Lab 2.1 — What Does My Query Actually Read?

Session 2.1 — Storage Formats & Distributed File Systems
Individual hands-on lab
Tools: DuckDB 1.x CLI or Python 3.11+ — parquet-tools (Rust) optional (Exercise 4)

---

1. Objective

Measure the concrete cost of file format choices. You will convert the same dataset into four formats, run identical queries on each, and observe how much data the engine actually reads. You will then deliberately break and restore predicate pushdown to understand what prevents it.

By the end you should be able to answer: *given a workload, which format minimises I/O — and why?*

2. Setup

2.1. Install

You can run all queries in this lab either with the DuckDB CLI or the Python API — pick whichever you prefer.

DuckDB CLI — one-liner installer:

```
curl https://install.duckdb.org | sh
duckdb          # interactive shell
```

Python API:

```
python3 -m venv .venv && source .venv/bin/activate
pip install duckdb
```

The SQL is identical in both. CLI users can prefix any query with `.mode line` to get readable output, and use `.quit` to exit.

2.2. Dataset

One month of NYC Yellow Taxi trips (50 MB Parquet, 3 million rows):

```
mkdir -p data
curl -L -o data/yellow_tripdata_2024-01.parquet \
  https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2024-01.parquet
```

Fallback if the URL is unavailable: replace `2024-01` with `2023-12` — the schema is identical.

2.3. Generating the comparison formats

```
COPY (SELECT * FROM 'data/yellow_tripdata_2024-01.parquet')
  TO 'data/trips.csv'             (FORMAT CSV, HEADER true);
COPY (SELECT * FROM 'data/yellow_tripdata_2024-01.parquet')
  TO 'data/trips.ndjson'          (FORMAT JSON);
COPY (SELECT * FROM 'data/yellow_tripdata_2024-01.parquet')
  TO 'data/trips_snappy.parquet'  (FORMAT PARQUET, COMPRESSION snappy);
COPY (SELECT * FROM 'data/yellow_tripdata_2024-01.parquet')
  TO 'data/trips_zstd.parquet'    (FORMAT PARQUET, COMPRESSION zstd);
```

Run these statements in the DuckDB CLI or wrap them in `con.execute(...)` calls in Python.

Columns used in the exercises: `fare_amount` (float), `payment_type` (integer), `trip_distance` (float), `tpep_pickup_datetime` (timestamp).

3. Exercise 1 — File sizes and compression ratio

Run `ls -lh data/trips.*` and compare the sizes of the four files.

3.1. Questions

1. Which format is largest? Why do field names repeated on every row make it worse than CSV?
2. Parquet Zstd is smaller than Parquet Snappy. What does Zstd trade to achieve that?
3. The original download (`yellow_tripdata_2024-01.parquet`) may be a different size from your `trips_snappy.parquet`. Why might two Parquet files with the same data differ in size?

4. Exercise 2 — Bytes read: projection pruning

We run a 2-column aggregation on a 19-column table.

4.1. Task

Run this query on each of the three formats and look for `Bytes Read` in the scan node:

```
EXPLAIN ANALYZE
SELECT payment_type, AVG(fare_amount)
FROM 'data/trips.csv'
GROUP BY payment_type ORDER BY payment_type;
```

Repeat with `'data/trips.ndjson'` and `'data/trips_snappy.parquet'` and compare the three plans.

4.2. Questions

1. CSV reads the whole file regardless of how many columns you select. Explain why.
2. Parquet reads a fraction of the file. Which file-format concept enables this?
3. If the table had 100 columns and your query used 2, how would the ratio change for each format?

5. Exercise 3 — Predicate pushdown

5.1. Part A — Pushdown working

Run this query and inspect the plan:

```
EXPLAIN ANALYZE
SELECT COUNT(*) FROM 'data/trips_snappy.parquet'
WHERE fare_amount > 50;
```

Look for `Filters:` and `Row Groups:` in the Parquet scan node. Note how many row groups are **read** vs **total**.

Before running: predict — out of 3 million rows, how many do you expect to have `fare_amount > 50`?

5.2. Part B — Breaking pushdown

Now wrap the column in a function and observe what changes:

```
EXPLAIN ANALYZE
SELECT COUNT(*) FROM 'data/trips_snappy.parquet'
WHERE ROUND(fare_amount, 0) > 50;
```

Compare the plan output for both queries: note whether `Filters:` appears in the scan node and how many row groups are read.

5.3. Part C — Other patterns that break pushdown

Test the following predicates. For each, record whether the filter is pushed into the scan:

```
predicates = [
    "fare_amount + 0 > 50",           # arithmetic on column
    "CAST(fare_amount AS INTEGER) > 50",  # cast
    "fare_amount > 50 AND fare_amount IS NOT NULL",  # compound
    "fare_amount BETWEEN 50 AND 200",  # range
]
```

5.4. Questions

1. Why can DuckDB skip row groups for `fare_amount > 50` but not for `ROUND(fare_amount, 0) > 50`?
2. What metadata does the Parquet file store that makes row-group skipping possible?
3. `BETWEEN 50 AND 200` — does it push? What two statistics does the engine need to decide whether a row group can be skipped?

6. Exercise 4 (bonus) — Parquet footer inspection

The Parquet footer is a binary Thrift blob appended to the file. Two tools let you read it without writing a query: `parquet-tools` (Rust, zero-runtime) and PyArrow (Python).

6.1. Option A — parquet-tools (Rust)

Install the CLI via Cargo (requires Rust — `curl https://sh.rustup.rs | sh` if absent):

```
cargo install parquet
```

Then inspect the file:

```
# Summary: row-group count, total rows, compression, encodings
parquet meta data/trips_snappy.parquet

# Per-column statistics (min/max/null count) for every row group
parquet row-group-meta data/trips_snappy.parquet
```

6.2. Option B — PyArrow

```
pip install pyarrow
```

```
import pyarrow.parquet as pq

meta = pq.read_metadata("data/trips_snappy.parquet")
print(f"Row groups : {meta.num_row_groups}")
print(f"Total rows : {meta.num_rows:,}")

rg = meta.row_group(0)
for i in range(rg.num_columns):
    col = rg.column(i)
    if "fare" in col.path_in_schema:
        stats = col.statistics
        print(f"Column     : {col.path_in_schema}")
        print(f"Min / Max  : {stats.min} / {stats.max}")
        print(f"Null count : {stats.null_count}")
        print(f"Encodings  : {col.encodings}")
```

6.3. Questions

1. How many row groups does the file have, and roughly how many rows per group?
2. What is the global max of `fare_amount`? Does it match the anomalous fares NYC taxi data is known for?
3. Is `fare_amount` dictionary-encoded? Why or why not (think about cardinality)?
4. Find a column that **is** dictionary-encoded. Which one, and why does low-cardinality favour dictionary encoding?
5. The footer is written **after** the data columns. What does this mean for a reader that wants only the statistics — must it read the whole file?

7. Key Takeaways

- Columnar formats (Parquet) read only the columns your query uses. Row formats (CSV, NDJSON) always scan every byte.
- Predicate pushdown uses per-column statistics stored in the footer to skip entire row groups **before reading any data values**.
- Pushdown is silently disabled by any transformation on the filtered column. The engine cannot infer the output range of an arbitrary function.
- File size and I/O cost are related but not identical: Zstd compresses better than Snappy but may decompress slower — the right choice depends on your CPU/IO balance.