DuckDB

Lecture 17

Dr. Colin Rundel

DuckDB

DuckDB is an open source, in-process, analytical relational database. It was created at CWI in the Netherlands and first released in 2019.

- In-process means there is no server. The engine is a library that runs inside R, Python, or the CLI, and a database is a single file.
- Analytical means online analytical processing (OLAP) vs online transaction processing (OLTP).
- Data is stored by column.
- Queries are multithreaded and use all available CPU cores by default.
- CSV, JSON, and Parquet files can be queried directly without an explicit import step.

OLTP vs OLAP, in-process vs server



Where we left off

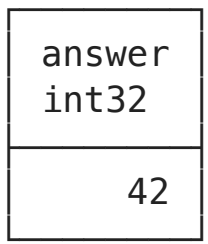
Last time we used DuckDB as the engine behind `DBI` and `dbplyr` in R and the DB-API in Python.

```r
con = DBI::dbConnect(duckdb::duckdb())
DBI::dbGetQuery(con, "SELECT 42 AS answer")
```

```
  answer
1     42
```

```r
DBI::dbDisconnect(con, shutdown = TRUE)
```

```python
import duckdb
duckdb.sql("SELECT 42 AS answer")
```



Today the focus is the SQL itself, using DuckDB's command line interface (CLI). Everything we use today works in the same way when run through `dbGetQuery()` or `duckdb.sql()`.

The DuckDB CLI

Installation

The CLI is a single executable with no dependencies. It is separate from the R and Python packages, which bundle the engine but not the shell.

```bash
# macOS
brew install duckdb
```

```bash
# Linux / macOS
curl https://install.duckdb.org | sh
```

```bash
# Windows
winget install DuckDB.cli
```

The CLI is already installed on the department servers.

DuckDB v2.0.0 has just been released and is the version used in these slides, but everything here also works with older versions.

Connecting

Just like with `DBI`, running `duckdb` with a file path opens (or creates) that database. With no path the database is initialized in memory and discarded on exit.

```bash
duckdb data/employees.duckdb
```

```
DuckDB v2.0.0 (Cyanoptera) d54cbbe3df
Enter ".help" for usage hints.
employees D
```

Once connected you type SQL statements, ending each with a semicolon.

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

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

The SQL in this deck is run through the CLI when the slides are built (via the `duckknit` package). The `D` prompt is

Dot commands

Commands beginning with `.` are instructions to the shell rather than SQL. They do not need a semicolon.

DuckDB

```
D .tables
```

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

DuckDB

```
D .schema employees
```

```
CREATE TABLE employees(
    "name" VARCHAR,
    salary DOUBLE,
    dept VARCHAR
);
```

DuckDB

```
D .version
```

```
DuckDB v2.0.0 (Cyanoptera) d54cbbe3df
clang-21.0.0
```

A first query

```
D SELECT * FROM employees;
```

DuckDB

name varchar	salary double	dept varchar
Alice	52000.0	Accounting
Bob	40000.0	Accounting
Carol	30000.0	Sales
Dave	33000.0	Accounting
Eve	44000.0	Sales
Frank	37000.0	Sales

SQL keywords are case insensitive. By convention they are written in upper case and identifiers in lower case. Whitespace and line breaks are ignored, and a query is ended by a semicolon.

Output formats

The default output format is called `duckbox`, alternative formats can be selected by `.mode`.

DuckDB
```
D .mode csv
```

DuckDB
```
D SELECT name, dept FROM employees LIMIT 3;
```

```
name,dept
Alice,Accounting
Bob,Accounting
Carol,Sales
```

DuckDB
```
D .mode markdown
```

DuckDB
```
D SELECT name, dept FROM employees LIMIT 3;
```

```
| name  |    dept     |
|-------|-------------|
| Alice | Accounting  |
| Bob   | Accounting  |
| Carol | Sales       |
```

DuckDB
```
D .mode json
```

DuckDB
```
D SELECT name, dept FROM employees LIMIT 3;
```

```
[{"name":"Alice","dept":"Accounting"},
{"name":"Bob","dept":"Accounting"},
{"name":"Carol","dept":"Sales"}]
```

DuckDB
```
D .mode line
```

DuckDB
```
D SELECT name, dept FROM employees LIMIT 3;
```

```
name = Alice
dept = Accounting

name = Bob
dept = Accounting

name = Carol
dept = Sales
```

Files as tables

Files as tables

DuckDB can treat a CSV (or Parquet or JSON) file as a table, so a file can be queried without importing it.

DuckDB

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

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

DuckDB

```
D SELECT * FROM
    read_csv('data/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 files

A table can be created from any query with `CREATE TABLE ... AS`, including a query on a file.

DuckDB

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

DuckDB

```
D SHOW TABLES;
```

name
varchar
employees
phone

DuckDB

```
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

A view stores the query rather than its result. It can be used like a table but is re-evaluated each time, so the file is read on every query.

DuckDB

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

DuckDB

```
D SHOW TABLES;
```

name
varchar
employees
phone
phone_view

DuckDB

```
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

Dropping tables and views

```
D DROP TABLE phone;
```
DuckDB

```
D DROP VIEW phone_view;
```
DuckDB

```
D SHOW TABLES;
```
DuckDB

name
varchar
employees

Dropping something that does not exist is an error, but `IF EXISTS` makes it a no-op. Along with `CREATE OR REPLACE` this lets a script be re-run safely.

```
D DROP VIEW phone_view;
```
DuckDB

```
Catalog Error: View with name phone_view
does not exist!
Did you mean "pg_views"?
```

```
D DROP VIEW IF EXISTS phone_view;
```
DuckDB

```
D CREATE OR REPLACE TABLE phone AS
    SELECT * FROM 'data/phone.csv';
```
DuckDB

DuckDB's SQL

Friendly SQL

DuckDB extends standard SQL with a collection of shortcuts that remove boilerplate. They are not portable to other databases, but they make interactive work much more pleasant.

Some commonly used examples,

- `FROM` first syntax and an optional `SELECT`
- `GROUP BY ALL` and `ORDER BY ALL`
- `SELECT * EXCLUDE (...)` and `SELECT * REPLACE (...)`
- `DESCRIBE` and `SUMMARIZE`
- `SEMI JOIN`, `ANTI JOIN`, and `QUALIFY`

FROM first

Starting with `FROM` lets you name the table before the columns, which matches the order you think in (and the order `dplyr` pipes run in). `SELECT *` is implied when it is omitted.

DuckDB

```
D FROM employees;
```

name varchar	salary double	dept varchar
Alice	52000.0	Accounting
Bob	40000.0	Accounting
Carol	30000.0	Sales
Dave	33000.0	Accounting
Eve	44000.0	Sales
Frank	37000.0	Sales

DuckDB

```
D FROM employees
  SELECT name, salary
  ORDER BY salary DESC
  LIMIT 2;
```

name varchar	salary double
Alice	52000.0
Eve	44000.0

BY ALL

`GROUP BY ALL` groups by every non-aggregated column in the `SELECT`, so the grouping columns do not need to be listed twice. `ORDER BY ALL` sorts by every selected column, left to right.

DuckDB

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

dept varchar	n int64
Accounting	3
Sales	3

DuckDB

```
D FROM employees
  SELECT dept, count(*) AS n
  GROUP BY ALL
  ORDER BY ALL;
```

dept varchar	n int64
Accounting	3
Sales	3

EXCLUDE and REPLACE

`SELECT *` can be modified to drop columns or replace them with an expression, rather than listing every column that should be kept.

DuckDB

```
D SELECT * EXCLUDE (dept)
  FROM employees LIMIT 3;
```

name varchar	salary double
Alice	52000.0
Bob	40000.0
Carol	30000.0

DuckDB

```
D SELECT * REPLACE (salary / 1000 AS salary)
  FROM employees LIMIT 3;
```

name varchar	salary double	dept varchar
Alice	52.0	Accounting
Bob	40.0	Accounting
Carol	30.0	Sales

DESCRIBE and SUMMARIZE

`DESCRIBE` reports the columns and types of a table or query. `SUMMARIZE` computes summary statistics for every column.

DuckDB

```
D DESCRIBE employees;
```

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

DuckDB

```
D SUMMARIZE employees;
```

column_name varchar	column_type varchar	min varchar	max varchar	… …	q75 varchar	count int64	null_percentage decimal(9,2)
name	VARCHAR	Alice	Frank	…	NULL	6	0.00
salary	DOUBLE	30000.0	52000.0	…	44000.0	6	0.00
dept	VARCHAR	Accounting	Sales	…	NULL	6	0.00

3 rows use .last to show entire result 12 columns (7 shown)

Semi and anti joins

DuckDB also adds join types. `SEMI JOIN` keeps the rows of the left table that have a match, and `ANTI JOIN` keeps those that do not. Neither adds columns from the right table.

DuckDB

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

name varchar	salary double	dept varchar
Bob	40000.0	Accounting
Carol	30000.0	Sales
Eve	44000.0	Sales
Frank	37000.0	Sales

DuckDB

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

name varchar	salary double	dept varchar
Alice	52000.0	Accounting
Dave	33000.0	Accounting

These are the same operations as `semi_join()` and `anti_join()` in `dplyr`. In standard SQL they are written with `WHERE`

A larger database

flights.duckdb

The `flights.duckdb` database contains `nycflights13::flights` and `nycflights13::planes` as tables. It was built from Parquet files with the file-as-table trick,

```
CREATE TABLE flights AS SELECT * FROM 'data/flights.parquet';
CREATE TABLE planes AS SELECT * FROM 'data/planes.parquet';
```

DuckDB

Large results are truncated for display, with the limits set by `.maxrows` and `.maxwidth`.

`.timer on` reports the time taken by each statement.

```
duckdb data/flights.duckdb
```

bash

```
D .maxrows 6
```

DuckDB

```
D .maxwidth 80
```

DuckDB

```
D .timer on
```

DuckDB

flights

DuckDB

```
D SELECT * FROM flights;
```

year int32	month int32	day int32	… …	hour double	minute double	time_hour timestamp with time zone
2013	1	1	…	5.0	15.0	2013-01-01 05:00:00-05
2013	1	1	…	5.0	29.0	2013-01-01 05:00:00-05
2013	1	1	…	5.0	40.0	2013-01-01 05:00:00-05
.	.	.	…	.	.	.
.	.	.	…	.	.	.
.	.	.	…	.	.	.
2013	9	30	…	12.0	10.0	2013-09-30 12:00:00-04
2013	9	30	…	11.0	59.0	2013-09-30 11:00:00-04
2013	9	30	…	8.0	40.0	2013-09-30 08:00:00-04

```
  336776 rows (6 shown)                                19 columns (6 shown)
Run Time (s): real 0.008 user 0.013426 sys 0.006671
```

planes

DuckDB

```
D SELECT * FROM planes;
```

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
.	.	.	…	.	.	.
.	.	.	…	.	.	.
.	.	.	…	.	.	.
N998AT	2002	Fixed wing multi engine	…	100	NULL	Turbo-fan
N998DL	1992	Fixed wing multi engine	…	142	NULL	Turbo-jet
N999DN	1992	Fixed wing multi engine	…	142	NULL	Turbo-jet

3322 rows (6 shown) 9 columns (6 shown)

Run Time (s): real 0.001 user 0.000659 sys 0.000167

Exercise 1

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

Write your query against `flights.duckdb` and check it with `.timer on`.

A solution?

Does the following seem correct?

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

DuckDB

sum(seats)
int128
614366

```
Run Time (s): real 0.003 user 0.005299 sys 0.000579
```

Does this answer make sense? Why or why not?

A correct solution

`flights` and `planes` share two column names, `tailnum` and `year`, but `year` means different things in each table. Naming the join column avoids matching on both.

DuckDB

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

```
┌──────────────┐
│ sum(seats)   │
│   int128     │
├──────────────┤
│   38851317   │
└──────────────┘
Run Time (s): real 0.002 user 0.003479 sys 0.000161
```

Worked examples

Example 1

For each carrier, how does its average departure delay compare to the average over all flights?

DuckDB

```
D SELECT carrier, round(avg(dep_delay), 2) AS carrier_avg
  FROM flights GROUP BY ALL ORDER BY carrier_avg;
```

carrier varchar	carrier_avg double
US	3.78
HA	4.9
AS	5.8
.	.
.	.
.	.
YV	19.0
EV	19.96
F9	20.22

16 rows 2 columns
(6 shown)

```
Run Time (s): real 0.001 user 0.003019 sys 0.001440
```

Scalar subqueries

A query that returns a single value can be used anywhere a value is expected. Here the overall average is computed once and divides every carrier's average.

DuckDB

```
D SELECT carrier,
    round(avg(dep_delay), 2) AS carrier_avg,
    round(carrier_avg / (SELECT avg(dep_delay) FROM flights), 2) AS ratio
  FROM flights
  GROUP BY ALL ORDER BY ratio;
```

carrier varchar	carrier_avg double	ratio double
US	3.78	0.3
HA	4.9	0.39
AS	5.8	0.46
.	.	.
.	.	.
.	.	.
YV	19.0	1.5
EV	19.96	1.58
F9	20.22	1.6

```
  16 rows (6 shown)    3 columns
Run Time (s): real 0.002 user 0.006068 sys 0.002056
```

Common table expressions

`WITH` names a subquery up front so the query reads top to bottom. `CROSS JOIN` adds the one row of `overall` to every flight.

DuckDB

```
D WITH overall AS (
    SELECT avg(dep_delay) AS overall_avg FROM flights
  )
  SELECT carrier, round(avg(dep_delay) / overall_avg, 2) AS ratio
  FROM flights
  CROSS JOIN overall
  GROUP BY carrier, overall_avg ORDER BY ratio;
```

```
┌─────────┬────────┐
│ carrier │ ratio  │
│ varchar │ double │
├─────────┼────────┤
│ US      │    0.3 │
│ HA      │   0.39 │
│ AS      │   0.46 │
│ .       │      . │
│ .       │      . │
│ .       │      . │
│ YV      │    1.5 │
│ EV      │   1.58 │
│ F9      │    1.6 │
└─────────┴────────┘
     16 rows
    (6 shown)
Run Time (s): real 0.003 user 0.006365 sys 0.000743
```

In `dplyr`

```R
system.time({
  res = nycflights13::flights |>
    mutate(overall_avg = mean(dep_delay, na.rm = TRUE)) |>
    summarize(
      carrier_avg = round(mean(dep_delay, na.rm = TRUE), 2),
      ratio = round(carrier_avg / first(overall_avg), 2),
      .by = carrier
    ) |>
    arrange(ratio)
})
```

```
   user  system elapsed
  0.652   0.034   0.694
```

```R
res
```

```
# A tibble: 16 × 3
  carrier carrier_avg ratio
  <chr>         <dbl> <dbl>
1 US             3.78  0.3
2 HA             4.9   0.39
3 AS             5.8   0.46
4 AA             8.59  0.68
5 DL             9.26  0.73
6 MQ            10.6   0.83
# ℹ 10 more rows
```

Example 2

For each origin airport, what are the three most common destinations?

DuckDB

```
D SELECT origin, dest, count(*) AS n
  FROM flights
  GROUP BY ALL ORDER BY origin, n DESC;
```

origin varchar	dest varchar	n int64
EWR	ORD	6100
EWR	BOS	5327
EWR	SFO	5127
.	.	.
.	.	.
.	.	.
LGA	SBN	6
LGA	MYR	3
LGA	LEX	1

224 rows 3 columns
(6 shown)

Window functions

A window function computes a value for each row from the other rows in its partition without collapsing them, like a grouped `mutate()`.

Filtering on that value in `WHERE` does not work as `WHERE` is evaluated before window functions.

DuckDB

```
D WITH counts AS (
    SELECT origin, dest, count(*) AS n FROM flights GROUP BY ALL
  )
  SELECT origin, dest, n,
         rank() OVER (PARTITION BY origin ORDER BY n DESC) AS rank
  FROM counts
  WHERE rank <= 3;
```

```
Binder Error: WHERE clause cannot contain window functions!

LINE 5:         rank() OVER (PARTITION BY origin ORDER BY n DESC) AS rank
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

QUALIFY

The standard fix is to wrap the query in a subquery and filter that. DuckDB adds `QUALIFY`, which filters on window results directly, as `HAVING` does for aggregates.

DuckDB

```
D WITH counts AS (
    SELECT origin, dest, count(*) AS n FROM flights GROUP BY ALL
  )
  SELECT origin, dest, n,
         rank() OVER (PARTITION BY origin ORDER BY n DESC) AS rank
  FROM counts
  QUALIFY rank <= 3
  ORDER BY origin, rank;
```

origin varchar	dest varchar	n int64	rank int64
EWR	ORD	6100	1
EWR	BOS	5327	2
EWR	SFO	5127	3
JFK	LAX	11262	1
JFK	SFO	8204	2
JFK	BOS	5898	3
LGA	ATL	10263	1
LGA	ORD	8857	2
LGA	CLT	6168	3

Exercise 2

Using `flights.duckdb`,

1. For each month, which carrier had the worst mean arrival delay? Use a window function and `QUALIFY`.
2. What fraction of each carrier's flights departed more than 30 minutes late, and how does that compare to the fraction over all flights? Use a CTE.

Beyond one file

Beyond one file

The file-as-table trick extends to many files. A glob pattern reads them as one table, which is how large data sets are usually distributed, e.g. one Parquet file per month.

DuckDB

```
D .shell ls data/flights
```

```
2013-01.parquet
2013-02.parquet
2013-03.parquet
2013-04.parquet
2013-05.parquet
2013-06.parquet
2013-07.parquet
2013-08.parquet
2013-09.parquet
2013-10.parquet
2013-11.parquet
2013-12.parquet
```

DuckDB

```
D SELECT count(*) FROM 'data/flights/*.parquet';
```

count_star()
int64
336776

Remote files

Files can also be remote. DuckDB fetches only the parts of a Parquet file that a query needs, here the footer of a 1.6 MB file holding the row count.

DuckDB

```
D SELECT count(*) FROM 'https://blobs.duckdb.org/train_services.parquet';
```

count_star()
int64
380959

Next time we will look at the Parquet format and why it is so well suited to this, and use `EXPLAIN` to see what DuckDB

217 million rows

The full on-time data

`nycflights13` is one year and three airports from the Bureau of Transportation Statistics on-time data. The whole thing, every domestic flight since 1987, is 38 Parquet files, one per year, available in `/data/rdu/` on the department servers.

DuckDB

```
D .shell du -sh .
```

```
9.7G    .
```

DuckDB

```
D FROM glob('Year=202*/*');
```

file
varchar
Year=2020/data_0.parquet
Year=2021/data_0.parquet
Year=2022/data_0.parquet
Year=2023/data_0.parquet
Year=2024/data_0.parquet

DuckDB

```
D CREATE VIEW ontime AS
    FROM 'Year=*/*.parquet';
```

```
Run Time (s): real 0.003 user 0.001695 sys 0.001245
```

DuckDB

```
D SELECT count(*) FROM ontime;
```

count_star()
int64
217389694

```
Run Time (s): real 0.045 user 0.347191 sys 0.076955
```

Folder names like `Year=2024` are Hive partitioning, which DuckDB turns into a `Year` column. The count is answered from

Four decades of delays

Every one of the 217 million rows contributes to the following query.

DuckDB

```
D SELECT Year // 10 * 10 AS decade,
         count(*) AS flights,
         round(avg(DepDelay), 1) AS dep_delay,
         round(100 * avg(DepDel15), 1) AS pct_late
  FROM ontime
  GROUP BY ALL ORDER BY ALL;
```

decade int64	flights int64	dep_delay double	pct_late double
1980	11555122	7.5	14.9
1990	52694390	7.6	14.6
2000	62857045	8.7	17.9
2010	62561043	9.4	18.3
2020	27722094	10.2	18.3

```
Run Time (s): real 0.303 user 3.429797 sys 0.211091
```

Flying out of RDU

The busiest destination from RDU each year:

DuckDB

```
D WITH rdu AS (
    SELECT Year, Dest, count(*) AS n
    FROM ontime
    WHERE Origin = 'RDU'
    GROUP BY ALL
  )
  SELECT * FROM rdu
  QUALIFY rank() OVER (PARTITION BY Year ORDER BY n DESC) = 1
  ORDER BY Year DESC LIMIT 6;
```

Year int64	Dest varchar	n int64
2024	ATL	2296
2023	JFK	4730
2022	JFK	4573
2021	ATL	4008
2020	ATL	3552
2019	ATL	5290

```
Run Time (s): real 0.150 user 1.707782 sys 0.136680
```

Summary

Takeaways

- DuckDB is an in-process analytical database. The same engine runs inside R, Python, and the CLI, and a database is a single file or lives in memory.
- The CLI is the quickest way to explore data with SQL. Dot commands (`.tables`, `.schema`, `.mode`, `.timer`) control the shell, and everything else is SQL ending in a semicolon.
- Scalar subqueries and CTEs break a complex query into steps, window functions compute within-group values without collapsing rows, and `QUALIFY` filters on them.
- CSV, JSON, and Parquet files can be queried as tables. `CREATE TABLE ... AS` copies a file into the database, while `CREATE VIEW` keeps reading it from disk.
- DuckDB's friendly SQL (`FROM` first, `GROUP BY ALL`, `EXCLUDE`, `SUMMARIZE`) removes a lot of boilerplate at the cost of portability.
- A glob of Parquet files is a table too. A laptop can aggregate hundreds of millions of rows in seconds because only the columns a query touches are read.

Learning more

- DuckDB documentation, in particular the SQL introduction and CLI pages
- DuckDB guides, task-oriented recipes for the CLI, Python, and R
- Friendly SQL
- SQL for Data Scientists by Renée Teate
- Modern SQL style guide