3.2 — ClickHouse: Real-Time Analytics at Scale

Data Processing at Scale

2026-03-27

Data Processing at Scale — Day 3

Course website

https://vbergeron.github.io/data-processing-at-scale/

This presentation

https://vbergeron.github.io/data-processing-at-scale/3.2-clickhouse.pdf

Introduction


ClickHouse

A brief history

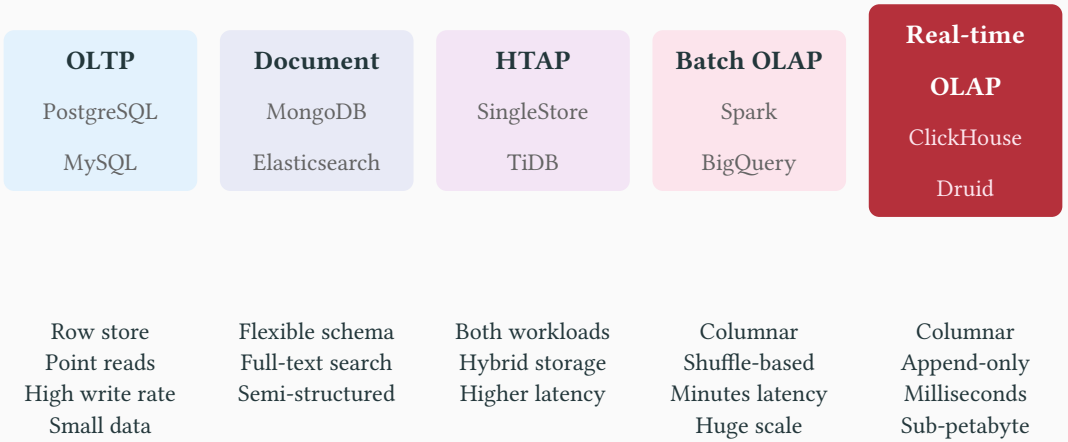
Year	Event
2009	Started at Yandex by Alexey Milovidov to power Yandex.Metrica — the web analytics platform processing hundreds of billions of events per day
2016	Open-sourced under the Apache License 2.0; immediately adopted by teams outside Yandex needing sub-second OLAP at scale
2021	ClickHouse Inc. founded; Series A & B funding; cloud-managed service launched

Today, ClickHouse is adopted at Cloudflare (DNS query logs, 13M+ events/s), Uber, Discord, ByteDance, Bloomberg, Stripe, and many others.

In the beginning

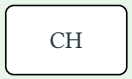
ClickHouse started as an internal tool for one specific problem — aggregating clickstream data fast enough to power real-time dashboards — and its design has never deviated from that goal. Every architectural decision traces back to **making analytical queries faster**.

The OLTP – OLAP continuum

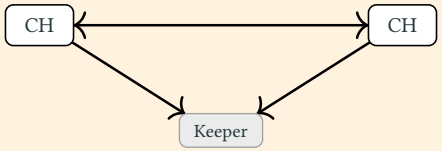


Topology & deployment

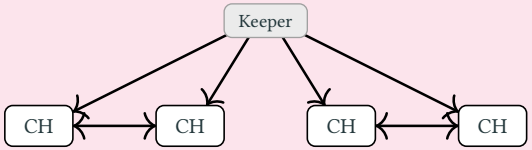
Standalone



Replicated



Sharded cluster



A single ClickHouse node scales to tens of terabytes and is the right starting point for most teams. Replication adds fault-tolerance; sharding adds horizontal scale — both require CH Keeper for coordination.

Protocols

Protocol	Port	Notes
Native binary	9000	ClickHouse's own TCP protocol — columnar wire, LZ4/ZSTD, streaming, progress callbacks. Used by `clickhouse-client` and native drivers
HTTP	8123	Plain HTTP/1.1 — format negotiated via `FORMAT` clause (JSON, CSV, Parquet, Arrow…). Any `curl` or BI tool works out of the box
MySQL wire	9004	MySQL protocol compatibility — connect any MySQL-compatible client or BI tool without a ClickHouse-specific driver
Arrow Flight SQL	9100	gRPC + Apache Arrow — zero-copy columnar transfers for tools built on the Arrow ecosystem (Pandas, Polars, DuckDB)

The MergeTree Engine Family

MergeTree

ClickHouse's primary storage engine. The name is literal: data is written in sorted, immutable **parts**, and a background process continuously **merges** them.

MergeTree and LSM trees

MergeTrees shares core ideas with **LSM trees** (RocksDB, Cassandra, LevelDB).

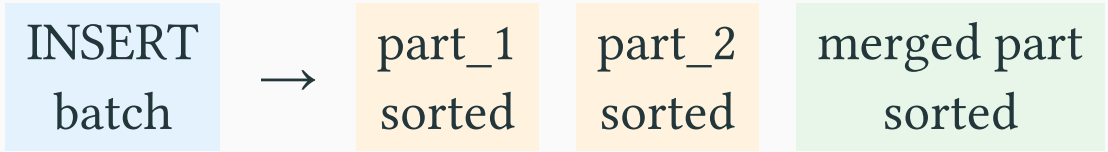
- writes are cheap because they are sequential appends,
- reads amortize the cost of background compaction.

Key differences:

- **column-oriented** — each part stores data column-by-column
- compaction is driven by **analytical query patterns** (sort order, aggregation state)

The write path

ClickHouse writes are **append-only**. Every `INSERT` creates one or more immutable, sorted **parts** on disk. Parts are never modified — only merged.



- Each part is a directory of column files, a primary index, and metadata
- Background **merges** consolidate many small parts into fewer large ones
- The merge process is where engine-specific logic runs (deduplication, aggregation, rollup…)
- Inserts should be **batched** — many tiny inserts create too many parts and slow merges

ORDER BY — the most important schema decision

`ORDER BY` determines the **physical sort order** of every part. ClickHouse builds a **sparse primary index** over this order.

```
CREATE TABLE events (
  ts       DateTime,
  user_id  UInt64,
  action   String,
  amount   Float64
) ENGINE = MergeTree
ORDER BY (ts, user_id);
```

- ClickHouse stores one index entry per **granule** (8 192 rows by default)
- Queries filtering on the leading `ORDER BY` columns skip irrelevant granules entirely
- A date-range query on `(ts, user_id)` is fast — a user-id filter alone must scan everything

ORDER BY is not a preference — it is the primary index. Choose based on your most frequent query filters, not on what feels natural.

The engine family

Engine	Merge behavior
`MergeTree`	No special logic — just compaction and sorting
`ReplacingMergeTree`	Deduplicates by primary key; keeps the latest version
`CollapsingMergeTree`	Cancels rows with opposite `sign` values — efficient deletes
`VersionedCollapsingMergeTree`	Same as Collapsing, but safe with out-of-order data
`AggregatingMergeTree`	Merges intermediate aggregation states — backbone of materialized views
`SummingMergeTree`	Sums numeric columns on merge — lightweight pre-aggregation

ReplacingMergeTree and Eventual Consistency

`ReplacingMergeTree` deduplicates rows with the same primary key, keeping the row with the highest version (or the last inserted if no version column).

```
CREATE TABLE users (
  user_id UInt64,
  name    String,
  version UInt64
) ENGINE = ReplacingMergeTree(version)
ORDER BY user_id;
```

Deduplication is asynchronous. Merges happen in the background — a query may still return duplicates until a merge has run.

ReplacingMergeTree and Eventual Consistency

Two patterns to handle this:

- `SELECT … FINAL` — forces deduplication at query time; correct but up to 2× slower
- `argMax(name, version)` — pick the latest value in the application layer; fast, no `FINAL` needed

Deletes in MergeTree

MergeTree is append-only — there are two ways to delete rows, with very different cost profiles.

```
-- Lightweight DELETE (recommended): marks rows as deleted via a hidden
_row_exists mask.
-- Fast to issue; physical removal happens on the next background merge.
DELETE FROM events WHERE ts < now() - INTERVAL 90 DAY;

-- Heavyweight mutation: rewrites every affected part entirely.
-- Avoid on large tables; use only when immediate physical removal is required.
ALTER TABLE events DELETE WHERE ts < now() - INTERVAL 90 DAY;
```

Alternatively, for large deletes, partition by time and use `DROP PARTITION` or `TRUNCATE`.

Deletes in MergeTree

	Lightweight `DELETE`	**`ALTER TABLE … DELETE`**
Mechanism	Writes a `_row_exists` mask	Rewrites all columns in affected parts
Visibility	Rows hidden immediately; physically removed on next merge	Rows gone after mutation completes
Cost	Low — proportional to matched rows	High — proportional to data in affected parts

Storage Layout & Compression

Columnar layout on disk

Every column gets its own pair of files:

- `column.bin` holds the compressed data
- `column.mrk3` holds the **marks** — byte offsets into `.bin`

`ts.bin`	`ts.mrk3`	`amount.bin`	`amount.mrk3`	`user_id.bin`	`user_id.mrk3`	`primary.idx`
data	offsets	data	offsets	data	offsets	sparse index

Compression and encoding

ClickHouse applies two layers: a **codec** (data-type-aware transformation) followed by a general-purpose **compressor**.

Column type	Codec	Effect
Timestamps	`Delta + ZSTD`	Store differences between consecutive values
Low-cardinality strings	`LowCardinality`	Dictionary-encodes the column
Floats	`Gorilla`	XOR-based delta — efficient for slowly changing metrics
General	`LZ4` (default)	Fast compression / decompression; 3–5× ratio on typical event data
High compression need	`ZSTD`	Slower, but 5–10× ratio; better for cold storage

The sparse primary index

ClickHouse does not use a B-tree. It stores **one index entry per granule** in `primary.idx`.

#0-8191	#8192-16383	#16384-24575	#24576-32767
min ts: 2024-01-01	min ts: 2024-01-02	min ts: 2024-01-03	min ts: 2024-01-05

Skip indexes

Beyond the primary index, ClickHouse supports secondary **data skipping indexes** — lightweight metadata stored per granule.

Type	Stores per granule	Best for
`minmax`	Min and max value	Numeric ranges, timestamps
`set(N)`	Set of up to N distinct values	Low-cardinality filtering (`WHERE status = 'error'`)
`bloom_filter`	Probabilistic membership	High-cardinality string matching (`WHERE user_id IN (...)`)
`tokenbf_v1`	Token bloom filter	Full-text search, log analysis

Skip indexes are **advisory** — they add a granule-level check but never guarantee a match. A false positive costs one extra granule read; a true negative skips it entirely. They complement the primary index; they do not replace it.

Query Execution

Vectorized execution

ClickHouse processes data in **column-oriented batches** of one granule (8 192 rows) at a time.

- Pull one granule (8 192 rows) at a time.
- Apply each operator to the full column vector.
- SIMD-friendly, L1/L2 cache stays warm.
- The column fits in cache — no per-row overhead

On arithmetic-heavy aggregations, vectorized execution is 10–100× faster than tuple-at-a-time.
ClickHouse also generates **runtime-compiled code** (via LLVM) for the hot path of complex expressions.

Reading EXPLAIN

```
EXPLAIN indexes = 1
SELECT sum(amount)
FROM events
WHERE ts >= '2024-01-01' AND ts < '2024-02-01';
```

Key fields to read:

- `ReadFromMergeTree` — which table and how many **selected marks** (granules) will be read
- `Condition (ts ...)` — which parts of the index were applied to prune granules
- `Parts: 42/180` — 42 parts matched out of 180 total

Compare `selected_marks` before and after adding an index or changing `ORDER BY`. This is the primary tool for understanding if your schema is working.

Observability — system.query_log

Every executed query is logged to `system.query_log` (after a short flush delay). The most useful columns:

```
SELECT
  query_duration_ms,
  read_rows,
  read_bytes,
  memory_usage,
  query
FROM system.query_log
WHERE type = 'QueryFinish'
ORDER BY query_duration_ms DESC
LIMIT 10;
```

- `read_rows` vs total rows in the table — how much was skipped
- `read_bytes` — actual I/O after decompression; compare with compressed file sizes
- `memory_usage` — peak memory; watch for aggregations that spill

ClickHouse SQL

Array functions

`arrayMap`, `arrayFilter`, and `arraySum` operate without unnesting

`arrayJoin` turns each element into a row.

```
SELECT arrayJoin(tags) AS tag, count() AS occurrences
FROM events
GROUP BY tag
ORDER BY occurrences DESC;
```

`arrayJoin` is a lateral unnest — a row with N tags becomes N rows. No secondary table required.

Approximate aggregation

For billion-row tables, exact distinct counts and percentiles are expensive
ClickHouse ships purpose-built approximate functions with bounded error guarantees.

```
SELECT
    toStartOfDay(ts)          AS day,
    uniq(user_id)             AS approx_dau,     -- HyperLogLog
    quantile(0.95)(duration) AS p95_duration,   -- Reservoir sampling
    topK(10)(page)            AS top_pages      -- Space-Saving
FROM events
GROUP BY day;
```

Aggregation combinators

ClickHouse aggregation functions accept **combinators** — suffixes that modify their behavior without a subquery.

```
SELECT
    vendor_id,
    count()                          AS total_trips,
    countIf(payment_type = 'card') AS card_trips,
    sumIf(fare, tip > 0)             AS tipped_revenue
FROM trips
GROUP BY vendor_id
WITH TOTALS;
```

`-If` applies a filter condition to any aggregate: `avgIf`, `maxIf`, `uniqIf`. `WITH TOTALS` appends a grand-total row. Combinators compose: `uniqArrayIf` is valid.

SAMPLE

`SAMPLE` reads a deterministic fraction of granules at the storage level — not post-scan filtering.

```
SELECT count() / 0.1 AS estimated_total
FROM trips SAMPLE 0.1;
```

`SAMPLE 0.1` skips 90% of granules entirely — unlike `ORDER BY rand() LIMIT N` which scans the full table first. The same fraction always returns the same rows on the same data, making results reproducible.

ASOF JOIN

`ASOF JOIN` matches each left row to the closest preceding right row on a time column.

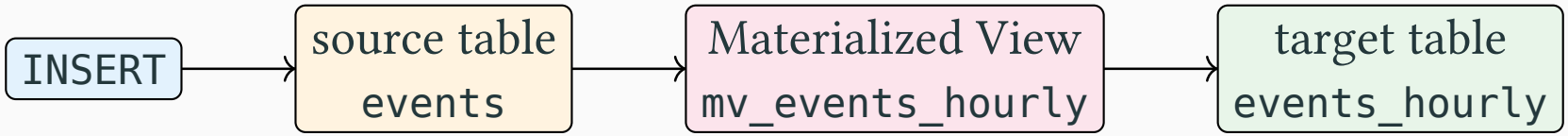
```
SELECT t.symbol, t.quantity, p.price, t.quantity * p.price AS market_value
FROM trades AS t
ASOF LEFT JOIN prices AS p
    ON  t.symbol = p.symbol
    AND t.ts    >= p.ts;
```

The right table must be sorted by the join key then the time column.
The inequality (>= or >) determines which side "closest" means.

Materialized Views & Projections

Materialized views — incremental pre-aggregation

A ClickHouse materialized view is a **trigger on insert**, not a scheduled refresh. When rows land in the source table, the MV fires, aggregates the new batch, and writes the result to a target table.



Materialized views and aggregation

Key constraint: **the MV only sees the inserted batch**, not the full table. This means the aggregate function must be **combinable** across partial results.

- `SUM`, `COUNT`, `MIN`, `MAX` — combinable: `SUM(SUM(x)) = SUM(x)` over the union
- `AVG`, `MEDIAN` — **not** directly combinable: `AVG(AVG(x)) ≠ AVG(x)` without knowing batch sizes

Materialized views and AggregatingMergeTree

For non-trivially combinable aggregates, use `AggregatingMergeTree` with `*State` functions.

`AggregatingMergeTree` stores **intermediate aggregate state** as binary blobs. States from different parts merge correctly during compaction.

Materialized views and AggregatingMergeTree

```
-- Target table: stores intermediate state
CREATE TABLE events_hourly (
  hour     DateTime,
  views    AggregateFunction(count, UInt64),
  revenue  AggregateFunction(sum,   Float64),
  uniq_u   AggregateFunction(uniq,  UInt64)
) ENGINE = AggregatingMergeTree
ORDER BY hour;
```

Materialized views and AggregatingMergeTree

```
-- Materialized view: fires on every insert into `events`
CREATE MATERIALIZED VIEW mv_events_hourly TO events_hourly AS
SELECT
  toStartOfHour(ts)   AS hour,
  countState()        AS views,
  sumState(amount)    AS revenue,
  uniqState(user_id)  AS uniq_u
FROM events;
```

Materialized views and AggregatingMergeTree

```
-- Query: merge states at read time
SELECT hour, countMerge(views), sumMerge(revenue), uniqMerge(uniq_u)
FROM events_hourly
GROUP BY hour
ORDER BY hour;
```

`*State` functions accumulate binary intermediate state. `*Merge` functions combine those states at query time — even across parts that were written at different times.

Projections — multiple sort orders, one table

A **projection** is an alternative physical layout of a table, stored as hidden parts alongside the main data.

Projections — multiple sort orders, one table

```
-- Main table: sorted by (date, user_id) — good for date-range queries
CREATE TABLE events (...) ENGINE = MergeTree ORDER BY (date, user_id);

-- Projection: sorted by (user_id, date) — good for per-user queries
ALTER TABLE events ADD PROJECTION proj_by_user (
  SELECT * ORDER BY (user_id, date)
);

-- Catchup: materialize the projection
ALTER TABLE events MATERIALIZE PROJECTION proj_by_user;
```

Projections — pre-aggregation

Projections can also include a `GROUP BY` clause, turning them into an embedded pre-aggregation layer — no separate target table needed.

```
ALTER TABLE events ADD PROJECTION proj_daily_revenue (
  SELECT
    toStartOfDay(ts) AS day,
    user_id,
    sum(amount)      AS total_amount,
    count()          AS num_events
  GROUP BY day, user_id
);
ALTER TABLE events MATERIALIZE PROJECTION proj_daily_revenue;
```

Projections — multiple sort orders, one table

ClickHouse automatically rewrites matching queries to read from the projection instead of scanning the raw data. The projection is updated synchronously on every insert — no trigger logic to manage.

```
-- This query hits the projection, not the raw table
SELECT day, sum(total_amount) FROM events GROUP BY day;
```

Projections — multiple sort orders, one table

	Materialized views	**Projections**
Storage	Separate, explicit	Hidden table
Schema	Arbitrary	Reordering or subset
Update	Triggered on insert	Synchronous
Query routing	Explicit query against the target table	Implictly optimized
TTL	Independent	Inherits
JOINs in definition	Supported	Not supported
WHERE in definition	Supported	Not supported
Chaining	Supported	Not supported
LW Deletes	Supported	Not supported

Interoperability & the Modern Stack

ClickHouse integrations

ClickHouse has a **rich ecosystem** of integrations with other systems.
https://clickhouse.com/docs/engines/table-engines/integrations

- **Kafka**
- HDFS / S3 / Iceberg
- MySQL / PostgreSQL
- Arrow Flight SQL (Client and Server)

Tiered storage

Storage tiering is configured in two sections: **disks** (the physical media) and **policies** (how parts move between them). Each table then declares which policy it uses.

```
CREATE TABLE events (...) ENGINE = MergeTree
ORDER BY ts
SETTINGS storage_policy = 'tiered';
```


CAUTION
POTENTIAL
BIOHAZARD PRESENT
UNAUTHORIZED
PERSONNEL KEEP OUT

Tiered storage — disks

```
<storage_configuration>
  <disks>
    <hot>
      <type>local</type><path>/nvme/clickhouse/</path>
    </hot>
    <warm>
      <type>local</type><path>/hdd/clickhouse/</path>
    </warm>
    <cold>
      <type>s3</type><endpoint>https://s3.amazonaws.com/my-bucket/ch/</
endpoint>
    </cold>
  </disks>
  <!-- policies go here -->
</storage_configuration>
```

Tiered storage — policies

```
<policies>
  <tiered>
    <volumes>
      <hot><disk>hot</disk>
        <max_data_part_size_bytes>10737418240</max_data_part_size_bytes>
      </hot>
      <warm><disk>warm</disk>
        <max_data_part_size_bytes>107374182400</max_data_part_size_bytes>
      </warm>
      <cold><disk>cold</disk>
      </cold>
    </volumes>
    <move_factor>0.2</move_factor>
  </tiered>
</policies>
```

Tiered storage — TTL-driven moves

Space pressure is reactive. For predictable archival, use `TTL … TO VOLUME` to move parts by age regardless of disk usage.

```
ALTER TABLE events MODIFY TTL
    ts + INTERVAL 7  DAY  TO VOLUME 'warm',
    ts + INTERVAL 90 DAY  TO VOLUME 'cold';
```

Tiered storage — TTL-driven moves

Both mechanisms compose: TTL handles planned archival, `move_factor` handles unexpected load spikes. Manual overrides are also possible:

```
ALTER TABLE events MOVE PART '20240101_1_1_0' TO DISK 'cold';
ALTER TABLE events MOVE PARTITION '2024-01'   TO DISK 'cold';
```

Tiered storage — trade-offs

Blob storage is slower than local disk, but it is cheaper and more scalable.

For analytical workloads this is acceptable: a query that scans 10 GB from S3 at 1 GB/s takes 10 seconds — fine for a scheduled report, not for a live dashboard.

ClickHouse caches recently accessed S3 parts on local disk (`remote_filesystem_local_cache`). Repeated access to the same cold part pays S3 latency only once.

Wrap-Up

Key vocabulary

Term	Definition
OLAP	Online Analytical Processing — column-oriented, scan-heavy, aggregation-first workloads
Part	Immutable sorted directory of column files written by a single INSERT batch in ClickHouse
Granule	The atomic read unit in ClickHouse (8 192 rows); the sparse primary index has one entry per granule
Sparse primary index	One index entry per granule — small, always fits in RAM, enables granule skipping
ORDER BY	In ClickHouse, defines the physical sort order and is the primary index — the most important schema decision
MergeTree	Base storage engine in ClickHouse; variants define behavior during background merges
ReplacingMergeTree	Deduplicates by primary key on merge; deduplication is asynchronous — `FINAL` or `argMax` needed for consistency

Key vocabulary

AggregatingMergeTree	Stores and merges intermediate aggregate state (`*State` / `*Merge` functions)
Vectorized execution	Processes one column-batch (granule) per operator call — SIMD-friendly, cache-warm
Skip index	Per-granule metadata (minmax, set, bloom filter) enabling additional granule pruning
Materialized view	A trigger on INSERT that incrementally aggregates new data into a target table
`*State` / `*Merge`	Function pair for incremental aggregation: `sumState` accumulates, `sumMerge` finalizes
Projection	An alternative physical sort of a table stored inline; ClickHouse selects automatically per query
Arrow Flight SQL	gRPC protocol for columnar query results over Apache Arrow — zero-copy end-to-end

Key vocabulary

Tiered storage	Storage policy routing parts across NVMe / block store / S3 by age or access frequency

One sentence to remember

ClickHouse is built without compromise: **columnar, analytical, and append-only**.

One sentence to remember

Lab

https://vbergeron.github.io/data-processing-at-scale/lab-3.2-clickhouse.pdf