ClickHouse Course

Flavio Copes

Updated August 3, 2026

Contents

Learn ClickHouse for analytical workloads: columnar storage, MergeTree design, ordering keys, batch ingestion, materialized views, operations, and recovery.

What you'll learn: Design, load, query, secure, monitor, and recover a ClickHouse event pipeline using measured analytical access patterns.

Analytical foundations

OLAP workloads, columnar execution, local setup, Cloud, clients, SQL, and data types.

Choose an analytical database

Recognize event and aggregate workloads that fit ClickHouse and keep transactional application state in an OLTP database.

ClickHouse is a column-oriented analytical database. It is designed to scan, filter, and aggregate large collections of events quickly.

The database world splits along a workload line. **OLTP** (online transaction processing) means many small reads and writes that must be exact right now: create an order, update a balance, load one user's profile. **OLAP** (online analytical processing) means questions over lots of history: how many requests failed per hour last month, which pages grew fastest this quarter.

A query like this is ClickHouse's home turf:

```
SELECT toStartOfDay(ts) AS day, count() AS pageviews
FROM events
WHERE ts >= now() - INTERVAL 30 DAY
GROUP BY day
ORDER BY day;
```

Scanning a billion events to answer that takes ClickHouse seconds. The same query can bring a busy PostgreSQL instance to its knees, because a row store must read every column of every row it touches.

Why not use it for everything

It is not the default home for a shopping cart transaction or one user profile update. ClickHouse trades away the things OLTP needs: updating or deleting individual rows is an expensive background operation, and there are no classic multi-statement transactions to keep an order and its payment consistent.

Keep transactional invariants in PostgreSQL, MySQL, or another OLTP database, then send analytical events to ClickHouse when the workload justifies it.

Real products are built exactly this way. Plausible Analytics, the privacy-friendly web analytics tool, runs both databases side by side: PostgreSQL holds user accounts and site settings, ClickHouse holds the pageview event stream that powers every dashboard chart.

"When the workload justifies it" is worth taking seriously. Tens of millions of rows with occasional reporting queries? PostgreSQL handles that fine. ClickHouse earns its place when event volume or query latency makes the row store visibly struggle.

Classify before you build

Take these four workloads and decide where each lives: an account balance, an application log stream, a product dashboard, and a user session.

The balance is transactional state — OLTP, no discussion. The log stream is append-only events at high volume — ClickHouse. The dashboard reads aggregates over history — ClickHouse. The session is the interesting one: the *live* session your app checks on every request is OLTP state, while the *record* of past sessions you analyze for behavior is an event stream for ClickHouse.

For each dataset, name the system of record and, separately, the analytical copy. Events flow one way, from the source of truth into ClickHouse — never back.

Run ClickHouse locally

Start a disposable server and client with an official package or container, then keep it bound and removable for the lab.

Use an official package or container to run ClickHouse locally. Connect with `clickhouse-client` or the local command supported by your installation.

Keep the first server on the development machine. Do not expose an unauthenticated database port to the internet. A ClickHouse Cloud service is another option when you want managed storage and operations.

Run `SELECT version()`, create a practice database, and record the exact command that stops and removes the local environment.

Run a disposable server and connect with the native client:

```
docker run --rm -d --name clickhouse \
  -p 8123:8123 -p 9000:9000 clickhouse/clickhouse-server
docker exec -it clickhouse clickhouse-client
```

Inside the client, run SELECT version() and create a separate practice database. Stop the container when finished. Do not publish the native or HTTP ports on an internet-facing host without authentication and network controls.

Understand columnar storage

See why reading selected columns and processing compressed vectors suits analytical queries over wide event tables.

A row database stores values from one record together. A columnar database stores values from the same column together inside data parts.

Picture a pageview events table with twenty columns: timestamp, URL, referrer, country, browser, and so on. In a row store, one pageview's twenty values sit side by side on disk. In ClickHouse, all the timestamps sit together in one file, all the URLs in another, all the countries in a third.

Why the layout wins for analytics

An analytical query often reads a few columns from many rows:

```
SELECT country, count() AS views
FROM pageviews
WHERE ts >= '2026-07-01'
GROUP BY country;
```

This query needs `ts` and `country`. Nothing else. ClickHouse can skip unrelated columns entirely — the URLs, referrers, and seventeen other columns are never read from disk. A row store has no such option: the values are interleaved, so it drags every column of every row through memory to use two of them.

The second win is **compression**. A column file contains millions of values of the same type, and similar values sitting together compress extremely well. A country column is thousands of repeats of a few dozen strings; it might shrink fifty-fold. Smaller files mean less disk I/O, which is what analytical queries spend most of their time on.

The third win is **vectorized execution**. Because values arrive as long same-typed arrays, Click-

House processes them in batches using CPU instructions that operate on many values at once, instead of interpreting one row at a time.

The cost of the layout

The same layout makes row-oriented work expensive. Fetching one complete event means visiting twenty separate column files and reassembling the row. That inverts the row-store trade-off, and it's why the previous lesson told you to keep OLTP state elsewhere.

Try it on paper

Create a wide events table on paper — say fifteen columns for a web analytics event. Mark which columns a daily count query reads and which columns ClickHouse can avoid.

Then do the same for "show me everything about event X". You'll see the first query touches two or three files out of fifteen, while the second touches all of them. That ratio — columns read versus columns stored — is the single best predictor of whether a query will fly or crawl on columnar storage, and it's the lens to keep for the whole course.

Check your understanding: analytical foundations

Check the key models, decisions, commands, security boundaries, and failure cases from analytical foundations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Schema and MergeTree

Types, MergeTree parts, ORDER BY, primary indexes, partitions, skipping, and denormalized models.

Create a MergeTree table

Choose explicit types and create the central ClickHouse table engine used for scalable analytical storage.

Every ClickHouse table declares an **engine**, and most general ClickHouse tables use the MergeTree family. Inserts create immutable parts, and background merges combine parts over time.

That sentence is the whole storage model, so let's slow down on it. Each insert writes a new **part**: a self-contained directory of column files, sorted and never modified again. A background process later merges small parts into bigger ones. Reads see the current set of parts; writes never block them.

Here's an events table for a service that tracks API requests:

```
CREATE TABLE events (
  ts          DateTime,
  service     LowCardinality(String),
  user_id     UInt64,
  event_type  LowCardinality(String),
```

```
  duration_ms UInt32,
  metadata    String
)
ENGINE = MergeTree
ORDER BY (service, ts);
```

`ORDER BY` defines how rows are sorted inside each part. It's required, and choosing it well is the single biggest schema decision — big enough that it gets its own lesson next.

Types are a performance decision

Choose narrow accurate types, use dates and timestamps deliberately, and avoid nullable values when a clear default or separate state fits. Schema choices affect compression and query work.

`UInt32` for a duration in milliseconds beats `UInt64` — half the bytes to scan for a value that never needs the range. `LowCardinality(String)` is the right wrapper for columns like `service` or `event_type` that repeat a small set of values; ClickHouse dictionary-encodes them and both storage and `GROUP BY` get faster. And `Nullable(String)` adds a hidden mask column to every read — an empty string default is usually the better call.

Insert and verify

Insert five realistic rows:

```
INSERT INTO events VALUES
  ('2026-08-03 10:00:01', 'api',     101, 'request',  42, '{"path":"/v1/users"}'),
  ('2026-08-03 10:00:02', 'api',     102, 'request',  55, '{"path":"/v1/orders"}'),
  ('2026-08-03 10:00:02', 'checkout', 101, 'payment', 310, '{"amount":49}'),
  ('2026-08-03 10:00:03', 'api',     103, 'request',  38, '{"path":"/v1/users"}'),
  ('2026-08-03 10:00:05', 'checkout', 104, 'payment', 290, '{"amount":19}');
```

Now look at the storage that insert created:

```
SELECT name, rows, active FROM system.parts
WHERE table = 'events';
-- all_1_1_0   5   1
```

One insert, one part, five rows. Every part your table ever creates shows up in `system.parts`, and checking it becomes second nature when you debug ingestion later.

One expectation to correct early: there's no cheap `UPDATE` here. Parts are immutable, so changing a row means rewriting parts. If you catch yourself designing a MergeTree table around modifying rows, the design is wrong — model events as append-only facts instead.

Choose the ordering key

Design ORDER BY from frequent filters and cardinality because physical row order drives data skipping and compression.

In ClickHouse, `ORDER BY` defines how rows are sorted inside each part. The sparse primary index uses that order to skip granules.

Put frequently filtered, lower-cardinality dimensions early when they eliminate useful ranges, then time and other fields according to query patterns. This is not the same decision as a PostgreSQL uniqueness constraint.

Compare ordering events by timestamp alone with ordering by service, event type, and timestamp. List the queries each layout can skip efficiently.

Create two disposable tables with the same rows but different orderings:

```
create table by_time (service LowCardinality(String), ts DateTime)
engine = MergeTree order by ts;

create table by_service (service LowCardinality(String), ts DateTime)
engine = MergeTree order by (service, ts);
```

Load enough data to span many granules. Filter one service over a time range and compare read rows and bytes. The result turns ORDER BY from a naming rule into a measured physical-design decision.

Partition and model deliberately

Use coarse partitions for lifecycle operations, denormalize common dimensions, and avoid creating thousands of small partitions or runtime joins.

Partitions group parts for operations such as dropping old data. They are not a replacement for the ordering key.

This distinction trips up almost everyone coming from other databases. The ordering key is what makes queries fast. A partition is a lifecycle unit: a chunk of the table you can drop, move, or detach as one cheap operation.

Monthly partitions are common for time data:

```
CREATE TABLE events (
  ts      DateTime,
  service LowCardinality(String),
  user_id UInt64
)
ENGINE = MergeTree
PARTITION BY toYYYYMM(ts)
ORDER BY (service, ts);
```

Now a retention policy is one statement:

```
ALTER TABLE events DROP PARTITION '202507';
```

Dropping a partition deletes its parts instantly, with none of the cost of deleting rows one by one.

The too-many-partitions failure

High-cardinality partition keys create too many parts. Partition by day and keep three years of data: 1,095 partitions. Partition by `user_id`: potentially millions. Every partition holds its own parts that can never merge across partition boundaries, so part counts explode, inserts touching many partitions slow down, and eventually inserts fail with a `Too many parts` error.

The symptom shows up in `system.parts`:

```
SELECT partition, count() AS parts
FROM system.parts
WHERE table = 'events' AND active
```

```
GROUP BY partition
ORDER BY parts DESC;
```

Hundreds of partitions with a handful of tiny parts each means the partition key is too fine. My advice: partition by month, or don't partition at all. You need a reason — usually retention — to partition, not a reason to skip it.

Denormalize the hot path

Analytical tables often denormalize dimensions used in every query. In a normalized OLTP schema you'd store `service_id` and join to a services table. Here, if every dashboard query filters or groups by service name, copy the name into the events table.

Dictionaries or joins can still help — a dimension that changes often, or one used rarely, can stay external. But copying stable labels into events may make the hot path simpler: no join to plan, no second table to keep available, and `LowCardinality(String)` makes the repeated values nearly free to store.

Try it

Choose a partition, ordering key, and retention operation for one year of application events. Estimate the number of partitions before creating the table.

Monthly gives you 12; daily gives you 365. Say the retention rule out loud — "each month, drop the partition from 13 months ago" — and check the operation matches the partition unit. If your retention is expressed in days but your partitions are months, one of the two needs to change.

Check your understanding: schema and mergetree

Check the key models, decisions, commands, security boundaries, and failure cases from schema and mergetree.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Ingestion and data flow

Batch inserts, asynchronous inserts, retries, deduplication, imports, and event-pipeline design.

Insert in batches

Send blocks of rows instead of one synchronous insert per event so ClickHouse creates healthy parts and uses resources efficiently.

ClickHouse performs best when clients insert batches. One tiny insert per event creates many small parts and too much merge work.

The reason follows from the storage model. Every `INSERT` creates at least one new part on disk — a directory with a file per column. A part holding one row costs almost as much bookkeeping as a part holding 100,000 rows. Insert row by row at even a modest event rate and you're creating thousands of parts per minute while the background merges fall further and further behind.

ClickHouse defends itself when that happens. Inserts start failing with:

```
DB::Exception: Too many parts (300 with average size of 2.31 KiB) in table 'analytics.events'
Merges are processing significantly slower than inserts.
```

If you see this error, don't reach for the setting that raises the limit. The error is the database telling you your inserts are shaped wrong.

Batch at the source

Buffer events in the application, agent, or queue, then send a block in a supported format:

```
cat events.jsonl | clickhouse-client --query \
  "INSERT INTO analytics.events FORMAT JSONEachRow"
```

A good starting shape is batches of 10,000 to 100,000 rows. One insert with 50,000 rows creates one healthy part; 50,000 single-row inserts create 50,000 parts and a merge storm.

Bound the buffer by count and time so low traffic still arrives. The rule is ”flush at 50,000 rows or after 5 seconds, whichever comes first”. Without the time bound, a quiet service's events sit in the buffer indefinitely; without the count bound, a traffic spike balloons memory.

Measure the difference yourself

Insert the same thousand events as one batch and as many tiny inserts in a disposable table. Compare elapsed time and part counts:

```
SELECT count() AS parts, sum(rows) AS rows
FROM system.parts
WHERE table = 'events_test' AND active;
```

The batched version finishes in a fraction of the time and shows one part. The row-by-row version shows hundreds of parts for identical data — each one waiting for a merge that the batch never needed.

One caveat before you build elaborate buffering into every producer: if your writers are many and small — say, serverless functions that each see a few events — client-side batching gets awkward. That's the situation server-side asynchronous inserts were built for, and it's exactly where the next lesson picks up.

Use asynchronous inserts carefully

Let the server buffer small client writes while understanding acknowledgement, deduplication, memory, and failure tradeoffs.

Asynchronous inserts can buffer small writes on the server and flush them as larger blocks. They help when changing every producer is difficult.

The previous lesson said to batch on the client. Sometimes you can't: hundreds of short-lived processes, edge functions, or third-party agents each deliver a handful of rows. Async inserts move the batching into ClickHouse itself — small inserts land in an in-memory buffer, and the server writes one healthy part when the buffer fills or a timeout passes.

You enable it per query or per user with settings:

```
INSERT INTO analytics.events
SETTINGS async_insert = 1, wait_for_async_insert = 1
VALUES ('2026-08-03 10:00:01', 'api', 101, 'request', 42);
```

The server collects these small inserts and flushes them together, controlled by thresholds like `async_insert_max_data_size` and `async_insert_busy_timeout_ms`.

The acknowledgement decision

Choose whether the client waits for the flush. That's what `wait_for_async_insert` controls, and it's the most consequential setting here.

With `wait_for_async_insert = 1`, the insert returns only after the buffer is flushed to storage. A success response means your data is durable. Throughput is still better than raw small inserts, because many clients share each flush.

With `wait_for_async_insert = 0`, the insert returns as soon as the data enters the buffer. Acknowledging before durable insertion changes failure semantics: if the server crashes before the flush, or the buffered data turns out to be malformed, your client already got a success response for rows that never landed. The error surfaces later, in server logs nobody is watching.

My advice is to keep `wait_for_async_insert = 1` unless you've measured that you can't afford it and you can tolerate losing acknowledged rows.

Retries and duplicates

Retries need an explicit deduplication strategy, especially when materialized views transform inserted data. A client that times out and resends may deliver the same rows twice, and with async inserts the automatic block-level deduplication you might rely on for synchronous inserts behaves differently — buffered blocks are recombined, so identical-payload detection is no longer a guarantee you can lean on.

Verify what's actually flowing through the buffer:

```
SELECT status, count()
FROM system.asynchronous_insert_log
WHERE event_time > now() - INTERVAL 1 HOUR
GROUP BY status;
```

`Ok` rows flushed cleanly; `ParsingError` or `FlushError` rows are the writes your fire-and-forget clients think succeeded.

Configure async inserts in a lab, stop the server at different moments, and document which acknowledged writes survive. Do this before production traffic depends on the answer — with `wait_for_async_insert = 0` you will lose the buffered tail, and it's much better to learn that from an experiment.

Design an idempotent event pipeline

Give events stable identities, handle retries and malformed batches, preserve source timestamps, and monitor ingestion lag.

Event delivery can repeat. A producer times out and resends. A queue redelivers after a consumer crash mid-batch. A deploy replays an hour of traffic. None of these are rare — at-least-once delivery is the honest contract of almost every pipeline, so duplicates are a design input, not an edge case.

Idempotent means processing the same event twice leaves the same result as processing it once. Your dashboards' credibility depends on it: a retry storm that double-counts revenue is a very visible bug.

Stable identity

Add a stable event ID or another deduplication boundary before retrying the same batch:

```
CREATE TABLE events (
  event_id   UUID,
  ts         DateTime,
  service    LowCardinality(String),
  event_type LowCardinality(String)
)
ENGINE = MergeTree
ORDER BY (service, ts);
```

The ID must be minted where the event happens — the producer — not where it's stored. An ID assigned at insert time makes two copies of the same event look like two different events.

With identity in place you can layer defenses. Resending a batch with the same `insert_deduplication_token` setting lets ClickHouse drop the repeat delivery. And whatever slips through remains detectable, and cleanable, because duplicates share an `event_id`:

```
SELECT event_id, count() AS copies
FROM events
GROUP BY event_id
HAVING copies > 1;
```

Two timestamps, not one

Keep event time separate from ingestion time. The moment a user clicked and the moment the row reached ClickHouse can differ by seconds or, after an outage, by hours. Store both — `ts` from the source and an `ingested_at DateTime DEFAULT now()`. Charts group by event time; the gap between the two is your **ingestion lag**, and watching its maximum tells you when the pipeline is falling behind.

Reject loudly, not silently

Validate required fields at the producer or staging boundary, retain failed batches for inspection, and avoid silently replacing malformed values with misleading defaults. A parser that turns a broken timestamp into `1970-01-01` doesn't fix bad data; it hides it inside your charts where nobody can find it. Route rejects to a **dead-letter** path — a file, a queue topic, a separate table — so they can be inspected and replayed after the producer bug is fixed.

Sketch the pipeline

Sketch producer, queue, ClickHouse, and dead-letter paths. Name who retries, how duplicates are detected, and how ingestion lag is measured.

If any of those three questions has no owner, that's the hole an incident will find. The pipelines that survive messy weeks are the ones where every arrow in the sketch has an answer for "what happens when this step runs twice?"

Check your understanding: ingestion and data flow

Check the key models, decisions, commands, security boundaries, and failure cases from ingestion and data flow.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Queries and acceleration

Aggregations, query plans, system tables, projections, materialized views, and measured optimization.

Write analytical queries

Group, aggregate, filter, and calculate time windows while selecting only required columns and preserving meaningful units.

ClickHouse speaks SQL, but analytical queries often scan long time ranges and aggregate millions of rows. Select only the columns you need and filter on fields supported by the ordering key. `SELECT *` on a columnar store throws away its main advantage — every column you name is a file ClickHouse has to read.

Here's the workhorse pattern, an hourly rollup for one service:

```
SELECT
  toStartOfHour(ts) AS hour,
  count() AS requests,
  countIf(status >= 500) / count() AS error_rate,
  quantile(0.95)(duration_ms) AS p95_ms
FROM events
WHERE service = 'api'
  AND ts >= now() - INTERVAL 24 HOUR
GROUP BY hour
ORDER BY hour;
```

Three ClickHouse idioms are doing the work.

`toStartOfHour(ts)` truncates each timestamp to its hour, turning a raw event stream into time buckets. The whole family exists: `toStartOfDay`, `toStartOfWeek`, `toStartOfInterval` for custom windows.

`countIf(status >= 500)` is a **conditional aggregate** — it counts only rows matching the condition, in one pass. Most aggregates have an `-If` variant (`sumIf`, `avgIf`), which replaces the self-joins or `CASE` pyramids you'd write elsewhere.

`quantile(0.95)(duration_ms)` computes an approximate 95th percentile. Approximate is the default posture here: `uniq(user_id)` estimates distinct counts with a small, bounded error and runs far faster than `uniqExact(user_id)`. For a dashboard, trading a fraction of a percent of accuracy for a big speedup is almost always right; use the exact versions when the number feeds billing or an SLA.

Keep units visible

Label units in names and results. `duration_ms`, `p95_ms`, `bytes_out` — the suffix travels with the column into dashboards and CSV exports. An unlabeled `p95` of `310` will eventually be read as seconds by someone, and that someone will page you.

The same discipline applies to rates: `error_rate` above is a fraction between 0 and 1. If a chart needs percent, multiply at the display layer, not in the query that other queries copy from.

Verify against a dataset you can count by hand

Build hourly request counts, error rates, and duration percentiles for one service. Verify each result against a tiny hand-checked dataset:

```
INSERT INTO events VALUES
  ('2026-08-03 10:05:00', 'api', 1, 'request', 40,  200),
  ('2026-08-03 10:20:00', 'api', 2, 'request', 60,  200),
  ('2026-08-03 10:40:00', 'api', 3, 'request', 300, 500);
```

Three rows, one hour: the count must be 3 and the error rate 0.333. If the query is wrong on three rows you can check mentally, it's wrong on three billion — you just wouldn't have noticed. Validate the logic small, then point it at the real table.

Inspect query work

Use EXPLAIN, query logs, system tables, read rows, read bytes, parts, and profile events before changing schema or adding accelerators.

Optimization starts with evidence. Read how many rows and bytes the query processed, which parts and granules it skipped, its memory use, and where time went. Guessing leads to cargo-cult schema changes; the numbers below tell you exactly what a query cost.

The quickest evidence is free: `clickhouse-client` prints a summary after every query.

```
1 row in set. Elapsed: 0.018 sec. Processed 8.19 thousand rows, 65.54 KB
```

`Processed ... rows` is the headline number. A query that returns 20 rows but processes 500 million is doing enormous work to find its answer — that's your signal, long before anyone complains about latency.

Ask the planner what it will skip

Use `EXPLAIN` and relevant `system` tables to inspect queries and storage. With `indexes = 1`, `EXPLAIN` shows how much data the primary index eliminates:

```
EXPLAIN indexes = 1
SELECT count() FROM events WHERE service = 'api';
```

```
PrimaryKey
  Keys: service
  Condition: (service in ['api', 'api'])
  Parts: 2/6
  Granules: 41/482
```

Read the fractions. `Granules: 41/482` means the index narrowed the scan to 41 blocks out of 482 — the filter is doing its job. A filter on a column the ordering key can't help with reads `482/482`: a full scan. That's the signature of the classic failure — an ordering key chosen for one query shape while the dashboards ask a different one. A faster machine cannot repair an ordering key that forces every query to scan all events.

The query log remembers

Every finished query lands in `system.query_log` with its full cost:

```
SELECT query_duration_ms, read_rows,
  formatReadableSize(read_bytes) AS data,
  formatReadableSize(memory_usage) AS mem
FROM system.query_log
WHERE type = 'QueryFinish'
ORDER BY event_time DESC
LIMIT 5;
```

This is how you audit yesterday's slow dashboard without reproducing it: find the query, read `read_rows` and `memory_usage`, and you know whether the problem is scanning, aggregation memory, or something else entirely. The `ProfileEvents` column on the same row breaks the work down further when you need it.

Baseline before you change anything

Run one selective and one unselective query. Save their read rows, bytes, duration, and plan before changing anything.

That baseline is the whole discipline. Schema changes, projections, and materialized views all have costs, and the only way to know a change paid off is comparing the same numbers before and after. ”It feels faster” has fooled everyone at least once; `read_rows` dropping from 500 million to 2 million has never fooled anyone.

Add a materialized view

Precompute a specific repeated aggregation into a target table while planning backfill, retries, duplicates, and schema changes.

An incremental materialized view transforms newly inserted blocks and writes results into a target table. It moves repeated query work to ingestion time.

Think of it as an insert trigger, not a cached query. When a block of rows lands in the source table, the view's `SELECT` runs on just that block and appends the result to a target table you own. If your dashboard recomputes the same hourly counts every thirty seconds, this trades that repeated scan for a little work on each insert.

The target table comes first, then the view that feeds it:

```
CREATE TABLE events_hourly (
  hour     DateTime,
  service  LowCardinality(String),
  requests UInt64
)
ENGINE = SummingMergeTree
ORDER BY (service, hour);

CREATE MATERIALIZED VIEW events_hourly_mv TO events_hourly AS
SELECT
  toStartOfHour(ts) AS hour,
  service,
```

```
  count() AS requests
FROM events
GROUP BY hour, service;
```

`SummingMergeTree` matters here: each insert block produces partial counts, and the target engine sums rows sharing the same key during merges. Query it with `sum(requests)` and a `GROUP BY` so partial rows that haven't merged yet are combined correctly.

Backfill without double counting

The view does not automatically rewrite old source rows. It only sees inserts that happen after it exists, so history needs a manual backfill. Plan a backfill and avoid double counting while live inserts continue.

The safe pattern is a cutoff. The view handles everything from its creation moment forward; you insert everything strictly before that moment:

```
INSERT INTO events_hourly
SELECT toStartOfHour(ts) AS hour, service, count() AS requests
FROM events
WHERE ts < '2026-08-03 12:00:00'
GROUP BY hour, service;
```

Backfill a separate time range, then compare the view result with the raw query. If the boundary overlaps — or you run the backfill twice — those hours count double, and nothing warns you.

The failure modes are duplicates

Understand how retries and source deduplication affect the target. A retried insert that the source table deduplicates may still have been processed by the view, and a view whose `SELECT` throws can fail the original insert. Duplicated source rows become duplicated aggregates. Whenever a dashboard number looks slightly wrong, this comparison is the diagnostic:

```
SELECT sum(requests) FROM events_hourly
WHERE hour = toStartOfHour(now() - INTERVAL 1 HOUR);

SELECT count() FROM events
WHERE toStartOfHour(ts) = toStartOfHour(now() - INTERVAL 1 HOUR);
```

Matching numbers mean the pipeline is honest. Diverging numbers mean a backfill overlap or a retry got counted twice — and because you own `events_hourly` as a real table, you can drop the affected hours and rebuild them from the source.

Create an hourly count target for new events, backfill, and run that comparison before you point any dashboard at the target.

Check your understanding: queries and acceleration

Check the key models, decisions, commands, security boundaries, and failure cases from queries and acceleration.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Security and operations

Users, network controls, parts, merges, retention, replication, backups, monitoring, and recovery.

Secure ClickHouse access

Create least-privilege users, restrict networks, require encrypted connections, protect credentials, and set query resource limits.

Do not expose an unauthenticated ClickHouse server. A fresh install listens on port 8123 (HTTP) and 9000 (native) with a `default` user that historically has no password — bind it to the internet and automated scanners will find it within hours. Restrict network paths, use TLS, create separate users and roles, and grant only the databases and operations each workload needs.

Network first: keep ClickHouse listening on private interfaces only, and let applications reach it through a private network or a tunnel. TLS on the client ports keeps credentials and query results off the wire. Then split identities per workload — a pipeline that only inserts and a dashboard that only reads should not share credentials, because a leaked dashboard key must not be able to delete a table.

Least-privilege users

SQL-driven access control makes this concrete:

```
CREATE USER ingest IDENTIFIED WITH sha256_password BY 'a-long-random-secret';
GRANT INSERT ON analytics.events TO ingest;

CREATE USER dashboards IDENTIFIED WITH sha256_password BY 'another-long-secret';
GRANT SELECT ON analytics.* TO dashboards;
```

`ingest` can write events and nothing else — it cannot even read them back. `dashboards` can read anything in `analytics` and change nothing. With more than a couple of users, create roles (`CREATE ROLE readonly_analytics`), grant privileges to the role, and grant the role to users.

Resource limits are security too

Analytical queries can consume large CPU and memory. Apply quotas, timeouts, and resource settings so one dashboard or ad hoc query cannot exhaust the cluster:

```
CREATE SETTINGS PROFILE dashboard_limits SETTINGS
  max_memory_usage = 10000000000,
  max_execution_time = 30
TO dashboards;
```

Now a runaway `GROUP BY` from the dashboard user dies at 10 GB or 30 seconds instead of taking ingestion down with it. A `CREATE QUOTA` can additionally cap queries per hour for ad hoc users. This is availability protection: an unbounded query is a denial of service, whether or not anyone meant it.

Prove the boundaries hold

Create an ingestion user and a read-only dashboard user. Prove each account is denied the other account's privileged operation:

```
$ clickhouse-client --user dashboards --password '...' \
```

```
    --query "INSERT INTO analytics.events VALUES (...)"
Code: 497. DB::Exception: dashboards: Not enough privileges.
```

Run the mirror test too — `ingest` attempting a `SELECT` should fail the same way. An access model you haven't tested from the outside is a diagram, not a control. The realistic failure here isn't an exotic exploit: it's one shared superuser credential pasted into every service config, waiting for the first leak to become a full compromise.

Operate parts, retention, and replication

Monitor part growth and merges, apply TTL or partition retention, and separate replicated availability from distributed sharding.

Healthy MergeTree tables maintain a manageable stream of parts and background merges. Too many tiny inserts or partitions can overwhelm that work, and part count is the vital sign that tells you early:

```
SELECT count() AS active_parts, sum(rows) AS total_rows
FROM system.parts
WHERE table = 'events' AND active;
```

A steady table holds tens to low hundreds of active parts. A count climbing into the thousands means merges are losing the race against inserts — fix the insert pattern before the `Too many parts` errors start. `system.merges` shows what merge work is running right now.

Retention that matches the storage model

Use TTL rules or deliberate partition drops for retention. A TTL clause makes expiry automatic:

```
ALTER TABLE events MODIFY TTL ts + INTERVAL 90 DAY;
```

ClickHouse removes expired rows during background merges — eventually, not at midnight on day 90. Partition drops (`ALTER TABLE events DROP PARTITION '202505'`) are the manual alternative: instant, predictable, and cheap because ClickHouse deletes the partition's parts outright instead of rewriting anything.

What retention should *not* be is `ALTER TABLE events DELETE WHERE ts < ...` on a schedule. That's a **mutation**, and mutations are asynchronous: the statement returns immediately while ClickHouse rewrites every affected part in the background. Check what's actually finished:

```
SELECT command, is_done, latest_fail_reason
FROM system.mutations
WHERE table = 'events';
```

The classic surprise is running a delete, seeing it ”succeed”, and finding the rows still present in the next query — `is_done = 0` means the rewrite is still grinding, possibly for hours on a big table. A stuck mutation with a `latest_fail_reason` blocks the ones queued behind it. Mutations are for rare corrections; TTL and partition drops are for routine retention.

Replication and sharding solve different problems

Replication provides copies and failover; distributed tables and shards spread data and query work. Each adds failure modes and operational cost.

A `ReplicatedMergeTree` table keeps the same data on several servers, coordinated through Keeper.

Lose one replica and the others keep serving — availability. Sharding splits the dataset across servers behind a `Distributed` table so no single machine holds or scans everything — capacity. Losing one replica of a replicated table costs you redundancy; losing a shard with no replicas costs you that slice of the data. That asymmetry is why production clusters replicate each shard, and why a single well-sized server with backups is the right starting point until data volume forces the complexity.

Set a short retention policy on disposable events. Observe parts before and after merges, then explain how one replica failure differs from one shard failure — if the explanation takes more than two sentences, revisit this lesson before operating a cluster.

Back up, monitor, and restore

Back up metadata and data, monitor query and storage health, test restore into isolation, and keep a recovery plan for operator errors.

Replicas are not backups because a bad delete or mutation can propagate. Replication faithfully copies your mistake to every server within seconds — a dropped partition, a wrong `ALTER TABLE ... DELETE`, a truncate on the wrong environment. Only a backup holds the state from *before* the mistake.

Back up the tables, metadata, users, and configuration required by the recovery goal. ClickHouse has native commands for the data and schema:

```
BACKUP TABLE analytics.events
TO Disk('backups', 'events-2026-08-03.zip');
```

The `backups` destination is a disk you configure in the server's storage settings; S3-compatible object storage is the other common target. The backup contains both schema and data, and later backups to the same destination can be incremental. Don't forget the parts a table backup misses: access entities (users, roles, grants) and server configuration files need their own copy, or your restored cluster has data nobody can query.

Keep the raw event source long enough to rebuild when that is part of the design — a pipeline whose queue or object store retains 30 days of events is itself a recovery path for recent data.

Monitor the things that fail quietly

Monitor query failures, ingestion lag, disk, parts, merges, replication queues, memory, and backup jobs. Most of these come straight from system tables — failed queries from `system.query_log`, part counts from `system.parts`, replication debt from `system.replication_queue`, and every backup's outcome:

```
SELECT name, status, error
FROM system.backups
ORDER BY start_time DESC
LIMIT 5;
```

A backup job that has been failing for three weeks is indistinguishable from no backup at all, and it's always discovered at the worst moment. Alert on `status != 'BACKUP_CREATED'`, not just on the job running.

A restore you haven't run is a hypothesis

Restore a backup into an isolated database or cluster — never onto the production table first:

```
RESTORE TABLE analytics.events AS analytics.events_restored
FROM Disk('backups', 'events-2026-08-03.zip');
```

Then compare row counts, one aggregate, permissions, and the dashboard result before accepting the recovery:

```
SELECT count() FROM analytics.events_restored;
SELECT service, count() FROM analytics.events_restored
GROUP BY service ORDER BY service;
```

The counts tell you the data arrived; the aggregate tells you it's the *right* data; a login with the dashboard user tells you access survived; and loading one real dashboard against the restored copy tells you the whole chain works.

Run this drill on a schedule, not during an incident. The point of rehearsal is finding the missing grant or the misconfigured backup disk on a calm Tuesday, when it's a ticket instead of an outage.

Check your understanding: security and operations

Check the key models, decisions, commands, security boundaries, and failure cases from security and operations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/clickhouse/.