice@company.com","salary":5
{"name":"Bob","email":"bob@company.com","salary":40000.
{"name":"Carol","email":"carol@company.com","salary":30
{"name":"Dave","email":"dave@company.com","salary":3300
{"name":"Eve","email":"eve@company.com","salary":44000.
{"name":"Frank","email":"frank@comany.com","salary":370
```

```
.mode insert
SELECT * FROM employees;
```

```
INSERT INTO "table"("name",email,salary,dept) VALUES('A
INSERT INTO "table"("name",email,salary,dept) VALUES('B
INSERT INTO "table"("name",email,salary,dept) VALUES('C
INSERT INTO "table"("name",email,salary,dept) VALUES('D
INSERT INTO "table"("name",email,salary,dept) VALUES('E
INSERT INTO "table"("name",email,salary,dept) VALUES('F
```

A brief tour of SQL

Basic Keywords

We can subset and rename columns, sort results, and limit output using `SELECT`, `ORDER BY`, and `LIMIT`. Additionally, `DISTINCT` can be used to remove duplicate values.

```sql
SELECT name AS first_name, salary
FROM employees
ORDER BY salary DESC
LIMIT 3;
```

first_name varchar	salary double
Alice	52000.0
Eve	44000.0
Bob	40000.0

```sql
SELECT DISTINCT dept
FROM employees;
```

dept varchar
Accounting
Sales

SQL clauses are evaluated in a different order than they are written: `FROM` → `WHERE` → `GROUP BY` → `SELECT` → `ORDER BY` →

filter() using WHERE

We can filter rows using a `WHERE` clause

```
1 SELECT * FROM employees WHERE salary < 40000;
```

name varchar	email varchar	salary double	dept varchar
Carol	carol@company.com	30000.0	Sales
Dave	dave@company.com	33000.0	Accounting
Frank	frank@comany.com	37000.0	Sales

```
1 SELECT * FROM employees WHERE salary < 40000 AND dept = 'Sales';
```

name varchar	email varchar	salary double	dept varchar
Carol	carol@company.com	30000.0	Sales
Frank	frank@comany.com	37000.0	Sales

GROUP BY and HAVING

`GROUP BY` groups rows for aggregation; `HAVING` filters *after* aggregation (like `WHERE` but for grouped results).

```
1 SELECT dept, COUNT(*) AS n, ROUND(AVG(salary),2) AS avg_salary
2 FROM employees GROUP BY dept;
```

dept varchar	n int64	avg_salary double
Accounting	3	41666.67
Sales	3	37000.0

```
1 SELECT dept, COUNT(*) AS n, ROUND(AVG(salary),2) AS avg_salary
2 FROM employees GROUP BY dept
3 HAVING avg_salary > 38000;
```

dept varchar	n int64	avg_salary double
Accounting	3	41666.67

Exercise 1

Using DuckDB calculate the following quantities for `employees.duckdb`,

1. The total costs in payroll for this company
2. The number of employees in each department who earn more than $35,000

Reading from CSV files

DuckDB has a neat trick in that it can treat files as tables (for supported formats), this lets you query them without having to explicitly create a table in the database.

We can also make this explicit by using the `read_csv()` function, which is useful if we need to use custom options (e.g. specify a different delimiter)

```
1 SELECT * FROM 'Lec21/phone.csv';
```

name varchar	phone varchar
Bob	919 555-1111
Carol	919 555-2222
Eve	919 555-3333
Frank	919 555-4444

```
1 SELECT * FROM
2   read_csv('Lec21/phone.csv', delim = ',');
```

name varchar	phone varchar
Bob	919 555-1111
Carol	919 555-2222
Eve	919 555-3333
Frank	919 555-4444

Tables from CSV

If we wanted to explicitly create a table from the CSV file this is also possible,

```
1  .tables
```

```
── employees ──
─────── main ───────
```

```
┌─────────────┐
│  employees  │
│             │
│ name   varchar │
│ email  varchar │
│ salary double  │
│ dept   varchar │
│             │
│   6 rows    │
└─────────────┘
```

```
1  CREATE TABLE phone AS
2    SELECT * FROM 'Lec21/phone.csv';
3  .tables
```

```
──────────── employees ────────────
──────────── main ─────────────
```

```
┌─────────────┐  ┌─────────────┐
│  employees  │  │    phone    │
│             │  │             │
│ name   varchar │  │ name  varchar │
│ email  varchar │  │ phone varchar │
│ salary double  │  │             │
│ dept   varchar │  │   4 rows    │
│             │  └─────────────┘
│   6 rows    │
└─────────────┘
```

```
1  SELECT * FROM phone;
```

name varchar	phone varchar
Bob	919 555-1111
Carol	919 555-2222
Eve	919 555-3333
Frank	919 555-4444

Views from CSV

It is also possible to create a view from a file - this acts like a table but the data is not copied from the file. This means the file must remain accessible and is re-read on every query, but no storage is used within the database.

```
1 CREATE VIEW phone_view AS
2   SELECT * FROM 'Lec21/phone.csv';
```

```
1 .tables
```

```
──────────────────── employees ────────────────────
──────────────────────── main ────────────────────────
┌──────────────────┐ ┌──────────────────┐ ┌──────────────────┐
│    employees     │ │      phone       │ │    phone_view    │
│                  │ │                  │ │                  │
│ name   varchar   │ │ name  varchar    │ │ name  varchar    │
│ email  varchar   │ │ phone varchar    │ │ phone varchar    │
│ salary double    │ │                  │ └──────────────────┘
│ dept   varchar   │ │     4 rows       │
│                  │ └──────────────────┘
│      6 rows      │
└──────────────────┘
```

```
1 SELECT * FROM phone_view;
```

name varchar	phone varchar
Bob	919 555-1111
Carol	919 555-2222
Eve	919 555-3333
Frank	919 555-4444

Deleting tables and views

Tables and views can be deleted using DROP

```
1 .tables
```

```
──────────────────── employees ────────────────────
──────────────────────── main ─────────────────────
```

employees	phone	phone_view
name varchar	name varchar	name varchar
email varchar	phone varchar	phone varchar
salary double	4 rows	
dept varchar		
6 rows		

```
1 DROP TABLE phone;
2 DROP VIEW phone_view;
```

```
1 .tables
```

```
── employees ──
──── main ────
```

employees
name varchar
email varchar
salary double
dept varchar
6 rows

Inner Join

DuckDB requires an `ON` or `USING` clause for joins - `JOIN` alone is a syntax error.

```
1 SELECT * FROM employees JOIN phone USING(name);
```

name varchar	email varchar	salary double	dept varchar	phone varchar
Bob	bob@company.com	40000.0	Accounting	919 555-1111
Carol	carol@company.com	30000.0	Sales	919 555-2222
Eve	eve@company.com	44000.0	Sales	919 555-3333
Frank	frank@comany.com	37000.0	Sales	919 555-4444

`USING(col)` joins on a named column (no duplicate column in result); `ON t1.colA = t2.colB` is needed when column names differ. `NATURAL JOIN` automatically matches all common column names - convenient but can produce

Other Joins

All standard joins are supported: `LEFT JOIN`, `RIGHT JOIN`, `FULL JOIN`, `CROSS JOIN`, `SEMI JOIN`, `ANTI JOIN`, etc.

```
1 SELECT * FROM employees LEFT JOIN phone USING(name);
```

name varchar	email varchar	salary double	dept varchar	phone varchar
Bob	bob@company.com	40000.0	Accounting	919 555-1111
Carol	carol@company.com	30000.0	Sales	919 555-2222
Eve	eve@company.com	44000.0	Sales	919 555-3333
Frank	frank@comany.com	37000.0	Sales	919 555-4444
Alice	alice@company.com	52000.0	Accounting	NULL
Dave	dave@company.com	33000.0	Accounting	NULL

```
1 SELECT * FROM employees FULL JOIN phone USING(name);
```

name varchar	email varchar	salary double	dept varchar	phone varchar
Bob	bob@company.com	40000.0	Accounting	919 555-1111
Carol	carol@company.com	30000.0	Sales	919 555-2222
Eve	eve@company.com	44000.0	Sales	919 555-3333
Frank	frank@comany.com	37000.0	Sales	919 555-4444
Alice	alice@company.com	52000.0	Accounting	NULL
Dave	dave@company.com	33000.0	Accounting	NULL

```
1 SELECT * FROM employees RIGHT JOIN phone USING(name);
```

name varchar	email varchar	salary double	dept varchar	phone varchar
Bob	bob@company.com	40000.0	Accounting	919 555-1111
Carol	carol@company.com	30000.0	Sales	919 555-2222
Eve	eve@company.com	44000.0	Sales	919 555-3333
Frank	frank@comany.com	37000.0	Sales	919 555-4444

```
1 SELECT * FROM employees ANTI JOIN phone USING(name);
```

name varchar	email varchar	salary double	dept varchar
Alice	alice@company.com	52000.0	Accounting
Dave	dave@company.com	33000.0	Accounting

Subqueries

Tables can be nested within tables for the purpose of creating more complex queries,

```
SELECT * FROM (
  SELECT * FROM employees NATURAL LEFT JOIN phone
) combined WHERE phone IS NULL;
```

name varchar	email varchar	salary double	dept varchar	phone varchar
Alice	alice@company.com	52000.0	Accounting	NULL
Dave	dave@company.com	33000.0	Accounting	NULL

```
SELECT * FROM (
  SELECT * FROM employees NATURAL LEFT JOIN phone
) combined WHERE phone IS NOT NULL;
```

name varchar	email varchar	salary double	dept varchar	phone varchar
Bob	bob@company.com	40000.0	Accounting	919 555-1111
Carol	carol@company.com	30000.0	Sales	919 555-2222
Eve	eve@company.com	44000.0	Sales	919 555-3333
Frank	frank@comany.com	37000.0	Sales	919 555-4444

Common Table Expressions

A CTE (defined with `WITH`) is a named subquery that can be referenced by name in the main query - useful for breaking complex queries into readable steps, or reusing a subquery multiple times.

```
1 WITH combined AS (
2   SELECT * FROM employees NATURAL LEFT JOIN phone
3 )
4 SELECT * FROM combined WHERE phone IS NULL;
```

name varchar	email varchar	salary double	dept varchar	phone varchar
Alice	alice@company.com	52000.0	Accounting	NULL
Dave	dave@company.com	33000.0	Accounting	NULL

Exercise 2

1. What percentage of the total payroll does each department account for?
2. How much more (or less) than their department's average salary does each employee earn?

Query plan

Setup

To give us a bit more variety (and data), we have created another DuckDB database `flights.duckdb` that contains both `nycflights13::flights` and `nycflights13::planes`, the latter of which has details on each plane identified by tail number.

To create the database you can run,

```
db = DBI::dbConnect(duckdb::duckdb(), "flights.duckdb")
dplyr::copy_to(db, nycflights13::flights, name = "flights", temporary = FALSE, ove
dplyr::copy_to(db, nycflights13::planes, name = "planes", temporary = FALSE, overv
DBI::dbDisconnect(db)
```

Alternative you can use the copy provided in the `exercises` repo.

Opening `flights.duckdb`

The database can then be opened from the terminal tab using,

```
> duckdb flights.duckdb
```

Before we start we will set a couple of configuration options so that our output is readable.

We also include `.timer on` so that we get timings for our queries.

```
.maxrows 20
.maxwidth 80
.timer on
```

flights

```
1 SELECT * FROM flights LIMIT 10;
```

year int32	month int32	day int32	… …	distance double	hour double	minute double	time_hour timestamp
2013	1	1	…	1400.0	5.0	15.0	2013-01-01 10:00:00
2013	1	1	…	1416.0	5.0	29.0	2013-01-01 10:00:00
2013	1	1	…	1089.0	5.0	40.0	2013-01-01 10:00:00
2013	1	1	…	1576.0	5.0	45.0	2013-01-01 10:00:00
2013	1	1	…	762.0	6.0	0.0	2013-01-01 11:00:00
2013	1	1	…	719.0	5.0	58.0	2013-01-01 10:00:00
2013	1	1	…	1065.0	6.0	0.0	2013-01-01 11:00:00
2013	1	1	…	229.0	6.0	0.0	2013-01-01 11:00:00
2013	1	1	…	944.0	6.0	0.0	2013-01-01 11:00:00
2013	1	1	…	733.0	6.0	0.0	2013-01-01 11:00:00

```
  10 rows          use .last to show entire result          19 columns (7 shown)
Run Time (s): real 0.001 user 0.000672 sys 0.000502
```

planes

```
1 SELECT * FROM planes LIMIT 10;
```

tailnum varchar	year int32	type varchar	… …	seats int32	speed int32	engine varchar
N10156	2004	Fixed wing multi engine	…	55	NULL	Turbo-fan
N102UW	1998	Fixed wing multi engine	…	182	NULL	Turbo-fan
N103US	1999	Fixed wing multi engine	…	182	NULL	Turbo-fan
N104UW	1999	Fixed wing multi engine	…	182	NULL	Turbo-fan
N10575	2002	Fixed wing multi engine	…	55	NULL	Turbo-fan
N105UW	1999	Fixed wing multi engine	…	182	NULL	Turbo-fan
N107US	1999	Fixed wing multi engine	…	182	NULL	Turbo-fan
N108UW	1999	Fixed wing multi engine	…	182	NULL	Turbo-fan
N109UW	1999	Fixed wing multi engine	…	182	NULL	Turbo-fan
N110UW	1999	Fixed wing multi engine	…	182	NULL	Turbo-fan

10 rows use .last to show entire result 9 columns (6 shown)

Run Time (s): real 0.001 user 0.000518 sys 0.000075

tables

```
1 .tables
```

```
──────────────────── flights ────────────────────
──────────────────────── main ─────────────────────
┌──────────────────────────────┐┌──────────────────────────┐
│           flights            ││          planes          │
│                              ││                          │
│ year            integer      ││ tailnum      varchar     │
│ month           integer      ││ year         integer     │
│ day             integer      ││ type         varchar     │
│ dep_time        integer      ││ manufacturer varchar     │
│ sched_dep_time  integer      ││ model        varchar     │
│ dep_delay       double       ││ engines      integer     │
│ arr_time        integer      ││ seats        integer     │
│ sched_arr_time  integer      ││ speed        integer     │
│ arr_delay       double       ││ engine       varchar     │
│ carrier         varchar      ││                          │
│ flight          integer      ││       3322 rows          │
│ tailnum         varchar      │└──────────────────────────┘
│ origin          varchar      │
│ dest            varchar      │
│ air_time        double       │
│ distance        double       │
│ hour            double       │
│ minute          double       │
│ time_hour       timestamp    │
│                              │
│        336776 rows           │
```

Exercise 3

What is the total number of seats available on all of the planes that flew out of New York in 2013.

A Solution?

Does the following seem correct?

```
1 SELECT sum(seats) FROM flights NATURAL LEFT JOIN planes;
```

sum(seats) int128
614366

```
Run Time (s): real 0.002 user 0.007565 sys 0.000339
```

Why or why not?

Correct solution

Join and select:

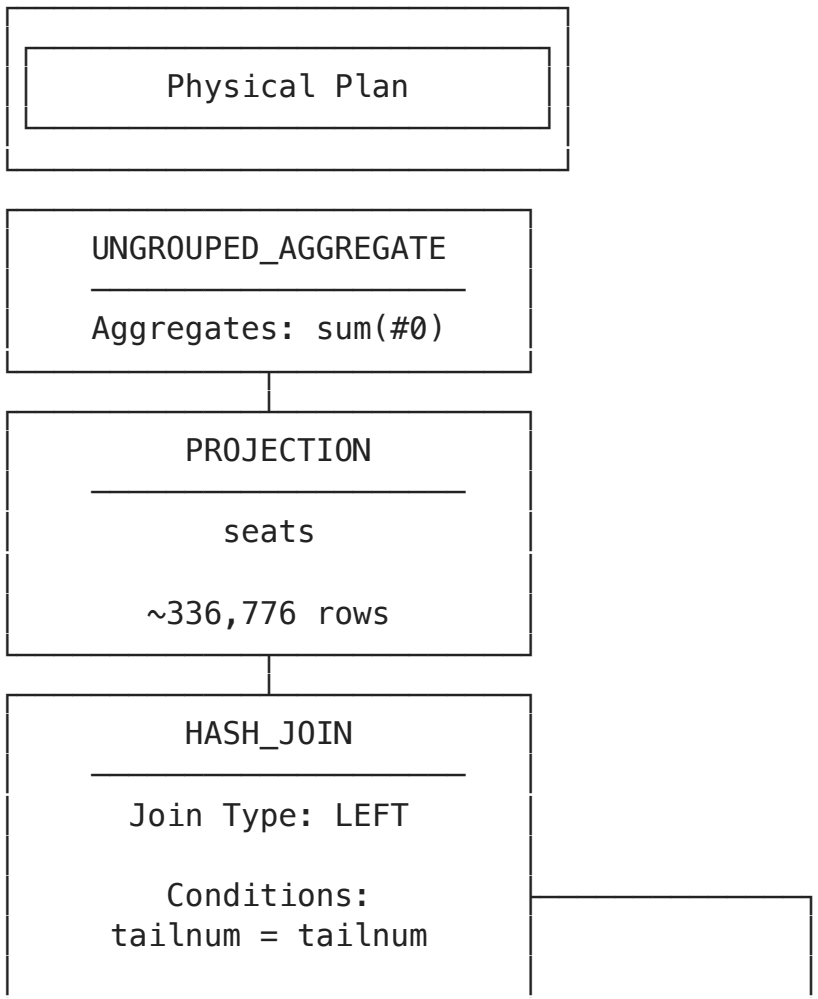
```
1 SELECT sum(seats) FROM flights LEFT JOIN planes USING (tailnum);
```

sum(seats)
int128
38851317

```
Run Time (s): real 0.003 user 0.007598 sys 0.000089
```

EXPLAIN

```
1 EXPLAIN SELECT sum(seats) FROM flights LEFT JOIN planes USING (tailnum);
```



`EXPLAIN` - key operators

Operator	Description
`SEQ_SCAN`	Read every row of a table sequentially
`FILTER`	Apply a `WHERE` condition to rows
`PROJECTION`	Compute and select output columns (`SELECT`)
`HASH_JOIN`	Join two tables by building a hash table on the smaller one
`HASH_GROUP_BY`	Group rows and compute aggregates using a hash table (`GROUP BY`)
`UNGROUPED_AGGREGATE`	Compute a single aggregate over all rows (no `GROUP BY`)
`CROSS_PRODUCT`	Cartesian product of two inputs (`CROSS JOIN`)
`ORDER_BY`	Sort rows (`ORDER BY`)
`TOP_N`	Efficiently returns the top N sorted rows (`ORDER BY` + `LIMIT`)

EXPLAIN ANALYZE

```
1 EXPLAIN ANALYZE SELECT sum(seats) FROM flights LEFT JOIN planes USING (tailnum);
```

```
┌─────────────────────────────┐
│┌───────────────────────────┐│
││ Query Profiling Information ││
│└───────────────────────────┘│
└─────────────────────────────┘
```

```
EXPLAIN ANALYZE SELECT sum(seats) FROM flights LEFT JOIN planes USING (tailnum);
```

```
┌─────────────────────────────┐
│┌───────────────────────────┐│
││ Total Time: 0.0032s       ││
│└───────────────────────────┘│
└─────────────────────────────┘
```

```
┌─────────────────────────────┐
│           QUERY             │
└──────────────┬──────────────┘
┌──────────────┴──────────────┐
│      EXPLAIN_ANALYZE        │
│    ──────────────────       │
│                             │
│           0 rows            │
│           0.00s             │
└──────────────┬──────────────┘
┌──────────────┴──────────────┐
│    UNGROUPED_AGGREGATE      │
│    ──────────────────       │
│    Aggregates: sum(#0)      │
│                             │
```

dplyr

```r
library(dplyr)
flights = nycflights13::flights
planes = nycflights13::planes

system.time({
  flights |>
    left_join(planes, by = c("tailnum" = "tailnum")) |>
    summarise(total_seats = sum(seats, na.rm = TRUE))
})
```

```
 user  system elapsed 
0.050   0.004   0.054 
```

A more complex query

For each carrier and departure airport, how much better or worse (as a %) is their average departure delay compared to the overall average?

```
WITH overall AS (
  SELECT AVG(dep_delay) AS avg_delay FROM flights
)
SELECT carrier,
       ROUND(AVG(dep_delay), 2) AS carrier_avg_del
       ROUND((AVG(dep_delay) - avg_delay)
             / avg_delay, 3)
       AS pct_vs_overall
FROM flights CROSS JOIN overall
GROUP BY carrier, avg_delay
ORDER BY pct_vs_overall;
```

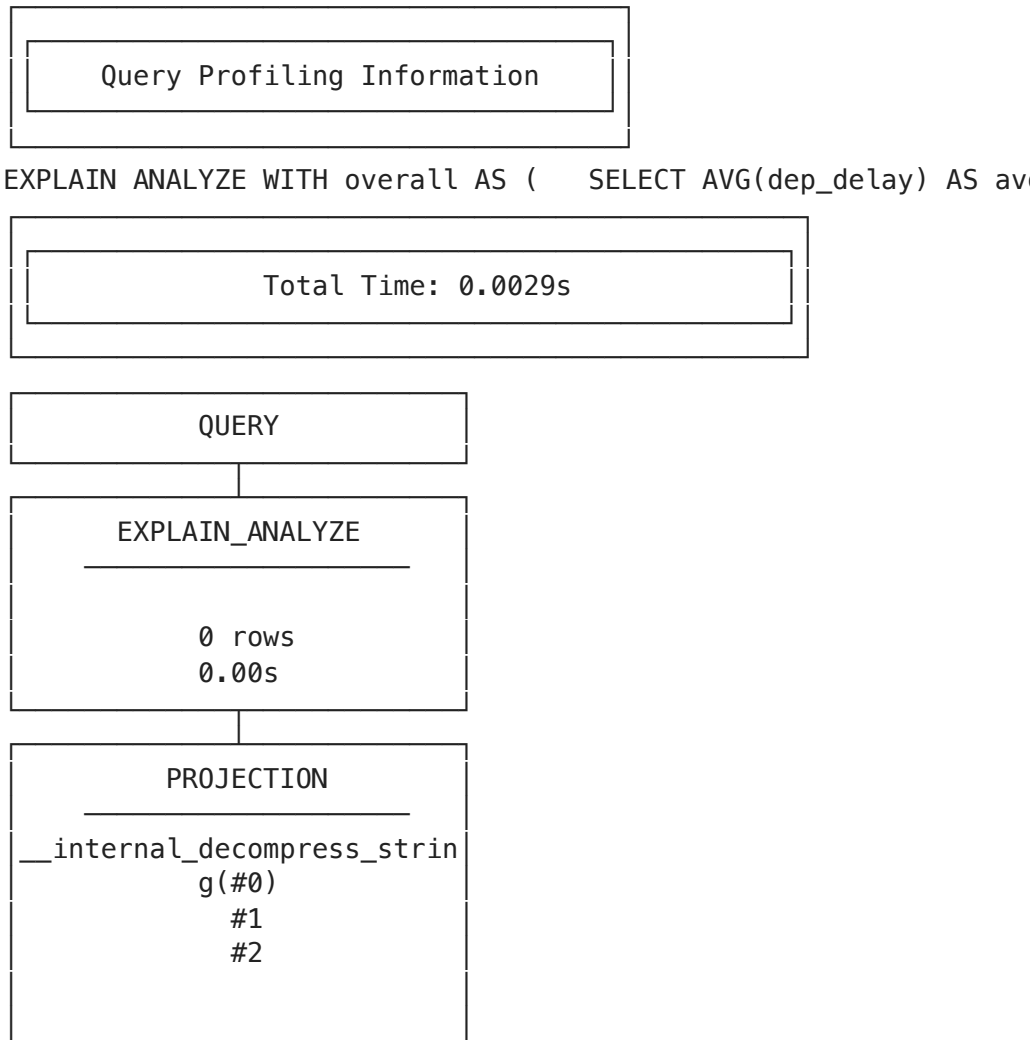
carrier varchar	carrier_avg_delay double	pct_vs_overall double
US	3.78	-0.701
HA	4.9	-0.612
AS	5.8	-0.541
AA	8.59	-0.321
DL	9.26	-0.267
MQ	10.55	-0.165
UA	12.11	-0.042
OO	12.59	-0.004
VX	12.87	0.018
B6	13.02	0.03
9E	16.73	0.323
WN	17.71	0.401
FL	18.73	0.482
YV	19.0	0.503
EV	19.96	0.579
F9	20.22	0.599

16 rows 3 columns

Run Time (s): real 0.003 user 0.007670 sys 0.002178

EXPLAIN ANALYZE

```
1  EXPLAIN ANALYZE
2  WITH overall AS (
3    SELECT AVG(dep_delay) AS avg_delay FROM flights
4  )
5  SELECT carrier,
6         ROUND(AVG(dep_delay), 2) AS carrier_avg_delay,
7         ROUND((AVG(dep_delay) - avg_delay)
8               / avg_delay, 3)
9         AS pct_vs_overall
10 FROM flights CROSS JOIN overall
11 GROUP BY carrier, avg_delay
12 ORDER BY pct_vs_overall;
```



dplyr

```r
system.time({
  overall_avg = nycflights13::flights |>
    summarise(avg_delay = mean(dep_delay, na.rm = TRUE)) |>
    pull(avg_delay)

  nycflights13::flights |>
    group_by(carrier) |>
    summarise(carrier_avg_delay = round(mean(dep_delay, na.rm = TRUE), 2), .groups
    mutate(pct_vs_overall = round((carrier_avg_delay - overall_avg) / overall_avg,
    arrange(pct_vs_overall)
})
```

```
   user  system elapsed 
  0.011   0.001   0.012 
```

NYC Taxi Demo

NYC Taxi Data

The NYC TLC Trip Record Data is a large public dataset of taxi and for-hire vehicle trips in New York City, published monthly since 2009.

Each row is one trip and includes:

Column	Description
`tpep_pickup_datetime` / `tpep_dropoff_datetime`	Trip start and end times
`trip_distance`	Distance traveled (miles)
`PULocationID` / `DOLocationID`	Pickup and dropoff taxi zone
`fare_amount`, `tip_amount`, `total_amount`	Fare components
`payment_type`	How the fare was paid (card, cash, etc.)
`passenger_count`	Number of passengers

Recent years are distributed as parquet files (one per month), with each file containing millions of rows.

Taxi data

```
(d = fs::dir_info("~/Scratch/nyctaxi/") |>
  dplyr::transmute(path = fs::path_file(path), size))
```

```
# A tibble: 73 × 2
   path                                  size
   <chr>                          <fs::bytes>
 1 taxi_zone_lookup.csv                12.04K
 2 yellow_tripdata_2020-01.parquet     89.23M
 3 yellow_tripdata_2020-02.parquet     87.87M
 4 yellow_tripdata_2020-03.parquet     42.38M
 5 yellow_tripdata_2020-04.parquet      4.24M
 6 yellow_tripdata_2020-05.parquet      5.94M
 7 yellow_tripdata_2020-06.parquet      9.07M
 8 yellow_tripdata_2020-07.parquet     12.77M
 9 yellow_tripdata_2020-08.parquet     15.83M
10 yellow_tripdata_2020-09.parquet     20.39M
# ℹ 63 more rows
```

```
sum(d$size)
```

```
3.38G
```

Parquet

Parquet is a binary, columnar file format widely used in data science and analytics.

Row-oriented (CSV)

```
name,  salary, dept
Alice, 52000,  Accounting
Bob,   40000,  Accounting
Carol, 30000,  Sales
```

Reading one column requires reading every row.

Column-oriented (Parquet)

```
name:   [Alice, Bob, Carol, ...]
salary: [52000, 40000, 30000, ..
dept:   [Accounting, Accounting,
```

Reading one column skips all others entirely.

Key advantages over CSV: *compressed* (much smaller files), *typed* (no schema inference), *fast for analytical queries* (only reads needed columns). Widely used with Spark, Arrow, Pandas, and cloud data warehouses.

Reading from Parquet files

DuckDB can query Parquet files directly - no explicit import step needed.

```
D SELECT * FROM
D   '~/Scratch/nyctaxi/yellow_tripdata_2024-01.parquet'
D   LIMIT 5;
```

```
D SELECT * FROM
D   '~/Scratch/nyctaxi/yellow_tripdata_*.parquet'
D   LIMIT 5;
```

Glob patterns let you query multiple files as a single table - useful for partitioned datasets.

The same `CREATE TABLE` / `CREATE VIEW` pattern from CSV works with Parquet:

```
D CREATE VIEW taxi AS SELECT * FROM
D   read_parquet('~/Scratch/nyctaxi/yellow_tripdata_*.parquet');
```

DuckDB can also read remote Parquet files directly via HTTPS, enabling queries over S3 or other cloud storage without

Setup

Enable timings and create a view over all monthly Parquet files:

```
.timer on
CREATE VIEW taxi AS SELECT * FROM
  read_parquet('~/Scratch/nyctaxi/yellow_*.parquet');
```

```
Run Time (s): real 0.001 user 0.000336 sys 0.000338
```

How many rows?

```
SELECT count(*) FROM taxi;
```

```
┌──────────────┐
│ count_star() │
│    int64     │
├──────────────┤
│    223412046 │
└──────────────┘
```

```
Run Time (s): real 0.011 user 0.110206 sys 0.012373
```

Schema

```
1 DESCRIBE SELECT * FROM taxi;
```

```
┌─────────────────────────────────────┐
│                taxi                 │
│                                     │
│ VendorID               bigint       │
│ tpep_pickup_datetime   timestamp    │
│ tpep_dropoff_datetime  timestamp    │
│ passenger_count        double       │
│ trip_distance          double       │
│ RatecodeID             double       │
│ store_and_fwd_flag     varchar      │
│ PULocationID           bigint       │
│ DOLocationID           bigint       │
│ payment_type           bigint       │
│ fare_amount            double       │
│ extra                  double       │
│ mta_tax                double       │
│ tip_amount             double       │
│ tolls_amount           double       │
│ improvement_surcharge  double       │
│ total_amount           double       │
│ congestion_surcharge   double       │
│ airport_fee            integer      │
└─────────────────────────────────────┘
```

```
1 .tables
```

```
┌──────────── memory ────────────┐
└──────────────── main ───────────┘
┌─────────────────────────────────────┐
│                taxi                 │
│                                     │
│ VendorID               bigint       │
│ tpep_pickup_datetime   timestamp    │
│ tpep_dropoff_datetime  timestamp    │
│ passenger_count        double       │
│ trip_distance          double       │
│ RatecodeID             double       │
│ store_and_fwd_flag     varchar      │
│ PULocationID           bigint       │
│ DOLocationID           bigint       │
│ payment_type           bigint       │
│ fare_amount            double       │
│ extra                  double       │
│ mta_tax                double       │
│ tip_amount             double       │
│ tolls_amount           double       │
│ improvement_surcharge  double       │
│ total_amount           double       │
│ congestion_surcharge   double       │
```

Payment types

Create a lookup table for the numeric `payment_type` codes:

```
CREATE TABLE payment_types AS
SELECT * FROM (
  VALUES
    (0, 'Flex Fare trip'),
    (1, 'Credit card'),
    (2, 'Cash'),
    (3, 'No charge'),
    (4, 'Dispute'),
    (5, 'Unknown'),
    (6, 'Voided trip')
) AS t(payment_type, payment_type_desc);

SELECT * FROM payment_types;
```

```
Run Time (s): real 0.000 user 0.000225 sys 0.000
```

payment_type int32	payment_type_desc varchar
0	Flex Fare trip
1	Credit card
2	Cash
3	No charge
4	Dispute
5	Unknown
6	Voided trip

```
Run Time (s): real 0.000 user 0.000139 sys 0.000
```

Tip percentage

What fraction of the fare is tipped on average, by payment type?

```
SELECT payment_type_desc,
       round(avg(tip_amount / fare_amount), 4) AS mean_tip_frac,
       count(*) AS n
FROM taxi
JOIN payment_types USING (payment_type)
WHERE tip_amount >= 0 AND fare_amount > 0
GROUP BY payment_type, payment_type_desc
ORDER BY payment_type;
```

payment_type_desc varchar	mean_tip_frac double	n int64
Flex Fare trip	0.0542	18478107
Credit card	0.274	161375392
Cash	0.0	36624561
No charge	0.0012	940580
Dispute	0.0009	1453619
Unknown	0.0	20

Run Time (s): real 0.843 user 11.130475 sys 0.086085

Cost per mile

Average fare per mile for each pickup zone, joined with zone names:

```sql
SELECT * FROM (
  SELECT
    PULocationID AS pickup_zone,
    ROUND(AVG(fare_amount), 2) AS fare_amount,
    ROUND(AVG(trip_distance), 2) AS trip_distance,
    ROUND(AVG(fare_amount / trip_distance), 2) AS fare_per_mile,
    COUNT(*) AS num_rides
  FROM taxi
  WHERE trip_distance > 0 AND fare_amount > 0
  GROUP BY PULocationID
) NATURAL LEFT JOIN (
  SELECT LocationID AS pickup_zone, *
  FROM read_csv('~/Scratch/nyctaxi/taxi_zone_lookup.csv')
)
ORDER BY fare_per_mile DESC;
```

pickup_zone int64	fare_amount double	trip_distance double	fare_per_mile double	num_rides int64	LocationID int64	Borough varchar	
1	83.98	4.14	2338.06	7245	1	EWR	Newark Ai
216	47.36	24.97	292.28	74147	216	Queens	South Ozo
134	42.17	10.68	240.28	29004	134	Queens	Kew Garde
197	35.53	38.87	144.19	56805	197	Queens	Richmond
84	78.67	25.75	119.01	163	84	Staten Island	Eltingvil
204	84.66	27.6	101.43	144	204	Staten Island	Rossville
120	34.64	7.18	95.25	1937	120	Manhattan	Highbridg
265	45.28	10.03	90.66	283245	265	N/A	Outside o
207	42.14	6.16	89.88	5022	207	Queens	Saint Mic
109	56.26	520.98	88.84	204	109	Staten Island	Great Kil

129	27.39	25.52	68.58	98924	129	Queens	Jackson H
30	33.85	8.57	65.95	353	30	Queens	Broad Cha
221	38.6	51.89	59.04	1061	221	Staten Island	Stapleton
96	31.03	6.67	58.43	1577	96	Queens	Forest Pa
124	37.46	65.86	55.58	13304	124	Queens	Howard Be
180	31.82	13.44	53.74	10302	180	Queens	Ozone Par
219	53.8	55.74	48.42	31840	219	Queens	Springfie
206	51.61	383.92	46.89	594	206	Staten Island	Saint Geo
83	27.44	47.94	44.52	22221	83	Queens	Elmhurst/
145	30.09	11.09	43.2	167963	145	Queens	Long Isla
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

Cost per mile - median

Using `quantile_cont` instead of `AVG` for a more robust estimate:

```sql
SELECT * FROM (
  SELECT
    PULocationID AS pickup_zone,
    ROUND(quantile_cont(fare_amount, 0.5), 2) AS fare_amount,
    ROUND(quantile_cont(trip_distance, 0.5), 2) AS trip_distance,
    ROUND(quantile_cont(fare_amount / trip_distance, 0.5), 2) AS fare_per_mile,
    COUNT(*) AS num_rides
  FROM taxi
  WHERE trip_distance > 0 AND fare_amount > 0
  GROUP BY PULocationID
) NATURAL LEFT JOIN (
  SELECT LocationID AS pickup_zone, *
  FROM read_csv('~/Scratch/nyctaxi/taxi_zone_lookup.csv')
)
ORDER BY fare_per_mile DESC;
```

pickup_zone int64	fare_amount double	trip_distance double	fare_per_mile double	num_rides int64	LocationID int64	Borough varchar	
1	90.0	0.1	625.0	7245	1	EWR	Newark Ai
237	9.3	1.3	7.19	10183946	237	Manhattan	Upper Eas
161	11.5	1.6	7.13	9136233	161	Manhattan	Midtown C
236	10.0	1.49	6.88	9173243	236	Manhattan	Upper Eas
234	10.7	1.51	6.88	5801824	234	Manhattan	Union Sq
163	11.0	1.58	6.87	5887171	163	Manhattan	Midtown N
164	11.4	1.56	6.86	4698717	164	Manhattan	Midtown S
186	12.1	1.6	6.85	7208443	186	Manhattan	Penn Stat
100	11.0	1.5	6.82	3437441	100	Manhattan	Garment D
113	10.7	1.59	6.82	2993363	113	Manhattan	Greenwich

162	11.4	1.6	6.81	7190207	162	Manhattan	Midtown E
90	10.0	1.5	6.75	3579037	90	Manhattan	Flatiron
170	11.0	1.59	6.71	6345855	170	Manhattan	Murray Hi
230	12.1	1.7	6.7	6572024	230	Manhattan	Times Sq/
8	42.9	5.08	6.68	1569	8	Queens	Astoria P
68	12.5	1.8	6.67	5454880	68	Manhattan	East Chel
43	10.7	1.65	6.58	3457031	43	Manhattan	Central P
207	40.8	6.47	6.56	5022	207	Queens	Saint Mic
141	9.5	1.44	6.55	5355630	141	Manhattan	Lenox Hil
246	12.8	1.88	6.54	3462704	246	Manhattan	West Chel
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.