

Data Acquisition, Processing and Mining for AI

Week 02 — Relational Data & Advanced SQL

Silviu Maniu

Why SQL, why DuckDB

acquire → understand → clean → mine

- Last week: one table, in memory, in pandas
- Real datasets are **several tables** that must be combined before anything can be trained on them
- Today: the language that does the combining → **SQL** (structured query language), run in-process with **DuckDB**
- **M0** is due before next week's lab: source + scope + accounts

A title ending in (*) marks **optional** material: context or an extension, dropped first if we run short.

Why SQL in an AI course

- The dataset you train on is almost always **the result of a query**
- SQL is **declarative**: you state *what* you want, the engine picks *how* → it optimizes
- Relational model: **1970**¹, still the default interface to data in 2026

¹Codd, *A Relational Model of Data for Large Shared Data Banks*, CACM 13(6), 1970.

The relational model in one slide

- A **relation** (table) is a **set of rows** over named, typed columns. “set” means **no inherent order**
- A **primary key** identifies a row uniquely (LocationID in the zone table)
- A **foreign key** points at another table’s key (PULocationID in the trips table) → these are columns for joining
- **Fact table** (many rows, the events: trips) + **dimension tables** (few rows, the descriptions: zones); we are in the OLAP case

OLAP vs OLTP (*)

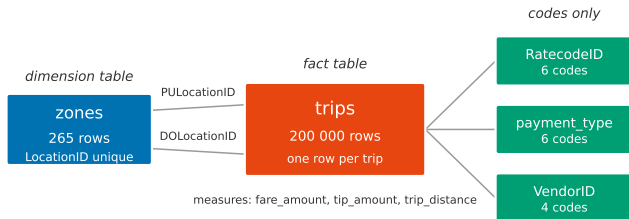
OLTP (on-line transaction processing) – the application's database

- many tiny reads/writes: one booking, one payment
- **normalized** (3NF, third normal form): each fact stored **once**, so an update cannot introduce inconsistency
- row-oriented, indexed for **point lookups**
- Postgres, MySQL, SQLite
- **Same SQL, different designs.** Normalize to keep data *correct*; denormalize to read it *fast* → tradeoff
- Real pipelines do OLTP → **ETL** (extract, transform, load) → OLAP. **This course: OLAP side**

OLAP (on-line analytical processing) – the analytics store

- few huge scans and aggregations
- **denormalized** into fact + dimensions (a **star schema**)
- **columnar**, mostly for aggregations / statistics
- DuckDB, BigQuery, Snowflake, Spark

The taxi data as a star schema



- zones is joined **twice**: PULocationID and DOLocationID both point at LocationID
- Only zones is a table. RatecodeID, payment_type and VendorID are **codes**, described in the data dictionary²
- LocationID is **unique** in zones: a join on it returns one row per trip

²https://www.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf

DuckDB: SQLite for analytics

- **In-process**: a Python library, serverless, just `pip install duckdb`. SQLite is the **transactional** one (OLTP), DuckDB the **analytical** one (OLAP)
- **Columnar** and vectorized: built for **scans and aggregations** over whole columns, not for one-row SELECT
- Reads **Parquet and CSV files directly**, without an import step, and can work on data **larger than RAM**
- Speaks modern, extended SQL dialect: window functions, QUALIFY, sampling, list types, ...

DuckDB in Python

```
import duckdb
```

```
duckdb.sql("SELECT count(*) FROM 'taxi_2025_01_sample.parquet'")
```

```
zones = duckdb.sql("SELECT * FROM 'taxi_zone_lookup.csv'").df()
```

```
duckdb.sql("SELECT Borough, count(*) FROM zones GROUP BY Borough")
```

- A file path **is** a table name; no CREATE TABLE, no loading step [DuckDB]
- `.df()` gives a pandas DataFrame; the DataFrame can be queried **by its Python variable name** [DuckDB]
- Conventions: [DuckDB] = an extension, [standard] = standard, portable to Postgres, Oracle, BigQuery...
- So SQL and pandas are **two views over** the same data

SQL recap

The structure of a SELECT

```
SELECT  PULocationID, count(*) AS trips, avg(fare_amount) AS avg_fare
FROM    'taxi_2025_01_sample.parquet'
WHERE   trip_distance > 0
GROUP BY PULocationID
HAVING  count(*) >= 100
ORDER BY trips DESC
LIMIT  10;
```

- One statement, six clauses, **always in this order** [standard]
- WHERE filters **rows**; HAVING filters **groups** → most common confusion
- LIMIT exception: not standard (which has FETCH FIRST 10 ROWS ONLY), but every engine implements it

Written order $\neq$ evaluation order

FROM $\rightarrow$ WHERE $\rightarrow$ GROUP BY $\rightarrow$ HAVING $\rightarrow$ SELECT $\rightarrow$ ORDER BY $\rightarrow$ LIMIT

- SELECT runs **late** $\rightarrow$ an alias defined there is **not** visible to WHERE or GROUP BY; only ORDER BY, which runs later, sees it [standard]
- [DuckDB] DuckDB relaxes this rule: aliases *do* work in WHERE, GROUP BY and HAVING (used in these slides)
- Aggregates are computed **after** grouping $\rightarrow$ WHERE count(*) > 5 is an error, use HAVING

Filtering

```
WHERE fare_amount BETWEEN 5 AND 100           -- inclusive on both ends
AND payment_type IN (1, 2)
AND store_and_fwd_flag = 'N'
AND tpep_pickup_datetime >= TIMESTAMP '2025-01-08'
```

- =, <>, <, >=, BETWEEN, IN, LIKE 'JFK%' — and AND/OR/NOT
- Strings are in **single** quotes; double quotes mean *identifier* (a column named "Zone")
- Filter **early**: every row you drop here is a row the joins and aggregations below never touch

NULL is not a value

```
SELECT count(*)           AS rows_total,      -- 200 000
       count(passenger_count) AS rows_with_value -- 168 946
FROM   'taxi_2025_01_sample.parquet';
```

- **NULL** means **unknown** → NULL = NULL is **not true**, it is NULL. Use IS NULL / IS NOT NULL
- count(*) counts rows; count(col) counts **non-NULL** values
- Aggregates **skip** NULLs: avg(x) divides by the number of non-NULL values, not by the row count
- coalesce(tip_amount, 0) gets the first non-null value

Aggregation

```
SELECT date_trunc('hour', tpep_pickup_datetime) AS hour,  
       count(*) AS trips,  
       round(avg(fare_amount),2) AS avg_fare,  
       max(trip_distance) AS longest  
FROM   'taxi_2025_01_sample.parquet'  
GROUP BY hour ORDER BY hour;
```

- count, sum, avg, min, max [standard] **Note:** median, quantile_cont are the short spellings of standard percentile_cont(0.5) WITHIN GROUP (ORDER BY x)
- hour(ts) is shorthand for standard EXTRACT(HOUR FROM ts) (0-23).
date_trunc('hour', ts) is **not the same**: it returns a *timestamp*, one group per hour of the month, and has no standard equivalent
- Every non-aggregated column in SELECT must appear in GROUP BY; otherwise the row is undefined
- Remember last week's df.groupby(...).mean()? **same split-apply-combine**, different syntax

CTEs: naming subqueries / steps

```
WITH clean AS (  
    SELECT * FROM 'taxi_2025_01_sample.parquet'  
    WHERE fare_amount > 0 AND trip_distance > 0  
)  
, per_zone AS (  
    SELECT PULocationID, count(*) AS trips  
    FROM clean GROUP BY PULocationID  
)  
SELECT * FROM per_zone WHERE trips >= 1000 ORDER BY trips DESC;
```

- **CTE** (common table expression, written `WITH ... AS`) = a named intermediate result, readable top to bottom [standard] (since SQL:1999)
- Build datasets as a chain of CTEs
- Costs (almost) nothing: the optimizer flattens them

Joins: building the dataset

Two tiny tables

trips (the fact table)

trip	PULoc	fare
t1	132	70.00
t2	141	18.40
t3	132	65.50
t4	99	22.00
t5	141	12.80

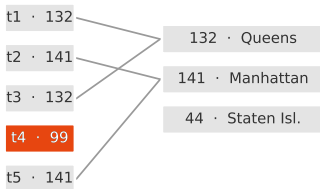
zones (the dimension table)

LocID	Borough	Zone
132	Queens	JFK Airport
141	Manhattan	Lenox Hill W
44	Staten Isl.	Charleston

- Join key: `trips.PULoc = zones.LocID`
- Two rows **match nothing**: trip **t4** contains a code without any zone; zone **44** is not a pickup point
- Every join type below differs **only** in what it does with those two

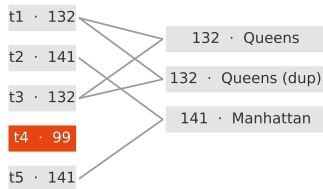
Join = matching

one row per zone: a matching



5 left rows → 4 matched pairs

zone 132 twice: the fan-out



5 left rows → 6 matched pairs

- Every join answers **one question**: which rows to keep, and what to do with unmatched ones
- t4 and zone 44 match nothing
- On the right the key **repeats**, so two trips grow **two lines each**: fan-out

INNER JOIN: keep only matches

```
SELECT t.trip, z.Borough, t.fare  
FROM trips t INNER JOIN zones z ON t.PULoc = z.LocID;
```

trip	Borough	fare
t1	Queens	70.00
t2	Manhattan	18.40
t3	Queens	65.50
t5	Manhattan	12.80

- 5 trips in, 4 rows out
- t4 is gone
- zone 44 is gone as well
- Fewer rows out than in is normal for an inner join

LEFT JOIN: keep everything on the left

```
SELECT t.trip, z.Borough, t.fare  
FROM trips t LEFT JOIN zones z ON t.PULoc = z.LocID;
```

trip	Borough	fare
t1	Queens	70.00
t2	Manhattan	18.40
t3	Queens	65.50
t4	NULL	22.00
t5	Manhattan	12.80

- 5 rows out: t4 survives, with NULL for the zone information
- For when losing rows is worse than having gaps
- FULL JOIN also brings back zone 44 (6 rows); RIGHT JOIN is the mirror (rarer)

SEMI and ANTI: filter by existence

```
-- zones that WERE used as a pickup point + those that never were
SELECT z.* FROM zones z SEMI JOIN trips t ON t.PULoc = z.LocID;
SELECT z.* FROM zones z ANTI JOIN trips t ON t.PULoc = z.LocID;
```

SEMI

LocID	Borough	Zone
132	Queens	JFK Airport
141	Manhattan	Lenox Hill W

ANTI

LocID	Borough	Zone
44	Staten Isl.	Charleston

- Both return **left rows only**: no columns added, and **no row can be duplicated**
- These keywords are **[DuckDB]** (also in Spark); or use WHERE (NOT) EXISTS [standard]
- ANTI is a **coverage check**. On the real tables: **24 of 265** zones saw no pickup, and **0** trips carried a missing code

The fan-out trap

Now suppose JFK got imported **twice** into zones:

LocID	Borough	Zone
132	Queens	JFK Airport
132	Queens	JFK Airport
141	Manhattan	Lenox Hill W

the same INNER JOIN		
trip	Borough	fare
t1	Queens	70.00
t1	Queens	70.00
t3	Queens	65.50
t3	Queens	65.50
t2, t5	Manhattan	...

- **4 rows became 6** and the mean fare changed, but no error
- Check the key **before** joining: `SELECT LocID FROM zones GROUP BY LocID HAVING count(*) > 1`
- The most common dataset bug. At scale it also puts **copies of one row on both sides of a train/test split**

The same join, on the real tables

trips per pickup borough

Borough	trips	avg fare
Manhattan	177 943	13.87
Queens	16 807	45.67
Brooklyn	3 801	24.02
Bronx	857	27.65
Unknown	470	18.11
N/A	82	52.39
EWB	26	70.49
Staten Isl.	14	35.10

- Same query as the toy one, but 200 000 rows instead of 5
- The fare column is an **airport map**: EWR (Newark) 70.49, Queens 45.67 (JFK, LaGuardia) corresponding to long distances and flat rates
- Staten Island: **14 trips** in a borough of 500 000 people → yellow cabs barely serve it
- Unknown and N/A are **not boroughs**

Window functions

Aggregate without collapsing

```
SELECT trip, PULoc, fare,  
       avg(fare) OVER (PARTITION BY PULoc) AS zone_avg  
FROM trips;
```

trip	PULoc	fare	zone_avg
t1	132	70.00	67.75
t2	141	18.40	15.60
t3	132	65.50	67.75
t5	141	12.80	15.60

- GROUP BY would return 2 rows; this returns 4 corresponding to every trip, but with the group statistic attached
- Meaning: “this fare relative to its zone”
- Ordinary aggregate + OVER (...).

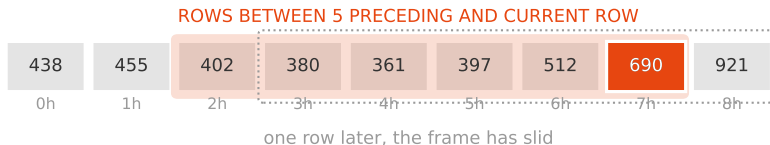
Structure of OVER

```
func(...) OVER (PARTITION BY <cols>      -- independent windows  
                ORDER BY    <cols>      -- order inside each window  
                ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) -- frame
```

- All of it is [standard] (SQL:2003)
- PARTITION BY → like GROUP BY, but the rows are kept
- ORDER BY → gives direction of the window: needed for ranking and running totals
- The **frame** says **which rows count** for the current row. With ORDER BY the default runs from the start of the partition to the current row — and it is RANGE, so **tied rows count together**
- All three omitted: the “window” is **the whole table**

The frame

default with ORDER BY: start of partition .. current row



- The frame **moves with the row**
- ROWS counts **rows**; the default with ORDER BY is RANGE, which counts **values**
- Another frame results in another value for the same PARTITION BY

Ranking, and QUALIFY

```
SELECT z.Borough, z.Zone, count(*) AS trips,
       row_number() OVER (PARTITION BY z.Borough
                          ORDER BY count(*) DESC) AS rk
FROM 'taxi_2025_01_sample.parquet' t
JOIN 'taxi_zone_lookup.csv' z ON t.PULocationID = z.LocationID
GROUP BY z.Borough, z.Zone
QUALIFY rk <= 2;           -- top-2 zone per borough
```

- row_number() 1,2,3...; rank() leaves gaps after ties; dense_rank() does not
- QUALIFY filters on a **window result** — what HAVING is to GROUP BY. [DuckDB] (also Snowflake, BigQuery, Teradata); elsewhere wrap it in a subquery and WHERE rk <= 2
- Manhattan: Midtown Center (9 855), Upper East Side South (9 425)

Running totals and moving averages

```
SELECT date_trunc('hour', tpep_pickup_datetime) AS hour,  
       count(*) AS trips,  
       sum(count(*)) OVER (ORDER BY hour) AS running_total,  
       avg(count(*)) OVER (ORDER BY hour  
                           ROWS BETWEEN 5 PRECEDING AND CURRENT ROW) AS moving_avg_6h  
FROM 'taxi_2025_01_sample.parquet' GROUP BY hour ORDER BY hour;
```

- Windows are applied **after** grouping → you can window over an aggregate
- Moving averages **smooth** a noisy series, standard for temporal series
- 2025-01-01: 438 trips at 00h, running total 893 by 01h, 1 735 by 04h

LAG and LEAD: look at the neighbours (*)

```
SELECT PULoc, pickup,  
       lag(pickup) OVER (PARTITION BY PULoc ORDER BY pickup) AS prev  
FROM trips;
```

PULoc	pickup	prev	gap
132	08:12	NULL	—
132	09:40	08:12	88 min
141	08:15	NULL	—
141	08:31	08:15	16 min

- lag = previous row, lead = next row → gaps, deltas, “time since last event”
- The first row of each partition has no previous row → NULL
- lead() looks into the future.

Sampling

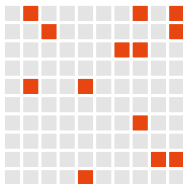
Why sample

- **Iteration speed**: a query you run 50 times while exploring data should be fast
- **Cost**: at 200 000 rows nobody cares; at 10^9 , this starts being felt
- **Class balance**: a 0.1 % positive class needs a sample built properly
- Your project's raw data will be sampled too

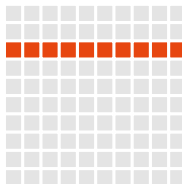
Sampling in DuckDB

```
SELECT * FROM trips USING SAMPLE 1% (bernoulli, 42);    -- seed
SELECT * FROM trips USING SAMPLE reservoir(5000 ROWS);  -- exactly 5000
```

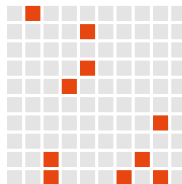
bernoulli 10 %
13 rows, scattered



system 10 %
10 rows, one block



reservoir(10 ROWS)
exactly 10 rows

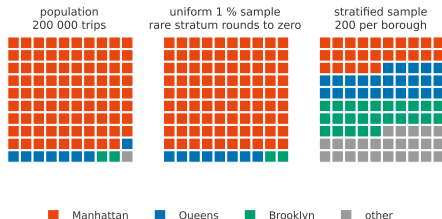


- `bernoulli` keeps each row with probability $p \rightarrow$ the count varies; `system` keeps whole blocks; `reservoir(n ROWS)` [DuckDB] fixes the count
- `TABLESAMPLE BERNOULLI (1) REPEATABLE (42)` is the [standard] version

```
SELECT * FROM trips USING SAMPLE 1% (bernoulli, 42);  -- 42 = the seed
```

- Same seed → same rows, every run, on every machine
- Unseeded sampling is an unreproducible dataset
- Our 200 000-row teaching file was itself cut with `reservoir(200000 ROWS) REPEATABLE (42)`
- Write the seed in your data card

Stratified sampling



```
-- feat table: trips joined to zones, one per trip (built in the lab)
SELECT * FROM feat
QUALIFY row_number() OVER (PARTITION BY Borough
                           ORDER BY random()) <= 200;
```

- A uniform sample is proportional to the population: at 1 %, a borough expecting 0.14 rows is **gone**
- **Stratified** = k rows **per stratum** → different distribution on purpose

Sampling pitfalls

- **Sample last, filter first:** sampling then filtering leaves you with an unknown, tiny, biased remainder
- **Sample once, then freeze it:** re-sampling between experiments makes two runs incomparable (keep seed + saved file)
- A sample answers questions about **the population it was drawn from** → January 2025 of yellow taxis is not “NYC transport data”
- Rare events are destroyed by sampling → count them *before* you decide

SQL vs pandas

SQL vs pandas

```
SELECT Borough, count(*) AS trips FROM trips t JOIN zones z
      ON t.PULocationID = z.LocationID
WHERE t.fare_amount > 0 GROUP BY Borough ORDER BY trips DESC;

(trips[trips.fare_amount > 0]
 .merge(zones, left_on="PULocationID", right_on="LocationID")
 .groupby("Borough").size()
 .sort_values(ascending=False))
```

- merge **is** pandas' join: on=/left_on= is the ON clause, how="inner"|"left"|"outer" picks the flavour (default: **inner**)
- Same operations, same result: SQL states the objective (declarative), pandas states the steps (imperative)
- pandas runs your steps **in the order you wrote them**; SQL is free to reorder to **optimize**

When to use which

SQL / DuckDB

- joins across several tables
- data larger than RAM
- heavy aggregation and windows
- the step is a **pipeline stage**

pandas

- interactive exploring a mid-size table
- plotting, scikit-learn, anything Python
- row-wise logic that is difficult to express in SQL

Interchangeable: `duckdb.sql(...).df()` moves between them, in most cases in-place.

Portable vs DuckDB-only (*)

[standard] portable	[DuckDB] ease of use
SELECT ... GROUP BY ... HAVING	file path as a table name
JOIN / LEFT / FULL	pandas DataFrame as a table
WHERE (NOT) EXISTS	SEMI JOIN / ANTI JOIN
WITH ... AS (CTEs)	QUALIFY
window functions + frames	USING SAMPLE, reservoir(n ROWS)
TABLESAMPLE ... REPEATABLE	date_trunc, hour(), median()

- Rule for project: if the query is a pipeline stage, prefer the portable spelling; for exploration, use the easiest / fastest to type

Lab and next week

Today's lab

- DuckDB on the taxi sample + zone lookup: joins into a **feature table**, an **anti-join** coverage check, a **window** function, **sampling** with a seed
- Same data as last week

Before next week

- M0 is due next week, 20:00 the evening before the lab
- A GitHub account is mandatory for everyone (need it for the API token)
- TMDB key / Twitch app with 2FA *if* you picked those sources
- Next week Web APIs: HTTP, REST, authentication, pagination, rate limits

References

- Codd, *A Relational Model of Data for Large Shared Data Banks*, CACM 13(6):377–387, 1970. <https://doi.org/10.1145/362384.362685>
- Stonebraker & Çetintemel, “One Size Fits All”: An Idea Whose Time Has Come and Gone, ICDE 2005 (OLTP vs OLAP engines).
https://cs.brown.edu/~ugur/fits_all.pdf
- Kimball & Ross, *The Data Warehouse Toolkit* (star schemas, dimensional modelling). <https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/books/data-warehouse-dw-toolkit/>
- DuckDB documentation: <https://duckdb.org/docs/> — sampling: <https://duckdb.org/docs/stable/sql/samples.html>
- DuckDB window functions: https://duckdb.org/docs/stable/sql/functions/window_functions
- NYC TLC Trip Record Data: <https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page>