DuckDB & SQL

Lecture 19

Dr. Colin Rundel

SQL

Structures 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

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 976f0 driver=<duckdb_driver dbdir=':memory:'
```

```
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 == 30) |>
  count(origin, dest) |>
  arrange(desc(n))
```

```
# Source:     SQL [?? x 3]
# Database:   DuckDB v1.2.1 [root@Darwin 24.4.0:R 4.4.3/: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      24
 5 LGA    CLT      21
 6 EWR    ORD      18
 7 LGA    BOS      16
 8 EWR    SFO      16
 9 EWR    LAX      16
10 JFK    BOS      16
# ℹ more rows
```

DuckDB CLI

Connecting via CLI

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

```
v1.1.2 f680b7d08f
Enter ".help" for usage hints.
```

```
1 D .
```

Table information

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

```
D .tables
```

```
## employees
```

```
D .schema employees
```

```
## CREATE TABLE employees("name" VARCHAR, email VARCHAR, salary I
```

```
D .indexes employees
```

```
D .maxrows 20
D .maxwidth 80
```

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

SELECT Statements

```
1 D 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 controled via `.mode` - the default is duckbox, see other possible output formats.

```
D .mode csv
D 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
```

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

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

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

```
## [{"name":"Alice","email":"alice@company.
## {"name":"Bob","email":"bob@company.com",
## {"name":"Carol","email":"carol@company.c
## {"name":"Dave","email":"dave@company.com
## {"name":"Eve","email":"eve@company.com",
## {"name":"Frank","email":"frank@comany.cc
```

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

```
## INSERT INTO "table"("name",email,salary,dep
## INSERT INTO "table"("name",email,salary,dep
## INSERT INTO "table"("name",email,salary,dep
## INSERT INTO "table"("name",email,salary,dep
## INSERT INTO "table"("name",email,salary,dep
## INSERT INTO "table"("name",email,salary,dep
```

A brief tour of SQL

select() using SELECT

We can subset for certain columns (and rename them) using `SELECT`

```
D SELECT name AS first_name, salary FROM employees;
```

```
┌────────────┬─────────┐
│ first_name │ salary  │
│  varchar   │ double  │
├────────────┼─────────┤
│ Alice      │ 52000.0 │
│ Bob        │ 40000.0 │
│ Carol      │ 30000.0 │
│ Dave       │ 33000.0 │
│ Eve        │ 44000.0 │
│ Frank      │ 37000.0 │
└────────────┴─────────┘
```

arrange() using ORDER BY

We can sort our results by adding `ORDER BY` to our `SELECT` statement and reverse the ordering by include `DESC`.

```
D SELECT name AS first_name, salary FROM emp
D   ORDER BY salary;
```

first_name varchar	salary double
Carol	30000.0
Dave	33000.0
Frank	37000.0
Bob	40000.0
Eve	44000.0
Alice	52000.0

```
D SELECT name AS first_name, salary FROM emp
D   ORDER BY salary DESC;
```

first_name varchar	salary double
Alice	52000.0
Eve	44000.0
Bob	40000.0
Frank	37000.0
Dave	33000.0
Carol	30000.0

filter() using WHERE

We can filter rows using a `WHERE` clause

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

```
┌─────────┬──────────────────────┬─────────┬─────────┐
│  name   │        email         │ salary  │  dept   │
│ varchar │       varchar        │ double  │ varchar │
├─────────┼──────────────────────┼─────────┼─────────┤
│ Carol   │ carol@company.com    │ 30000.0 │ Sales   │
│ Dave    │ dave@company.com     │ 33000.0 │ Accounting │
│ Frank   │ frank@comany.com     │ 37000.0 │ Sales   │
└─────────┴──────────────────────┴─────────┴─────────┘
```

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

```
┌─────────┬──────────────────────┬─────────┬─────────┐
│  name   │        email         │ salary  │  dept   │
│ varchar │       varchar        │ double  │ varchar │
├─────────┼──────────────────────┼─────────┼─────────┤
│ Carol   │ carol@company.com    │ 30000.0 │ Sales   │
│ Frank   │ frank@comany.com     │ 37000.0 │ Sales   │
└─────────┴──────────────────────┴─────────┴─────────┘
```

group_by() and summarize() via GROUP BY

We can create groups for the purpose of summarizing using `GROUP BY`.

```
D SELECT dept, COUNT(*) AS n FROM employees GROUP BY dept;
```

```
##
##   dept      n
##  varchar  int64
##
##  Sales        3
##  Accounting   3
##
```

head() using LIMIT

We can limit the number of results we get by using `LIMIT`

```
1 D SELECT * FROM employees LIMIT 3;
```

```
##
##   name     email     salary   dept
##  varchar  varchar   double   varchar
##
##  Alice    alice@company.com  52000.0  Accounting
##  Bob      bob@company.com    40000.0  Accounting
##  Carol    carol@company.com  30000.0  Sales
##
```

Exercise 1

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

1. The total costs in payroll for this company
2. The average salary within each department

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 read them into the database and create a table.

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 delimeter)

```
D SELECT * FROM 'phone.csv';
```

```
┌─────────┬──────────────┐
│  name   │    phone     │
│ varchar │   varchar    │
├─────────┼──────────────┤
│ Bob     │ 919 555-1111 │
│ Carol   │ 919 555-2222 │
│ Eve     │ 919 555-3333 │
│ Frank   │ 919 555-4444 │
└─────────┴──────────────┘
```

```
D SELECT * FROM read_csv('phone.csv', delim
```

```
┌─────────┬──────────────┐
│  name   │    phone     │
│ varchar │   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,

```
D .tables
```

```
## employees
```

```
D CREATE TABLE phone AS
D   SELECT * FROM 'phone.csv';
D .tables
```

```
## employees  phone
```

```
D SELECT * FROM phone;
```

```
##
##   name     phone
##  varchar   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

```
D .tables
```

```
## employees
```

```
D CREATE VIEW phone_view AS
D   SELECT * FROM 'phone.csv';
D .tables
```

```
## employees  phone  phone_view
```

```
D SELECT * FROM phone_view;
```

```
##
##   name    │     phone
##  varchar  │    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 D DROP TABLE phone;
2 D DROP VIEW phone_view;
```

Joins - Default

If not otherwise specified the default join in DuckDB will be an inner join.

```
D SELECT * FROM employees JOIN phone;
```

```
## Parser Error: syntax error at or near ";"
## LINE 1: SELECT * FROM employees JOIN phone;
```

Note that an `ON` or `USING` clause is required unless using `NATURAL`.

```
D SELECT * FROM employees NATURAL JOIN phone;
```

```
##
##  name    email           salary  dept        phone
##  varchar varchar         double  varchar     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
##
```

Inner Join - Explicit

```
D SELECT * FROM employees JOIN phone ON employees.name = phone.name;
```

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

to avoid the duplicate `name` column we can specify `USING` instead of `ON`

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

```
##
##  name    email    salary  dept    phone
##  varchar varchar  double  varchar 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
##
```

Left Join - Natural

```
1 D SELECT * FROM employees NATURAL LEFT JOIN phone;
```

```
##
##  name    email    salary  dept    phone
##  varchar varchar  double  varchar 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
##  Dave    dave@company.com    33000.0 Accounting
##
```

Left Join - Explicit

```
D SELECT * FROM employees LEFT JOIN phone ON employees.name = phone.name;
```

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

duplicate `name` column can be avoided by more restrictive `SELECT`,

```
D SELECT employees.*, phone FROM employees LEFT JOIN phone ON employees.name = phone.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	

Other Joins

As you would expect all other standard joins are supported including `RIGHT JOIN`, `FULL JOIN`, `CROSS JOIN`, `SEMI JOIN`, `ANTI JOIN`, etc.

```
1 D SELECT employees.*, phone FROM employees NATURAL FULL JOIN phone;
```

```
##
##   name    │      email      │ salary │   dept    │    phone
##  varchar  │     varchar     │ double │  varchar  │    varchar
##
## ──────────┼─────────────────┼────────┼───────────┼──────────────
##  Bob      │ bob@company.com │ 40000  │ Accounting│ 919 555-1111
##  Carol    │ carol@company.com│ 30000 │ Sales     │ 919 555-2222
##  Eve      │ eve@company.com │ 44000  │ Sales     │ 919 555-3333
##  Frank    │ frank@comany.com│ 37000  │ Sales     │ 919 555-4444
##  Alice    │ alice@company.com│ 52000 │ Accounting│
##  Dave     │ dave@company.com│ 33000  │ Accounting│
##
```

```
1 D SELECT employees.*, phone FROM employees NATURAL RIGHT JOIN phone;
```

```
##
##   name    │      email      │ salary │   dept    │    phone
##  varchar  │     varchar     │ double │  varchar  │    varchar
##
## ──────────┼─────────────────┼────────┼───────────┼──────────────
##  Bob      │ bob@company.com │ 40000  │ Accounting│ 919 555-1111
##  Carol    │ carol@company.com│ 30000 │ Sales     │ 919 555-2222
##  Eve      │ eve@company.com │ 44000  │ Sales     │ 919 555-3333
##  Frank    │ frank@comany.com│ 37000  │ Sales     │ 919 555-4444
##
```

Subqueries

We can nest tables within tables for the purpose of queries.

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

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

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

```
##
##  name    email    salary  dept    phone
##  varchar varchar  double  varchar 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
##
```

Exercise 2

Lets try to create a table that has a new column - `abv_avg` which contains how much more (or less) than the average, for their department, each person is paid.

Hint - This will require joining a subquery.

Query plan

Setup

To give us a bit more variety (and data), we have created another SQLite database `flights.sqlite` that contains both `nycflights13::flights` and `nycflights13::planes`, the latter of which has details on the characteristics of the planes in the dataset as identified by their tail numbers.

```
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, overw
DBI::dbDisconnect(db)
```

All of the following code will be run in the DuckDB command line interface, make sure you've created the database and copied both the flights and planes tables into the db or use the version provided in the `exercises/` repo.

Opening `flights.sqlite`

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

```
> duckdb flights.duckdb
```

As before 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.

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

flights

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

```
##
##  year  month  day  …  distance  hour  minute  time_hour
##  int32  int32  int32     double  double  double  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                                  19 columns (7 shown)
##
## Run Time (s): real 0.020 user 0.000784 sys 0.002284
```

planes

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

```
##
##  tailnum  year  type                    …  seats  speed  engine
##  varchar  int32  varchar                 int32  int32  varchar
##
##  N10156    2004  Fixed wing multi e…  …     55         Turbo-fan
##  N102UW    1998  Fixed wing multi e…  …    182         Turbo-fan
##  N103US    1999  Fixed wing multi e…  …    182         Turbo-fan
##  N104UW    1999  Fixed wing multi e…  …    182         Turbo-fan
##  N10575    2002  Fixed wing multi e…  …     55         Turbo-fan
##  N105UW    1999  Fixed wing multi e…  …    182         Turbo-fan
##  N107US    1999  Fixed wing multi e…  …    182         Turbo-fan
##  N108UW    1999  Fixed wing multi e…  …    182         Turbo-fan
##  N109UW    1999  Fixed wing multi e…  …    182         Turbo-fan
##  N110UW    1999  Fixed wing multi e…  …    182         Turbo-fan
##
##  10 rows                                  9 columns (6 shown)
##
## Run Time (s): real 0.003 user 0.000819 sys 0.000018
```

Exercise 3

Write a query that determines the total number of seats available on all of the planes that flew out of New York in 2013.

Solution?

Does the following seem correct?

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

```
##
##  ┌──────────────┐
##  │ sum(seats)   │
##  │   int128     │
##  ├──────────────┤
##  │       614366 │
##  └──────────────┘
##
## Run Time (s): real 0.012 user 0.016061 sys 0.002386
```

Why?

Correct solution

Join and select:

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

```
## ┌──────────────┐
## │ sum(seats)   │
## │   int128     │
## ├──────────────┤
## │   38851317   │
## └──────────────┘
## Run Time (s): real 0.005 user 0.010150 sys 0.000291
```

EXPLAIN

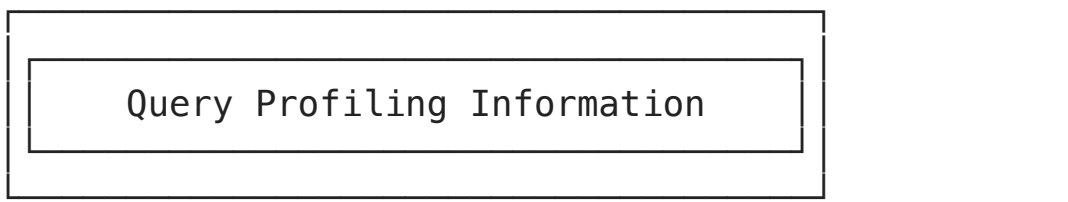
```
1 D EXPLAIN SELECT sum(seats) FROM flights LEFT JOIN planes USING (tailnum);
```

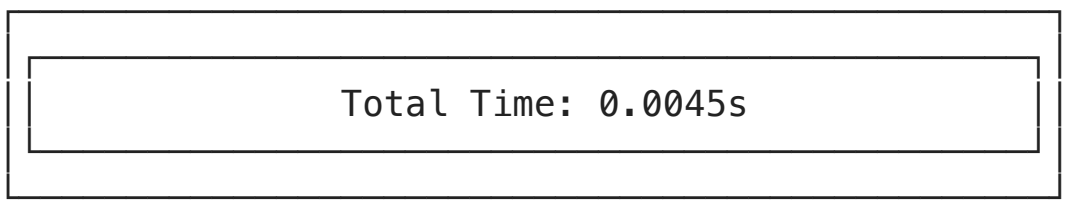
```
## ┌─────────────────────────────┐
## │┌───────────────────────────┐│
## ││       Physical Plan       ││
## │└───────────────────────────┘│
## └─────────────────────────────┘
## ┌───────────────────────────┐
## │    UNGROUPED_AGGREGATE    │
## │    ────────────────────   │
## │    Aggregates: sum(#0)    │
## └─────────────┬─────────────┘
## ┌─────────────┴─────────────┐
## │         PROJECTION        │
## │    ────────────────────   │
## │           seats           │
## │                           │
## │       ~336776 Rows        │
## └─────────────┬─────────────┘
## ┌─────────────┴─────────────┐
## │         HASH_JOIN         │
## │    ────────────────────   │
## │      Join Type: LEFT      │
## │                           │
## │        Conditions:        ├──────────────┐
```

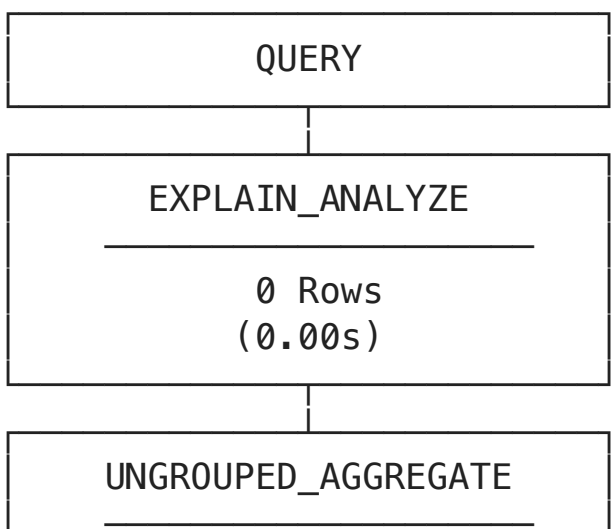
EXPLAIN ANALYZE

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

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







dplyr

```
1 library(dplyr)
2 flights = nycflights13::flights
3 planes = nycflights13::planes
4
5 system.time({
6   flights |>
7     left_join(nycflights13::planes, by = c("tailnum" = "tail
8     summarise(total_seats = sum(seats, na.rm = TRUE))
9 })
```

```
   user  system elapsed
  0.046   0.003   0.050
```

NYC Taxi Demo