# Lab 3.2 — ClickHouse from the Inside Out

Session 3.2 — ClickHouse: Real-Time Analytics at Scale
Guided lab
**Tools:** `clickhouse local`, NYC Taxi dataset (Parquet)

---

## 1. Objective

Understand **why** ClickHouse is fast by observing its physical behavior — not just its query times. Each step has a concrete, measurable output: a file listing, a compression ratio, a row count in `system.query_log`. Every claim from the session slides has a corresponding experiment here.

## 2. Setup

### 2.1. Install ClickHouse

ClickHouse ships as a single static binary. No Docker, no server, no configuration:

```
curl https://clickhouse.com/ | sh
```

Start an interactive session with a persistent on-disk database:

```
./clickhouse local --path ./ch-lab
```

The `--path` flag tells ClickHouse to write all data to `./ch-lab/`. Everything you create persists across sessions. Open a second terminal alongside — you will browse the file system while queries run.

### 2.2. Dataset

Download one month of NYC Taxi trip data in Parquet format:

```
wget https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2024-01.parquet
```

The file is 60 MB compressed, 2.9 million rows. It is enough to make storage structure observable without long load times.

## 3. Walkthrough

### 3.1. Step 1 — Query before ingesting

**Goal:** establish that ClickHouse can read Parquet files directly, without any schema declaration.

Run this query against the raw file:

```
SELECT
    toStartOfHour(tpep_pickup_datetime) AS hour,
```

```
    count()                             AS trips,
    round(avg(fare_amount), 2)          AS avg_fare,
    round(sum(fare_amount), 0)          AS total_revenue
FROM file('yellow_tripdata_2024-01.parquet')
GROUP BY hour
ORDER BY hour;
```

Then inspect what ClickHouse read:

```
SELECT read_rows, read_bytes, query_duration_ms
FROM system.query_log
WHERE type = 'QueryFinish'
ORDER BY event_time DESC
LIMIT 1;
```

**Observe:** `read_rows` is the full row count — no pruning is possible without an index. Note the duration. This is the baseline you will beat in later steps.

**Note:** ClickHouse inferred the schema directly from the Parquet metadata. You can inspect what it found with:

```
DESCRIBE file('yellow_tripdata_2024-01.parquet');
```

## 3.2. Step 2 — Ingest into MergeTree and inspect the file system

**Goal:** load the data into a real MergeTree table and observe what ClickHouse writes to disk.

```
CREATE TABLE trips (
    pickup_at    DateTime,
    dropoff_at   DateTime,
    vendor_id    UInt8,
    passengers   UInt8,
    distance     Float32,
    fare         Float32,
    tip          Float32,
    total        Float32,
    payment_type Enum8('card'=1, 'cash'=2, 'no_charge'=3, 'dispute'=4, 'unknown'=5)
) ENGINE = MergeTree
ORDER BY (pickup_at, vendor_id);

INSERT INTO trips
SELECT
    tpep_pickup_datetime,
    tpep_dropoff_datetime,
    vendor_id,
    passenger_count,
    trip_distance,
    fare_amount,
    tip_amount,
    total_amount,
    payment_type_id
FROM file('yellow_tripdata_2024-01.parquet')
WHERE fare_amount > 0 AND trip_distance > 0;
```

While the insert runs, switch to your second terminal:

```
ls ./ch-lab/data/default/trips/
```

You will see one or more part directories appear, named like `20240101_1_1_0`. After the insert completes:

```
ls ./ch-lab/data/default/trips/20240101_1_1_0/
```

**Observe and identify each file:**

| File | Contents |
|---|---|
| `pickup_at.bin` | Compressed column data for `pickup_at` — this is the only file a date-range query reads |
| `pickup_at.mrk3` | Mark file: maps each granule index entry to a byte offset in `pickup_at.bin` |
| `primary.idx` | Sparse primary index: one entry per  8 192 rows, always kept in RAM |
| `count.txt` | Total row count of this part |
| `columns.txt` | Column names and types stored in this part |
| `checksums.txt` | Per-file checksums for integrity verification |

Compare the size of `primary.idx` against `pickup_at.bin`:

```
ls -lh ./ch-lab/data/default/trips/20240101_1_1_0/primary.idx
ls -lh ./ch-lab/data/default/trips/20240101_1_1_0/pickup_at.bin
```

**Question:** the index is kilobytes; the data file is megabytes. Why is this called a "sparse" index? What does ClickHouse sacrifice to keep it this small?

#### 3.2.1. Merging parts

If multiple parts were created (one per insert batch), trigger a manual merge:

```
OPTIMIZE TABLE trips FINAL;
```

Watch the parts directory in your second terminal — subdirectories consolidate into one. Then confirm via:

```
SELECT name, rows, parts, bytes_on_disk
FROM system.tables
WHERE name = 'trips';
```

## 3.3. Step 3 — Per-column compression ratios

**Goal:** measure how well each column compresses and explain why, based on the data's statistical properties.

**Before running:** predict the ranking. Which column will compress best? Which worst? Write your prediction down.

```
SELECT
    column,
    data_compressed_bytes,
    data_uncompressed_bytes,
    round(data_uncompressed_bytes / data_compressed_bytes, 1) AS ratio
FROM system.columns
WHERE table = 'trips'
ORDER BY ratio DESC;
```

Compare your prediction against the result.

**Guidance for interpretation:**

- `pickup_at` — timestamps stored in sort order are nearly monotone. The Delta codec encodes differences between consecutive values, then ZSTD compresses near-zero deltas extremely well.
- `vendor_id` — only 2–3 distinct values in the entire column. A byte that repeats 2.9 million times compresses to almost nothing.

- `payment_type` — similarly low-cardinality; stored as an `Enum8` (one byte), high ratio expected.
- `distance`, `fare`, `tip` — continuous floats. Adjacent values have no predictable relationship. LZ4 compresses them modestly but not dramatically.
- `passengers` — small integer, low cardinality, good ratio.

**Experiment:** alter the `fare` column to use the Gorilla codec (designed for slowly-changing floats) and compare:

```
ALTER TABLE trips MODIFY COLUMN fare Float32 CODEC(Gorilla, ZSTD);
OPTIMIZE TABLE trips FINAL;  -- force rewrite

SELECT column, round(data_uncompressed_bytes / data_compressed_bytes, 1) AS ratio
FROM system.columns
WHERE table = 'trips' AND column = 'fare';
```

Does Gorilla improve the ratio for taxi fares? Why or why not? (Hint: Gorilla performs best when consecutive values are close — is that true for fares?)

## 3.4. Step 4 — The ORDER BY experiment

**Goal:** prove empirically that `ORDER BY` is the primary index, not a cosmetic sort preference.

Create the same table with the sort key reversed:

```
CREATE TABLE trips_vendor_first (
    pickup_at    DateTime,
    dropoff_at   DateTime,
    vendor_id    UInt8,
    passengers   UInt8,
    distance     Float32,
    fare         Float32,
    tip          Float32,
    total        Float32,
    payment_type Enum8('card'=1, 'cash'=2, 'no_charge'=3, 'dispute'=4, 'unknown'=5)
) ENGINE = MergeTree
ORDER BY (vendor_id, pickup_at);

INSERT INTO trips_vendor_first SELECT * FROM trips;
```

Now run the same date-range query on both tables:

```
SELECT sum(fare)
FROM trips
WHERE pickup_at >= '2024-01-10' AND pickup_at < '2024-01-11';

SELECT sum(fare)
FROM trips_vendor_first
WHERE pickup_at >= '2024-01-10' AND pickup_at < '2024-01-11';
```

Read the evidence from `system.query_log`:

```
SELECT
    left(query, 60)    AS q,
    read_rows,
    read_bytes,
    query_duration_ms
FROM system.query_log
WHERE type = 'QueryFinish' AND query LIKE '%sum(fare)%'
ORDER BY event_time DESC
LIMIT 4;
```

**Observe:** `read_rows` on `trips` should be a small fraction of `read_rows` on `trips_vendor_first`. The granule skipping mechanism, visible through `EXPLAIN indexes = 1`, is doing the work:

```
EXPLAIN indexes = 1
SELECT sum(fare) FROM trips
WHERE pickup_at >= '2024-01-10' AND pickup_at < '2024-01-11';
```

Find the line reporting `selected marks` — that is the number of granules (each  8 192 rows) that ClickHouse had to read.

**Flip the query:** now run `WHERE vendor_id = 2` on both tables. Which is faster? Predict before running, then verify.

## 3.5. Step 5 — Materialized views

**Goal:** build an incrementally updated summary table and verify its correctness with a known-value batch.

### 3.5.1. Create the target table and view

```
CREATE TABLE trips_hourly (
    hour    DateTime,
    trips   AggregateFunction(count, UInt8),
    revenue AggregateFunction(sum,   Float32),
    avg_tip AggregateFunction(avg,   Float32)
) ENGINE = AggregatingMergeTree
ORDER BY hour;

CREATE MATERIALIZED VIEW mv_trips TO trips_hourly AS
SELECT
    toStartOfHour(pickup_at) AS hour,
    countState()             AS trips,
    sumState(fare)           AS revenue,
    avgState(tip)            AS avg_tip
FROM trips;
```

The materialized view fires on every `INSERT INTO trips` — it sees only the new batch, not the full table. The `*State` functions store intermediate binary state that the `*Merge` functions combine at query time.

### 3.5.2. Verify with a known batch

Insert three rows with known values into the same hour:

```
INSERT INTO trips VALUES
    ('2024-01-20 09:05:00', '2024-01-20 09:25:00', 1, 2, 5.0, 15.00, 3.00, 19.00,
'card'),
    ('2024-01-20 09:30:00', '2024-01-20 09:50:00', 1, 1, 3.0, 12.00, 2.00, 15.00,
'card'),
    ('2024-01-20 09:45:00', '2024-01-20 10:05:00', 2, 3, 7.0, 18.00, 0.00, 19.00,
'cash');
```

The expected result for `hour = 2024-01-20 09:00:00`:

- 3 trips
- Total fare: 15.00 + 12.00 + 18.00 = **45.00**
- Average tip: (3.00 + 2.00 + 0.00) / 3 = **1.67**

Query the materialized view and verify:

```
SELECT
    hour,
```

```
    countMerge(trips)            AS trip_count,
    round(sumMerge(revenue), 2)  AS total_fare,
    round(avgMerge(avg_tip), 2)  AS mean_tip
FROM trips_hourly
WHERE hour = '2024-01-20 09:00:00'
GROUP BY hour;
```

**The numbers must match your mental arithmetic.** If they don't, the view is incorrect — investigate why before continuing.

#### 3.5.3. Inspect the view's storage on disk

The target table `trips_hourly` is a real MergeTree table with its own parts:

```
ls ./ch-lab/data/default/trips_hourly/
```

Compare query time and `read_rows` between the materialized view and the full source scan:

```
-- Via materialized view
SELECT hour, countMerge(trips), round(sumMerge(revenue), 2)
FROM trips_hourly GROUP BY hour ORDER BY hour;

-- Via full scan
SELECT toStartOfHour(pickup_at) AS hour, count(), round(sum(fare), 2)
FROM trips GROUP BY hour ORDER BY hour;
```

**Question:** why is `avgState`/`avgMerge` safe across partial batches when plain `AVG` would not be? (Hint: what intermediate state does `avgState` store — and what would a naive average of averages lose?)

## 3.6. Step 6 — Schema design under adversarial queries

**Goal:** reason about schema trade-offs before measuring, then validate your reasoning.

You have the `trips` table sorted by `(pickup_at, vendor_id)`. Three analytical questions arrive:

| Query | Pattern |
|---|---|
| Q1 | `SELECT sum(fare) FROM trips WHERE pickup_at >= … AND pickup_at < …` — date-range aggregation |
| Q2 | `SELECT avg(tip/fare) FROM trips WHERE payment_type = 'card'` — filter on non-leading column |
| Q3 | `SELECT count(), avg(distance) FROM trips WHERE vendor_id = 2` — filter on second ORDER BY column |

Run all three, then read `system.query_log` for `read_rows` and `query_duration_ms`.

#### 3.6.1. Design challenge

You must serve all three queries from a single table. You may:

- Change `ORDER BY` (at most two columns)
- Add up to two skip indexes

You may **not** create a separate table or a materialized view.

**Before writing any SQL:** fill in this analysis.

| Query | Current bottleneck | What would help |
|---|---|---|
| Q1 | | |

| | | |
|---|---|---|
| Q2 | | |
| Q3 | | |

### 3.6.2. Guided trade-off analysis

Consider the following four designs and their consequences:

**Design A — keep `ORDER BY (pickup_at, vendor_id)`**

Q1 benefits maximally from granule skipping. Q3 can skip on `vendor_id` only after the date prefix is resolved — partial help. Q2 gets no help from the primary index: `payment_type` is not in `ORDER BY`. To help Q2, add a `set(4)` skip index on `payment_type` (only 4 distinct values — a set index is exact, no false positives).

```
ALTER TABLE trips ADD INDEX idx_payment payment_type TYPE set(4) GRANULARITY 1;
ALTER TABLE trips MATERIALIZE INDEX idx_payment;
```

**Design B — change `ORDER BY (payment_type, pickup_at)`**

Q2 now benefits from granule skipping: all `card` rows are physically adjacent. Q1 now scans everything — date ranges cross all payment types. Q3 is unaffected positively. This design optimises the slowest query at the cost of the fastest one.

**Design C — change `ORDER BY (vendor_id, pickup_at)`**

Q3 benefits maximally. Q1 is slow. Q2 is unaffected. Only sensible if vendor-level reporting is the dominant workload.

**Design D — keep `ORDER BY (pickup_at, vendor_id)` + add a projection for vendor**

```
ALTER TABLE trips ADD PROJECTION proj_vendor (
    SELECT * ORDER BY (vendor_id, pickup_at)
);
ALTER TABLE trips MATERIALIZE PROJECTION proj_vendor;
```

Q1 and Q3 both get fast paths: Q1 from the main sort, Q3 from the projection. Q2 still needs the skip index. Storage grows by roughly the compressed size of the table (the projection stores all columns in a different sort order).

### 3.6.3. Implement and measure

Choose the design you think is best (or invent your own), implement it, re-run the three queries, and compare `read_rows` before and after:

```
SELECT
    left(query, 80) AS q,
    read_rows,
    query_duration_ms
FROM system.query_log
WHERE type = 'QueryFinish'
  AND query LIKE '%FROM trips%'
ORDER BY event_time DESC
LIMIT 12;
```

**Write a short justification** (3–5 sentences): which design did you choose, which query did you sacrifice if any, and why is that trade-off acceptable given a realistic workload where date-range queries run 10× more often than vendor-specific ones?

## 4. Key Takeaways

- `file()` makes ClickHouse a universal Parquet reader — no ingestion required for exploration
- Parts are directories of per-column files; `primary.idx` is kilobytes even for millions of rows because it stores one entry per granule, not one per row
- Compression ratio varies sharply by column: monotone sequences (timestamps, enums) compress 10–30×; continuous floats compress 2–4×
- `ORDER BY` is the primary index — the single most impactful schema decision, not a cosmetic sort
- `AggregatingMergeTree` + `*State`/`*Merge` is the correct pattern for materialized views: intermediate state is combinable across partial batches; plain `AVG` is not
- No `ORDER BY` satisfies every query equally; the engineering answer is to choose the dominant workload and use projections or skip indexes to cover the others