ClickHouse Performance Observability and Monitoring

The complete metric matrices for troubleshooting ClickHouse performance — queries, MergeTree storage, ingestion, memory, CPU, I/O, caches, replication, and Keeper — with alert thresholds and a symptom-driven diagnosis matrix.

ChistaDATA Performance Engineering

First Edition · July 2026

- **Metric matrices for every ClickHouse subsystem, with healthy targets and red flags**
- **The system-table map: where every signal lives and how long it is retained**
- **Alert threshold starter set and configuration audit checklist**
- **Symptom → cause → first-check troubleshooting matrix from field engagements**

Notices

This publication was developed by ChistaDATA Inc. for practitioners operating ClickHouse® in production environments. References to specific settings, system tables, and metric names reflect ClickHouse open-source releases current at the time of writing (mid-2026); names and defaults evolve between releases — always confirm against the version you operate.

This information is provided "AS IS" without warranty of any kind, express or implied. The thresholds and recommendations herein are engagement starting points drawn from ChistaDATA's field practice across OLAP, real-time analytics, and observability workloads; they are not absolutes. Any performance data contained herein was determined in specific operating conditions — actual results may vary. Always validate configuration changes in a non-production environment first. ChistaDATA Inc. assumes no liability for damages resulting from the use of this material.

COPYRIGHT

TRADEMARKS

ChistaDATA™ and the ChistaDATA wordmark are trademarks of ChistaDATA Inc. ClickHouse® is a registered trademark of ClickHouse, Inc.; this is an independent publication and is not affiliated with, endorsed by, or sponsored by ClickHouse, Inc. Apache Kafka®, Apache ZooKeeper®, and associated marks are trademarks of the Apache Software Foundation. Linux® is a registered trademark of Linus Torvalds. Grafana® is a trademark of Raintank, Inc. dba Grafana Labs. Prometheus® is a trademark of The Linux Foundation. Amazon S3™ is a trademark of Amazon.com, Inc. All other product and company names mentioned herein are trademarks or registered trademarks of their respective owners and are used for identification purposes only.

EDITION NOTICE

First Edition, July 2026. This edition applies to ClickHouse 24.x and later open-source releases, and to ChistaDATA-supported ClickHouse deployments on bare metal, virtual machines, Kubernetes, and object-storage-backed clusters.

PUBLISHED BY

ChistaDATA Inc. · https://chistadata.com
Enterprise-class 24×7 ClickHouse consultative support and managed services.

Preface

ClickHouse is spectacularly observable: nearly everything the server knows about itself is exposed through SQL, in the `system` database. The difficulty is not access to signals — it is knowing *which* of the thousands of metrics, events, and log tables matter, what healthy looks like for each, and which subsystem a bad number points to. This book encodes those matrices the way ChistaDATA engineers carry them into production engagements.

WHO SHOULD READ THIS BOOK

Site reliability engineers, database administrators, platform engineers, and developers who operate ClickHouse in production — or are about to. Chapters assume working ClickHouse knowledge (MergeTree basics, replication concepts) but no prior monitoring investment. If you already run Prometheus and Grafana, Chapter 13 maps this book's matrices onto that stack directly.

HOW THIS BOOK IS ORGANIZED

Chapter 1 establishes the observability model: the three signal planes, the delta discipline, and the four-question triage that routes every investigation. **Chapter 2** maps the system tables — where every signal lives, at what granularity, and for how long. **Chapters 3–12** are the metric matrices, one subsystem per chapter: queries, MergeTree parts and merges, ingestion, memory, CPU and background pools, disk and object storage, caches, replication and Keeper, distributed processing, and error signals. **Chapter 13** turns the matrices into a monitoring stack with a starter alert set; **Chapter 14** is the configuration audit checklist; **Chapter 15** is the symptom-driven troubleshooting matrix. **Appendix A** collects the field queries referenced throughout.

CONVENTIONS USED IN THIS BOOK

Every matrix follows one format: the metric, its source (system table, ProfileEvent counter, or asynchronous metric), what it tells you, a **healthy target** in bold blue, and the **red flag** with the action it points to. Targets are starting points for typical analytical workloads on SSD-backed clusters — calibrate to your hardware and workload, and always judge counters on deltas between timed samples, never on raw totals.

ABOUT THE AUTHORS

This book was produced by the ChistaDATA Performance Engineering team — the group that runs ClickHouse health checks, performance audits, and 24×7 managed operations for ChistaDATA customers worldwide. The same matrices power `chaudit`, ChistaDATA's open-source ClickHouse performance auditor, which automates the collection queries in Appendix A.

> **Note:** Throughout this book, "Keeper" refers to ClickHouse Keeper and applies equally to Apache ZooKeeper deployments unless a difference is called out explicitly.

Contents

Important: ClickHouse event counters (`system.events`, ProfileEvents) are cumulative since server start, and reset on restart. Every threshold in this book assumes **deltas between two timed samples** — 30–60 seconds for triage, 5–15 minutes for trend work — captured under representative load. `system.metric_log` keeps a per-second history of all of them; enable it before you need it.

CHAPTER 1

The ClickHouse Observability Model

ClickHouse exposes its internals through three signal planes, all queryable in SQL. Understanding what each plane measures — and how the planes differ in granularity and lifetime — is the foundation for every matrix in this book.

1.1 The three signal planes

PLANE	PRIMARY SOURCES	WHAT IT MEASURES	CHARACTER
Gauges point-in-time	`system.metrics`, `system.asynchronous_metrics`	Current state: running queries, active merges, memory tracked, replica delay, background pool tasks, load average, disk free.	Instantaneous values. `metrics` updates in real time; `asynchronous_metrics` refreshes on an interval (~1 min default) and includes host-level readings.
Counters cumulative	`system.events`; per-query `ProfileEvents` in `query_log`	Everything that has happened since start: rows read, marks selected, cache hits/misses, delayed inserts, fsyncs, network bytes, lock waits.	Monotonic since server start. Meaningful only as deltas over an interval, or per query via the `ProfileEvents` map column.
Logs per-occurrence	`query_log`, `part_log`, `trace_log`, `text_log`, `crash_log`, `metric_log`	One row per event with full context: each query with its ProfileEvents, each part created/merged, sampled stack traces, server log lines.	MergeTree tables on disk; survive restarts. Must be enabled in config and given a TTL — they are your retrospective flight recorder.

1.2 The delta discipline

A counter total accumulated over three weeks of uptime answers no operational question. The working pattern is: sample `system.events` twice with a known interval and subtract — or better, read `system.metric_log`, which materializes exactly that: one row per second with every metric and event delta already computed. Retrospective questions ("what happened at 03:12?") are answered from `metric_log` and `query_log`, not from memory.

1.3 Baseline while healthy

Every threshold in this book is a starting point; your cluster's normal is the real reference. Capture a healthy-day profile — QPS, p95 latency, rows-read per query, merge rates, parts per partition, replication delay, memory high-water — and keep it with the runbook. Incidents then become difference questions, which are an order of magnitude easier than absolute ones.

1.4 The four-question triage

Nearly every ClickHouse performance investigation routes through four questions, in order. Each maps to a chapter group in this book:

QUESTION	WHAT ANSWERS IT	IF YES → GO TO
1. Is it the queries?	Slow or heavy SELECTs: rows read exploding, marks not pruned, memory per query, concurrency pile-ups. `query_log` ranked five ways.	Chapter 3 (queries), Chapter 6 (memory), Chapter 9 (caches)
2. Is it the storage engine?	Parts count rising, merge backlog, mutations stuck, compression degrading — the MergeTree machinery falling behind the write rate.	Chapter 4 (parts & merges), Chapter 5 (ingestion)
3. Is it the cluster?	Replication delay, readonly replicas, Keeper session instability, distributed queue backlogs, shard skew.	Chapter 10 (replication & Keeper), Chapter 11 (distributed)
4. Is it the host?	CPU saturation, disk latency, page-cache starvation, cgroup memory pressure, network errors — the layer beneath ClickHouse.	Chapter 7 (CPU), Chapter 8 (disk I/O), Chapter 6 (memory)

1.5 Incident first moves

Before changing anything, capture evidence: `system.processes` (what is running now), `SHOW PROCESSLIST` equivalent with `elapsed` and `memory_usage`, the last 30 minutes of `metric_log` aggregates, and `system.errors` ordered by `last_error_time`. Four queries, under a minute, and they survive the restart that someone will soon propose. Chapter 15 builds the full symptom matrix on top of these captures.

Tip: ClickHouse ships a built-in observability UI: the `/dashboard` HTTP endpoint renders the queries defined in `system.dashboards` against `metric_log` — zero-install monitoring for triage on a cluster you have never seen before. ChistaDATA engineers open it in the first five minutes of every engagement.

Note: Enable `query_log`, `part_log`, `metric_log`, and `text_log` on every production cluster, each with a TTL (14–30 days is typical). The storage cost is trivial next to the diagnostic value; a cluster without them can only be debugged forward, never backward.

CHAPTER 2

The System-Table Map

Where every signal lives. Instrumentation tables (this page) are in-memory and reflect now; log tables (next page) are MergeTree tables on disk and reflect history.

2.1 Instrumentation and state tables

TABLE	WHAT IS IN IT	USE IT FOR
system.metrics	Real-time gauges: `Query`, `Merge`, `MemoryTracking`, `ReadonlyReplica`, pool task counts, connection counts.	Current load and saturation; the first table an alert should query.
system.events	Cumulative counters: ~600 ProfileEvents from `SelectedRows` to `S3RequestsErrors`.	Rates via deltas; cache hit ratios; anomaly detection against baseline.
system.asynchronous_metrics	Periodic readings: `LoadAverage1`, `OSMemoryAvailable`, disk free per mount, `ReplicasMaxAbsoluteDelay`, jemalloc stats, uptime.	Host-level health and cluster-wide worst-case gauges without shelling into the box.
system.processes	One row per running query: `elapsed`, `read_rows`, `memory_usage`, `query`, settings.	What is running right now; the kill list during a pile-up (`KILL QUERY`).
system.merges	Active merges/mutations: progress, elapsed, rows read/written, memory, is_mutation.	Merge pressure now; stuck or giant merges.
system.mutations	ALTER UPDATE/DELETE state: `parts_to_do`, `is_done`, `latest_fail_reason`.	Mutation backlog and failures — a classic silent killer.
system.parts	One row per data part: size, rows, partition, level, compression, `active`.	Parts-per-partition counts, compression ratios, storage audit.
system.replicas	Per replicated table: `absolute_delay`, `queue_size`, `is_readonly`, `future_parts`, last exceptions.	Replication health; the single most alert-worthy table (Ch. 10).
system.replication_queue	Pending replication tasks with `num_tries`, `last_exception`, `postpone_reason`.	Why replication is stuck, entry by entry.
system.distribution_queue	Pending async inserts for Distributed tables: files, bytes, broken files.	Distributed insert backlog and poison-pill batches (Ch. 11).
system.errors	Every error code raised since start: count, `last_error_time`, last message and trace.	The cheapest anomaly detector in ClickHouse (Ch. 12).
system.disks / system.storage_policies	Disks, free/total bytes, policies, volumes.	Capacity and tiering audit (Ch. 8).
system.dictionaries, system.clusters, system.zookeeper	Dictionary load state and memory; cluster topology; live Keeper tree.	Dictionary failures; shard/replica inventory; Keeper inspection (Ch. 10).

2.2 Log tables — the flight recorder

TABLE	ONE ROW PER…	USE IT FOR
system.query_log	Query lifecycle event (start, finish, exception), with duration, rows/bytes read, memory peak, settings, and the full `ProfileEvents` map.	The center of query troubleshooting: top-N by any cost, latency percentiles, per-pattern aggregation via `normalized_query_hash` (Ch. 3).
system.query_thread_log	Thread that participated in a query.	Intra-query parallelism analysis; usually enabled only while investigating.
system.processors_profile_log	Pipeline operator per query: work time, wait time, rows in/out.	Which operator (join, aggregation, reading) actually burned the time.
system.part_log	Part lifecycle event: NewPart, MergeParts, DownloadPart, RemovePart, MutatePart, with duration and size.	Insert block sizes, merge throughput/duration history, fetch volume (Ch. 4–5).
system.metric_log	Second, containing every metric gauge and event delta.	Retrospective triage: reconstruct any dashboard for any past minute.
system.asynchronous_metric_log	Asynchronous metric reading.	Host-level history (load, memory, disk) without external monitoring.
system.trace_log	Sampled stack trace (CPU, real time, memory sampling).	Built-in profiler: flamegraphs of where the server spends CPU or allocates.
system.text_log	Server log line, structured.	Error-rate alerting in SQL; correlating messages with metric anomalies.
system.crash_log	Fatal signal, with stack trace.	Post-mortem after a crash; should be empty and alerted on.
system.query_views_log	Materialized view executed per source insert.	MV fan-out cost attribution (Ch. 5).
system.backup_log, system.session_log	Backup/restore operation; authentication event.	Backup duration/failure trending; auth audit.
system.opentelemetry_span_log	Trace span, when OTel is enabled.	Cross-service distributed tracing including inter-server query spans.

Important: Log tables are ordinary MergeTree tables — they participate in merges, occupy disk, and can themselves become a performance problem if left unbounded. Give each a TTL in its server-config `<engine>` clause, and exclude them from cluster-wide parts alerts to avoid self-inflicted noise.

Tip: Every counter in `system.events` and every gauge in `system.metrics` carries a `description` column. `SELECT * FROM system.events WHERE event ILIKE '%cache%'` is self-documenting — the fastest way to discover instrumentation you did not know existed.

CHAPTER 3

Query Performance

Rank `query_log` five ways — duration, CPU, rows read, memory, frequency — and aggregate by `normalized_query_hash` so a pattern executed 10,000 times outranks a one-off. The same few patterns usually top several lists.

> Source: `system.query_log` (type = 'QueryFinish') · `system.processes` · `system.metrics.Query`

3.1 Headline query KPIs

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
QPS by kind	`query_log` counts per minute, split SELECT / INSERT	Load shape; the denominator for every rate judgment.	**At baseline**	**QPS flat but latency up** — resource side; QPS spike — find the new client in `query_log.client_name`.
Latency percentiles	`quantiles(0.5,0.95,0.99)(query_duration_ms)`	User experience per query class; averages hide everything.	**p95 stable**	**p99 detaching from p50** — a subclass regressed: segment by `normalized_query_hash` before touching anything.
Read amplification	`read_rows ÷ result_rows`	Rows scanned per row delivered — the ClickHouse analogue of access-path quality.	**Near partition/PK selectivity**	**Millions read for tens returned** — primary key or partition key not matching the predicate; §3.3.
Marks efficiency	ProfileEvents: `SelectedMarks`, `SelectedParts`, `SelectedRanges`	How well the sparse primary index prunes granules (~8,192 rows each).	**Marks ≪ total marks**	**All marks selected** — full scan; predicate not prefix-aligned with ORDER BY key, or cast defeats the index.
Memory per query	`memory_usage` in `query_log`/`processes`	Peak tracked allocation per query.	**≪ `max_memory_usage`**	**Queries near the limit** — Ch. 6: spill settings, GROUP BY cardinality, joins ordered large-side-left.
Concurrency	`system.metrics.Query`; `max_concurrent_queries`	Running queries vs. the admission limit.	**< 60% of limit**	**Pinned at limit** — pile-up: find the slow head-of-line query in `system.processes` by `elapsed`; Ch. 12 error 202.
Failed queries	Events: `FailedSelectQuery`, `FailedInsertQuery`; `query_log` type = 'ExceptionWhileProcessing'	Errors masquerading as slowness — retries multiply load.	**≈ 0**	**Rising failure rate** — group exceptions by code; cross-reference Ch. 12 error matrix.

3.2 Decomposing a query's time — key ProfileEvents

Each `query_log` row carries the per-query `ProfileEvents` map. These pairs decompose *where* the time went — the ClickHouse equivalent of a wait-time breakdown:

SIGNAL	PROFILEEVENTS KEYS	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
CPU vs. wall clock	`OSCPUVirtualTimeMicroseconds` vs. `RealTimeMicroseconds`	CPU-bound (ratio ≈ threads used) vs. waiting on something.	**Ratio ≈ threads**	**CPU ≪ real time** — the query waits: disk, network, or locks; the rows below say which.
Disk read wait	`DiskReadElapsedMicroseconds`, `OSReadBytes`, `OSReadChars`	Time in read syscalls; bytes from device vs. from page cache.	**Cache-served**	**OSReadBytes ≈ OSReadChars** — working set exceeds page cache: Ch. 8/9.
Network wait	`NetworkReceiveElapsedMicroseconds`, `NetworkSendElapsedMicroseconds`	Time shipping data to client or between servers.	**Small share**	**Send-dominated** — huge result sets or slow client; distributed queries — Ch. 11 shard skew.
Aggregation spill	`ExternalAggregationWritePart`, `ExternalSortWritePart`	GROUP BY / ORDER BY exceeded memory and went to disk.	**0 for hot queries**	**Hot pattern spilling every run** — raise memory or pre-aggregate (MV); Ch. 6 spill settings.
Pipeline hotspots	`system.processors_profile_log`: `elapsed_us`, `input_wait_us`, `output_wait_us` per processor	Which operator burned the time vs. waited for upstream/downstream.	**Reading dominates**	**One JOIN/aggregation processor dominating** — rewrite target identified; check join algorithm and table order.
Lock waits	`RWLockAcquiredReadLocks` timing, `ContextLock` waits	Rare in ClickHouse but visible under heavy DDL + query mix.	**≈ 0**	**Lock wait time visible** — DDL storms colliding with queries; serialize schema changes.
Query queue time	`QueryQueueWaitMicroseconds` (workload scheduling), async insert queue waits	Admission delay before execution began.	**≈ 0**	**Growing queue wait** — concurrency control engaged; fix the head-of-line load, not the limit.

Tip: For any single slow query, the fastest deep profile is `EXPLAIN indexes = 1` (shows partition pruning and PK ranges actually used), then a re-run with `send_logs_level = 'trace'` — the per-stage log lines name the granules read, the merge steps, and the aggregation method chosen. No external profiler required.

3.3 The recurring query anti-patterns

Ninety percent of ChistaDATA query-tuning findings are one of these six. Each is visible directly in the matrices above:

ANTI-PATTERN	OBSERVABILITY SIGNATURE	FIX
Predicate off the sorting key	All marks selected; read amplification in the millions; `EXPLAIN indexes=1` shows no PK ranges.	Reorder query or table: align ORDER BY key prefix with the dominant filter; add a data-skipping index or projection for secondary predicates.
Partition scan	`SelectedParts` ≈ all parts; partition key column absent from WHERE.	Filter on the partition key expression itself (same function, e.g. `toYYYYMM`); avoid wrapping it in casts.
`SELECT *` on wide tables	`read_bytes` huge while `read_rows` modest; column count in the pipeline high.	Project needed columns only — columnar storage makes this the cheapest big win; add PREWHERE for selective predicates on wide reads.
JOIN with the big table on the right	Join processor dominates in `processors_profile_log`; memory near limit; hash-table build huge.	Smaller relation on the right (build side); consider `join_algorithm = 'parallel_hash'` or dictionary/Join-engine lookups for dimensions.
`FINAL` on hot paths	Latency multiple of the same query without FINAL; merge-like CPU inside the read.	Design so reads tolerate duplicates (aggregate around versions with `argMax`), or schedule real OPTIMIZE windows; FINAL is a merge at query time.
High-cardinality GROUP BY	Spill events; memory ramps linearly during execution; p99 blowups on aggregation queries.	Pre-aggregate with a materialized view / projection; cap with `max_rows_to_group_by` + `group_by_overflow_mode` where approximation is acceptable.

Note: Aggregate `query_log` by `normalized_query_hash`, not raw query text — it collapses literals so "the same query with different dates" counts as one pattern. Attribute cost to patterns first, clients second (`client_name`, `http_user_agent`, `user`), and individual queries last.

With query behavior understood, the next question is whether the storage engine underneath them is healthy — the part and merge machinery that Chapter 4 instruments.

CHAPTER 4

MergeTree Parts, Merges and Mutations

Every insert creates a part; background merges consolidate them. When merges fall behind the write rate, parts accumulate — and everything degrades: reads touch more files, inserts get delayed, then rejected. This is the most common ClickHouse failure mode in the field.

> Source: `system.parts` (active) · `system.merges` · `system.mutations` · `system.part_log` · events `MergedRows`, `MergedUncompressedBytes`

4.1 Parts health

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Parts per partition	`system.parts` WHERE active, grouped by table, partition	The number every insert must eventually merge and every read must open.	**< 100–300**	**Approaching** `parts_to_delay_insert` — inserts get throttled, then refused at `parts_to_throw_insert` (error 252). Fix ingestion batching (Ch. 5) and merge capacity (§4.2).
Part size distribution	`system.parts`: `bytes_on_disk`, `rows`, `level`	Whether merges are actually consolidating (levels rising, sizes growing).	**Few large, few small**	**Thousands of tiny level-0 parts** — insert batches far too small; the merge tree is being fed sand.
Partition count & design	`system.parts` distinct partitions per table	Partitioning granularity — too fine multiplies parts and defeats merges.	**10s–100s per table**	**Daily/hourly partitions × high-cardinality key** — repartition to monthly (typical); partition key is for data lifecycle, not query pruning alone.
Detached & broken parts	`system.detached_parts`; prefix `broken_`	Parts set aside after checksum or corruption findings.	**0**	**Any broken parts** — investigate disk/memory errors on that host before re-attaching; recurring corruption is a hardware conversation.
Inactive parts residue	`system.parts` WHERE NOT active	Old parts awaiting cleanup after merges.	**Cleared in minutes**	**Accumulating** — cleanup thread starved or parts held by long-running queries/backups; check `old_parts_lifetime` and running operations.
Primary-key memory	`system.parts`: `primary_key_bytes_in_memory`, marks bytes	RAM the sparse indexes and marks consume per table.	**Modest vs. RAM**	**Gigabytes on one table** — index granularity too fine or PK columns too wide; affects every query's base memory (Ch. 6).

4.2 Merge and mutation machinery

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Active merges vs. pool	`system.metrics`: `Merge`, `BackgroundMergesAndMutationsPoolTask` vs. pool size	Merge concurrency against its ceiling.	**Headroom at peak**	**Pool pinned at size continuously** — merges can't keep up: more CPU/pool size, or less part creation (Ch. 5). Watch write amplification before raising pools.
Merge throughput	Events: `MergedRows`/s, `MergedUncompressedBytes`/s; `part_log` MergeParts durations	Consolidation rate vs. ingest rate — the balance that decides parts trend.	**≥ insert rate**	**Merge bytes/s < insert bytes/s sustained** — parts will grow without bound; scale merge capacity or cut ingest fan-out.
Largest running merges	`system.merges`: `progress`, `elapsed`, `total_size_bytes_compressed`, `memory_usage`	Giant merges monopolizing the pool for hours.	**Minutes-scale**	**Multi-hour merges recurring** — check `max_bytes_to_merge_at_max_space_in_pool`; on object storage, verify network throughput to the store (Ch. 8).
Merge failures	`part_log` WHERE error != 0; `text_log` merge exceptions	Failed merges retry forever and pin the queue.	**0**	**Recurring failure on one part set** — usually disk space (needs ~1.5× merge size free) or corruption; error 243 `NOT_ENOUGH_SPACE`.
Mutation backlog	`system.mutations`: count not done, `parts_to_do`, age	ALTER UPDATE/DELETE rewrite work outstanding — mutations rewrite whole parts.	**Empty or draining**	**Stuck mutation with `latest_fail_reason`** — fix the cause and `KILL MUTATION` the poison entry; everything behind it is blocked per table.
TTL & move activity	`part_log` MovePart; events `PartsWithAppendedFiles`; disk-move errors in `text_log`	Lifecycle machinery: TTL deletes/moves between volumes.	**Runs in windows**	**TTL merges dominating the pool** — TTL too aggressive or `merge_with_ttl_timeout` too low; lifecycle work is starving consolidation.

Important: Never run `OPTIMIZE TABLE … FINAL` as routine maintenance on large tables. It rewrites entire partitions through the merge pool, evicts the page cache, and competes with the very merges it is meant to help. It is a repair tool for specific situations (deduplication after incident, pre-FINAL-free reads), not a scheduled job.

Note: Mutations are not UPDATEs. Each mutation rewrites every affected part in the background; a mutation touching a 2 TB table is a 2 TB write. Frequent single-row mutations are an anti-pattern — design with versioned inserts (`ReplacingMergeTree` + `argMax` reads) instead, and reserve mutations for GDPR-style bulk corrections.

4.3 Storage efficiency

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Compression ratio per column	`system.columns`: `data_compressed_bytes ÷ data_uncompressed_bytes`	How well codecs fit the data; the biggest storage/IO lever in ClickHouse.	**5–20× typical**	**Near 1× on big columns** — wrong codec or entropy-heavy encoding: try `ZSTD`, `Delta/DoubleDelta` for timestamps/counters, `LowCardinality(String)` for enums-in-disguise.
Sorting-key locality	Compression ratio trend after resort; read amplification (Ch. 3)	ORDER BY key clusters similar values, compounding compression and pruning.	**Aligned with access**	**Low ratio + poor pruning together** — the sorting key serves neither storage nor queries; redesigning it is a rebuild, but usually the highest-value one available.
Wide vs. compact parts	`system.parts.part_type`	Small parts stored compact (one file) vs. wide (file per column).	**Compact only tiny**	**Large compact parts** — `min_bytes_for_wide_part` mis-set; column reads pay full-part I/O.
Projections & skipping indexes	`system.parts` projections size; events `SelectedMarks` deltas per index	Secondary structures earning (or not) their storage and merge cost.	**Used by hot queries**	**Projection never chosen** — verify with `EXPLAIN indexes=1`; unused projections still pay on every merge — drop them.
Disk headroom for merges	`system.disks` free vs. largest partition size	Merges need transient free space up to ~1.5× the merge size.	**> 20% free**	**< 15% free** — merges start failing before the disk is technically full; Ch. 8 capacity actions now.

Tip: The single most predictive parts-health query ChistaDATA runs on every audit:

```
SELECT table, partition, count() AS parts, sum(rows) AS rows
FROM system.parts WHERE active GROUP BY table, partition
ORDER BY parts DESC LIMIT 20;
```

Trending this per hour (from `system.parts` snapshots or `part_log`) shows whether the merge machinery is winning or losing days before error 252 appears.

Parts problems are almost always ingestion problems wearing a disguise. Chapter 5 instruments the write path itself.

CHAPTER 5

Ingestion and Inserts

ClickHouse wants few, large inserts. Every batch becomes a part; the write path's health is measured in batch sizes, delay/reject signals, and the fan-out cost of materialized views.

> Source: events `InsertedRows`, `InsertedBytes`, `DelayedInserts`, `RejectedInserts` · `system.part_log` (NewPart) · `system.asynchronous_inserts` · `system.query_views_log`

5.1 The insert path

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Insert rate	`InsertedRows`/s, `InsertedBytes`/s deltas	Ingest volume — the number merge throughput must beat (Ch. 4).	**At baseline**	**Step change up** — new producer or replay storm; find it via `query_log` INSERT clients before the parts alarm fires.
Rows per insert block	`part_log` NewPart `rows`; `InsertedRows` ÷ `InsertQuery`	Batch size actually reaching the table — the #1 ingestion health number.	**10k–500k+ rows**	**Hundreds of rows per part** — micro-batching: fix producers, enable async inserts (§5.2), or front with a buffer.
Delayed inserts	Event `DelayedInserts`; metric `DelayedInsertsThrottled`	Inserts slowed because a partition neared the parts threshold.	**0**	**Any sustained delays** — the early warning before rejects; act on Ch. 4 §4.1 now.
Rejected inserts	Event `RejectedInserts`; error 252 in `system.errors`	Inserts refused: too many parts.	**0**	**Non-zero** — active incident: pause producers if possible, widen partitions or raise merge capacity; never just raise `parts_to_throw_insert` and walk away.
Insert latency	`query_log` INSERT durations; quorum: `ReplicasMaxInsertsInQueue`	Producer-visible write latency, including replication quorum if enabled.	**Sub-second typical**	**Seconds+ with quorum on** — a lagging replica gates every insert; Ch. 10 replica health.
Deduplication activity	Events: `DuplicatedInsertedBlocks`	Replicated tables dedupe identical insert blocks (retries made safe).	**Low, on retries**	**High steady dedup** — producers double-sending; wasted bandwidth and Keeper traffic even though data stays correct.

5.2 Async inserts, materialized views, and streaming sources

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Async insert buffering	`system.asynchronous_inserts`; events `AsyncInsertQuery`, `AsyncInsertBytes`, `AsyncInsertCacheHits`	Server-side batching for many small producers: pending bytes and flush behavior.	**Flushes at size, not timeout**	**Every flush by timeout with tiny sizes** — `async_insert_max_data_size`/`busy_timeout` mis-tuned; you kept micro-batching, just moved it inside the server.
MV fan-out cost	`system.query_views_log`: per-view duration, rows, memory, exceptions	Each insert pays for every attached materialized view, synchronously.	**Views cheap vs. insert**	**One view dominating insert latency** — heavy JOIN or GROUP BY inside an MV; move to async refresh or restructure. An MV exception can fail the whole insert.
MV cascade depth	Schema review + `query_views_log` chains	MVs feeding MVs multiply write amplification invisibly.	**1 level, few views**	**Deep cascades** — each source row becomes N part writes; count total NewPart events per source insert to see the true multiplier.
Kafka / streaming lag	`system.kafka_consumers`: lag, `num_rebalance_assignments`, last exception; broker-side lag	Consumer health for Kafka-engine ingestion.	**Lag bounded**	**Lag climbing + rebalance churn** — consumer group flapping (often after a poison message or MV failure); check `text_log` for the parse exception before restarting.
Buffer-table flushes	`StorageBufferFlush` events; `system.tables` Buffer engines; background flush errors	Legacy small-insert absorber: rows lost on crash while buffered.	**Flushing on thresholds**	**Flush errors in `text_log`** — destination-table problems back up into the buffer silently; prefer async inserts for new designs.

Note: The ingestion golden rule — **one insert per table per second, as large as latency tolerates** — resolves most parts crises without any configuration change. When producers cannot batch, async inserts move the batching into ClickHouse; when they can, client-side batching remains strictly cheaper.

Tip: Attribute write amplification with `part_log`: NewPart events per minute per table, split by source (INSERT vs. view name in `query_views_log`). ChistaDATA audits regularly find a forgotten materialized view generating 4× the parts of the base table it watches.

CHAPTER 6

Memory

ClickHouse tracks allocations hierarchically — per query, per user, per server — and enforces limits at each level. The observability job is knowing who holds memory now, what the tracker believes vs. what the OS sees, and which limit a failing query actually hit.

> Source: metric `MemoryTracking` · `system.processes.memory_usage` · asynchronous metrics `OSMemoryAvailable`, `MemoryResident`, `CGroupMemoryUsed`, jemalloc.* · error 241

6.1 Server-level memory

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Tracked vs. limit	`MemoryTracking` vs. `max_server_memory_usage` (default: ratio 0.9 × RAM)	Total allocation ClickHouse accounts for, against its self-imposed ceiling.	**< 80% steady**	**Sawtooth at the ceiling** — queries being killed at server scope (241); find the holders in §6.2 before raising anything.
Tracked vs. resident	`MemoryTracking` vs. `MemoryResident` (RSS)	Tracker accuracy; the gap is allocator overhead/fragmentation.	**Gap < 10–15%**	**RSS ≫ tracked** — jemalloc fragmentation or untracked allocations; check `jemalloc.resident` vs. `allocated`; restart-level remedy, investigate workload churn.
OS / cgroup headroom	`OSMemoryAvailable`, `CGroupMemoryUsed` vs. cgroup limit	Distance from the OOM killer, which does not negotiate.	**Ample page cache left**	**Available shrinking toward zero** — on Kubernetes, the cgroup limit is the real ceiling: verify `max_server_memory_usage` derives from it, not the node's RAM.
Page cache share	OS free vs. cached; `OSReadChars` vs. `OSReadBytes` ratio (Ch. 8)	ClickHouse relies on the OS page cache as its main data cache.	**Large cached share**	**ClickHouse RSS crowding out page cache** — read latency degrades cluster-wide; lower the memory ratio, it usually pays for itself in I/O.
Fixed consumers	Marks/uncompressed/query caches (Ch. 9); PK memory (Ch. 4); `system.dictionaries.bytes_allocated`	The standing allocations that shrink the query budget.	**Sized deliberately**	**Dictionary bloat** — a "small lookup" grown to tens of GB is a recurring field finding; audit `bytes_allocated` per dictionary quarterly.

6.2 Query-level memory

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Top holders now	`system.processes` ORDER BY `memory_usage`	Who owns the memory this minute.	**Distributed**	**One query holding half the server** — decide: kill it, or let it finish and gate its pattern with per-user limits afterward.
Peak per pattern	`query_log.memory_usage` by `normalized_query_hash`	Which patterns are memory-shaped problems.	**Stable per pattern**	**Pattern trending up over weeks** — data growth meeting an unbounded GROUP BY/JOIN; fix before it crosses the limit in production hours.
Limit kills (241)	`system.errors` MEMORY_LIMIT_EXCEEDED; exception scope in message (query / user / total)	Which limit fired — the message names the scope that matters.	**Rare, query-scope**	**"total" scope kills** — innocent queries dying for a guilty server; the fix is the big holder or admission control, not the victim's settings.
Spill configuration	`max_bytes_before_external_group_by` / `_sort` vs. `max_memory_usage`	Whether big aggregations degrade gracefully to disk or die at the cliff.	**Spill ≈ 50–70% of limit**	**Spill unset with heavy analytics** — binary success/death at the limit; set both, watch Ch. 3 spill events for how often you pay the slow path.
Overcommit behavior	Memory overcommit settings; `QueryMemoryLimitExceeded` event; waiting queries	Soft limits let queries borrow headroom and picks a victim under pressure.	**Deliberate policy**	**Surprise kills under concurrency** — overcommit ratios left default while limits were tuned; align the two intentionally.
User/profile budgets	`max_memory_usage_for_user`; per-profile settings in `system.settings_profiles`	Fairness between workloads sharing the server.	**Ad-hoc gated**	**Dashboards and ad-hoc in one budget** — one analyst can starve production dashboards; split profiles (Ch. 14).

Note: Memory error messages are precise — read them fully. `Memory limit (for query) exceeded` and `Memory limit (total) exceeded` are different diseases: the first is a query problem, the second is a server-pressure problem where the reported query is often merely the last straw.

Tip: For a memory mystery with no obvious query, sample `system.trace_log` with `memory` trace type (memory profiler): it attributes allocations to stacks, turning "something ate 40 GB" into a function name. Enable temporarily via `memory_profiler_sample_probability`.

CHAPTER 7

CPU and Background Pools

ClickHouse will happily use every core you give it — the observability question is whether cores go to queries or to background machinery, and whether any thread pool has become the hidden bottleneck.

> Source: asynchronous metrics `LoadAverage*`, `OSUserTimeNormalized`, `OSSystemTimeNormalized` · metrics `GlobalThread*`, `Background*PoolTask` · per-query CPU ProfileEvents

7.1 CPU accounting

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Load vs. cores	`LoadAverage1/15` vs. `CPUCoresAvailable`… (cgroup-aware)	Run-queue pressure on the host or container.	**≤ cores**	**Sustained ≫ cores** — saturation: split user vs. system time next; in containers verify the cgroup quota, not node cores.
User vs. system split	`OSUserTimeNormalized`, `OSSystemTimeNormalized`, iowait	Query work vs. kernel overhead vs. waiting on disk.	**User-dominated**	**High system share** — syscall storms (tiny reads, network churn) or THP compaction; high iowait — Ch. 8.
CPU by query pattern	`OSCPUVirtualTimeMicroseconds` per `normalized_query_hash`	Which patterns own the cores.	**Flat profile**	**1–2 patterns dominating** — Chapter 3's offender list; a query fix beats any pool tuning.
CPU by subsystem	`metric_log`: query CPU vs. `MergedUncompressedBytes` activity windows; `system.merges.memory/threads`	Queries vs. merges vs. replication fetches competing for cores.	**Queries lead by day**	**Merge CPU dominating business hours** — ingest/merge design pushing background work into peak; Ch. 4–5, or schedule heavy ingest off-peak.
Per-query threads	`max_threads` setting; `query_thread_log` counts	Parallelism each query may claim — default is all cores.	**Sized per class**	**Dashboards at `max_threads` = cores** — three concurrent dashboards oversubscribe the box 3×; cap interactive profiles at 4–8 threads, keep batch high.

7.2 Thread pools — the hidden queues

Each background subsystem runs in its own bounded pool. A pool pinned at its size is a queue you are not seeing — work is waiting invisibly behind it:

POOL	METRICS (TASK VS. SIZE)	WHAT WAITS WHEN IT SATURATES	HEALTHY	RED FLAG → ACTION
Merges & mutations	`BackgroundMergesAndMutationsPoolTask` / `PoolSize`	Part consolidation; mutation progress.	**Headroom at peak**	**Pinned + parts rising** — the Ch. 4 emergency pattern; scale pool with CPU/IO to spare, else fix ingest.
Fetches	`BackgroundFetchesPoolTask` / `PoolSize`	Replica part downloads — replication delay grows.	**Headroom**	**Pinned + delay growing** — Ch. 10; also check network throughput between replicas.
Schedule pool	`BackgroundSchedulePoolTask` / `PoolSize`	Housekeeping: cleanup, TTL scheduling, Keeper retries, Buffer flushes.	**Low usage**	**Saturated** — cluster-wide weirdness (stale cleanup, lagging flushes) with no obvious cause; raise and find the flood source in `text_log`.
Moves	`BackgroundMovePoolTask` / `PoolSize`	TTL volume moves (hot→cold tiering).	**Bursty, drains**	**Backed up** — cold tier too slow for the data rate; tiering policy meets physics, Ch. 8.
Message broker	`BackgroundMessageBrokerSchedulePoolTask`	Kafka/NATS/RabbitMQ consumption.	**Matches consumers**	**Saturated** — streaming lag with idle CPU elsewhere; raise pool alongside consumer count.
Global thread pool	`GlobalThread`, `GlobalThreadActive`, `GlobalThreadScheduled`	Everything: thread creation stalls delay queries at start.	**Scheduled ≈ active**	**Scheduled ≫ active** — `max_thread_pool_size` ceiling reached; queries queue before their first instruction.

Note: Raising pool sizes is not free — every merge slot is CPU and disk bandwidth taken from queries. The order of operations is always: reduce the work (batch inserts, fix fan-out, right-size partitions) first, add capacity second, raise pool ceilings last. A bigger queue in front of the same disk is still the same disk.

Tip: `system.trace_log` with type `CPU` is a built-in sampling profiler. Aggregate by the top frame over a bad five minutes and you get a flamegraph-grade answer to "what were the cores actually doing" — merges, a specific aggregate function, decompression, or the network stack.

CHAPTER 8

Disk I/O and Object Storage

ClickHouse read performance is a page-cache story on local disks and a request-economics story on object storage. Instrument both layers — bytes from device vs. bytes from cache, and request counts vs. throttling on S3-compatible stores.

> Source: `system.disks` · events `OSReadBytes/Chars`, `DiskReadElapsedMicroseconds`, `S3*`, `CachedReadBuffer*` · asynchronous metrics per-disk free space

8.1 Local disks and the page cache

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Free space per disk	`system.disks`: `free_space` ÷ `total_space`	Capacity headroom — merges need transient 1.5× space (Ch. 4).	**> 20% free**	**< 15%** — merge failures begin; < 10% — table can go read-only. TTL/DETACH old partitions, add capacity, check log-table TTLs first.
Page-cache hit ratio	1 − (`OSReadBytes` ÷ `OSReadChars`) over deltas	Share of reads served from RAM — the dominant read-latency factor.	**> 90% hot data**	**Falling ratio** — working set outgrew RAM or RSS crowding (Ch. 6); latency degrades cluster-wide before any single query looks guilty.
Read latency	`DiskReadElapsedMicroseconds` ÷ read ops; host `iostat await`	Device-level latency as ClickHouse experiences it.	**< 1–2 ms SSD**	**» device spec** — saturated or degrading device; correlate with merge windows — merges are the usual bandwidth hog.
Write/fsync pressure	`WriteBufferFromFileDescriptorWriteBytes`, `DiskWriteElapsedMicroseconds`; fsync counts	Write bandwidth from inserts + merges combined (write amplification made visible).	**< 60% device max**	**Near ceiling** — merges and inserts fighting; separate volumes, or slow the amplification sources (Ch. 5 MV fan-out).
Storage policy balance	`system.parts` bytes per disk/volume; move failures in `text_log`	Whether multi-volume tiering distributes as designed.	**Matches policy**	**Hot volume full while cold idles** — `move_factor`/TTL move rules not keeping up; Ch. 7 move pool.

8.2 Object storage (S3-compatible) and the filesystem cache

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Request volume & mix	Events: `S3GetObject`, `S3PutObject`, `S3ListObjects` et al.	API call economics — object storage bills and throttles per request.	**Dominated by large GETs**	**List/small-GET storms** — tiny parts on S3 multiply requests; batch harder (Ch. 5) — parts problems cost real money here.
S3 latency & errors	`S3ReadMicroseconds`/requests; `S3RequestsErrors`, throttle/retry events	Store-side health: latency percentiles, 5xx/SlowDown responses.	**No throttling**	**Rising retries/SlowDown** — request-rate limits hit: spread key prefixes, reduce request count, or negotiate limits; retries silently multiply query latency.
Filesystem cache hits	`CachedReadBufferReadFromCacheHits` / `Misses`; cache size/used metrics	Local NVMe cache in front of remote storage — the difference between local-ish and remote latency.	**> 80–90% hot workloads**	**Low hit ratio** — cache too small for the hot set or churned by scans; size it to the hot partitions and exclude giant backfills from caching.
Remote read waits	`RemoteFSReadMicroseconds`, prefetch events (`ThreadpoolReaderTaskMicroseconds`)	Time queries spend waiting on remote reads, and whether prefetch hides it.	**Prefetch-hidden**	**Queries dominated by remote wait** — raise prefetch/read threads, verify NIC bandwidth to the store, and confirm the FS cache is actually enabled for the disk.
Zero-copy replication state	Config + Keeper metadata; fetch events staying low while parts appear	Replicas sharing one S3 copy — powerful, operationally sharp-edged.	**Understood & tested**	**Enabled by rumor** — deletion and Keeper-metadata edge cases have caused field incidents; enable only with a tested runbook and version-checked defaults.

Important: On object-storage-backed clusters, Chapter 4's parts discipline stops being a performance nicety and becomes the cost model: every part is objects, every merge re-writes them, every list is a billable call. A parts-per-partition alert threshold that was "advisory" on NVMe should be treated as hard on S3.

Note: Keep ClickHouse server logs, Keeper logs, and Keeper snapshots on storage *separate* from data volumes. A disk-full event on a shared volume takes down coordination and data at once — the difference between an incident and an outage.

CHAPTER 9

Caches

Beyond the OS page cache, ClickHouse maintains purpose-built caches. Each has a hit-ratio pair in `system.events` and a size knob; the audit is hit ratio vs. memory spent (Ch. 6 budget).

> Source: events `MarkCacheHits/Misses`, `UncompressedCacheHits/Misses`, `QueryCacheHits/Misses`, `CompiledExpressionCache*` · `system.dictionaries` · metrics `MarkCacheBytes`, `UncompressedCacheBytes`

CACHE	SIGNALS	WHAT IT CACHES / WHEN IT MATTERS	HEALTHY	RED FLAG → ACTION
Mark cache	`MarkCacheHits/Misses`; `MarkCacheBytes` vs. `mark_cache_size`	Index marks (granule offsets) — consulted by nearly every read; misses add disk reads before data reads even start.	**> 95% hits**	**Miss rate climbing** — cache smaller than active marks (many tables/parts); raise `mark_cache_size` — the highest-leverage cache setting in ClickHouse.
Uncompressed cache	`UncompressedCacheHits/Misses`, `UncompressedCacheBytes`	Decompressed blocks; helps only many small hot queries re-reading the same granules; off by default per-query.	**High if enabled**	**Enabled broadly with low hits** — RAM taken from page cache for nothing; enable per-profile for the specific hot-dashboard workload or not at all.
Query cache	`QueryCacheHits/Misses`; entry count/bytes	Full result sets for repeated identical queries (dashboards); opt-in per query/profile.	**High for dashboards**	**Low hits with cache on** — non-deterministic functions or per-user conditions defeat it; verify eligibility rules before spending memory.
Filesystem cache	Ch. 8 §8.2 hit metrics	Remote-disk data blocks on local NVMe — the object-storage page cache.	**> 80–90%**	**Cold** — see Ch. 8; on S3 clusters this cache is not optional for interactive latency.
Dictionaries	`system.dictionaries`: `status`, `bytes_allocated`, `last_exception`, load times	In-memory lookup tables used by `dictGet` in hot paths.	**LOADED, fresh**	**FAILED/stale dictionary** — queries silently degrade or error; alert on status != LOADED and on reload exceptions; watch allocation growth (Ch. 6).
Compiled expressions	`CompiledExpressionCacheHits/Misses`	JIT-compiled expression fragments for repeated query shapes.	**Warm after startup**	**Constant misses** — highly dynamic SQL shapes; harmless usually, but explains post-restart latency until warm.

Tip: Cache decisions compound with Chapter 6: every byte pinned in a ClickHouse cache is a byte the OS page cache cannot use. When in doubt, the page cache — which serves compressed data for all tables adaptively — is the better default owner of spare RAM; grow specific caches only against measured miss rates.

CHAPTER 10

Replication and Keeper

Replication health is the most alert-worthy surface in ClickHouse: a readonly replica or an expiring Keeper session turns into data-freshness and ingest incidents within minutes. Instrument both the replicas and the coordination service beneath them.

> Source: `system.replicas` · `system.replication_queue` · metrics `ReadonlyReplica`, `ZooKeeperSession`, `ZooKeeperWatch` · events `ZooKeeperTransactions`, `ReplicatedPartFetches` · Keeper `mntr`

10.1 Replica health

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Readonly replicas	Metric `ReadonlyReplica`; `system.replicas.is_readonly`	Replica lost its Keeper session or metadata — inserts to it fail.	**0, always**	**Any non-zero** — page-severity alert: check Keeper health (§10.2) first, then the replica's `last_queue_update_exception`; often recovers with session, else `SYSTEM RESTART REPLICA`.
Absolute delay	`system.replicas.absolute_delay`; asynchronous metric `ReplicasMaxAbsoluteDelay`	Seconds a replica trails the freshest data.	**< 10–60 s**	**Growing steadily** — fetch pool saturated (Ch. 7), network, or a giant merge replaying; distributed reads may silently serve stale data past `max_replica_delay_for_distributed_queries`.
Queue size & age	`queue_size`, `inserts_in_queue`, `merges_in_queue`, oldest entry time	Outstanding replication work per table.	**Near 0, draining**	**Old entries with high `num_tries`** — a stuck entry blocks everything behind it; read its `last_exception` and `postpone_reason` in `replication_queue` — the diagnosis is written there.
Fetch health	Events `ReplicatedPartFetches`, `ReplicatedPartFailedFetches`; `ReplicatedChecksumsFailed`	Part download success between replicas.	**Failures ≈ 0**	**Failed fetches / checksum mismatches** — network or source-replica corruption; checksum failures escalate to data-integrity investigation, not retry tuning.
Quorum & sync state	`future_parts`, `parts_to_check`; insert quorum settings + failures	Parts announced but not committed; quorum-gated insert health.	**Transient only**	**Persistent future/suspicious parts** — replica diverged after crash; `SYSTEM SYNC REPLICA`, then restore-replica procedures if it will not converge.

10.2 Keeper / ZooKeeper

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Session stability	Metric `ZooKeeperSession`; expiration events in `text_log`	Each expiration flips replicas readonly and aborts in-flight commits.	**No expirations**	**Any expiration** — Keeper-side pauses (disk, GC on ZK, leader change) or network; correlate timestamps with Keeper logs — this is the root of most "random readonly" mysteries.
Request latency	Events `ZooKeeperWaitMicroseconds` ÷ `ZooKeeperTransactions`; Keeper `mntr`: `zk_avg_latency`, `zk_max_latency`	Coordination round-trip cost — inserts to replicated tables pay it on the commit path.	**< 10 ms avg**	**Spikes/growth** — Keeper log/snapshot disk latency (put Keeper on its own fast disk), leader saturation, or snapshot pauses.
Outstanding requests	Keeper `mntr`: `zk_outstanding_requests`; pending in `system.zookeeper_connection`	Queue depth at the coordination service.	**≈ 0**	**Persistent queue** — Keeper CPU or fsync-bound; check its host like any database host — it is one.
Znode/watch volume	`mntr`: `zk_znode_count`, `zk_watch_count`; `system.zookeeper` subtree sizes	Metadata scale — grows with tables × partitions × replicas and with parts churn.	**Stable, bounded**	**Explosive znode growth** — parts storms (Ch. 4) or too many replicated tables with tiny partitions; Keeper memory and snapshot times scale with it.
Leader changes	Keeper `mntr` role flips; elections in Keeper logs	Elections pause all coordination briefly.	**Rare, deliberate**	**Recurring elections** — flaky node or network partitions in the ensemble; fix quorum health before touching ClickHouse.
DDL queue	`system.distributed_ddl_queue`: pending, age, per-host status	`ON CLUSTER` operations waiting for every host to acknowledge.	**Empty**	**Stuck entries** — one dead/wedged host blocks cluster DDL; finish or remove the entry, fix the host, and gate future DDL on cluster health.

Important: Treat Keeper as tier-zero infrastructure: dedicated hosts (or at minimum dedicated disks), no co-location with data-heavy ClickHouse nodes, and the same monitoring rigor as the database itself. In ChistaDATA incident reviews, "Keeper on a shared, busy disk" is the most common root cause behind cluster-wide replication incidents.

CHAPTER 11

Distributed Processing and Network

Distributed tables fan queries out and (optionally) buffer inserts per shard. The failure modes are skew — one shard doing all the work — and silent backlog in the async insert queues.

Source: `system.distribution_queue` · metrics `TCPConnection`, `HTTPConnection`, `InterserverConnection` · events `DistributedSend`, network elapsed · `system.clusters`

METRIC	SOURCE	WHAT IT TELLS YOU	HEALTHY	RED FLAG → ACTION
Distributed insert backlog	`system.distribution_queue`: pending files/bytes, `is_blocked`, broken files	Async inserts queued on the initiator per shard.	**≈ 0, draining**	**Growing or broken files** — a shard down or a poison batch: read the error, fix or move the broken file; data in this queue is not yet on any shard.
Sync vs. async inserts	`insert_distributed_sync` setting; queue behavior	Whether producers wait for shard acknowledgment or trust the queue.	**Deliberate choice**	**Async by default, unmonitored** — the queue is a durability gap on initiator loss; either go sync (and size for latency) or alert on queue depth.
Shard skew	Per-shard `read_rows`/bytes from distributed `query_log`; per-host `metric_log`	Whether shards contribute equally to reads and hold data equally.	**Even ±20%**	**One shard hot** — sharding key skew (a mega-tenant) or topology drift; the slowest shard sets every distributed query's latency.
Connections	Metrics `TCPConnection`, `HTTPConnection`, `InterserverConnection` vs. limits	Client and inter-server connection load.	**Below limits**	**At `max_connections`** — connect storms or missing client pooling; also watch per-user limits (Ch. 12 error 202).
Inter-server transfer	`NetworkSendElapsedMicroseconds/Receive...` on remote queries; bytes over interserver port	Data volume shipped between nodes for distributed JOINs/GROUP BYs.	**Aggregated-then-shipped**	**Huge transfers per query** — query shape ships raw rows (e.g. GLOBAL JOIN of big tables); push aggregation down, co-locate joins with the sharding key.
Hedged / retried requests	Events for hedged connections, connection failures per replica	Initiators failing over between replicas mid-query.	**Rare**	**Frequent hedging** — a flaky replica making every query pay a timeout; find it in `system.clusters.errors_count` and drain it.

Note: In distributed troubleshooting, always determine *which host's* numbers you are reading. `clusterAllReplicas('cluster', system.metric_log)` and friends let you query every node's system tables from one seat — the difference between cluster observability and single-node guesswork.

CHAPTER 12

Errors and Saturation Signals

`system.errors` is the cheapest anomaly detector in ClickHouse: every error code ever raised, with count and last occurrence. Alert on rate, triage by code. These are the codes that matter operationally:

> Source: `system.errors` · `system.text_log` (level ≤ Error) · `system.crash_log` · events `FailedQuery`, `FailedInsertQuery`

ERROR (CODE)	WHAT IT MEANS	WHERE TO GO
`MEMORY_LIMIT_EXCEEDED` (241)	A query/user/server memory ceiling fired — read the scope in the message.	Chapter 6 — query-scope: fix the pattern; total-scope: find the real holder or admission-control.
`TOO_MANY_PARTS` (252)	A partition crossed `parts_to_throw_insert`; inserts refused.	Chapters 4–5 — the full parts playbook: batching, partition design, merge capacity.
`TOO_MANY_SIMULTANEOUS_QUERIES` (202)	Concurrency admission limit reached (server, user, or profile scope).	Chapter 3 §3.1 — find the head-of-line blocker; raise limits only for proven capacity.
`TIMEOUT_EXCEEDED` (159)	`max_execution_time` hit; may also surface as socket timeouts on clients.	Chapter 3 — usually a regression symptom, not a timeout-tuning problem.
`NOT_ENOUGH_SPACE` (243)	A merge or insert could not reserve disk.	Chapters 4/8 — merges need 1.5× transient space; check per-disk free, not cluster average.
`TABLE_IS_READ_ONLY` (242) / `KEEPER_EXCEPTION` (999)	Replica in readonly / coordination failure.	Chapter 10 — Keeper session health first, replica restart second.
`NETWORK_ERROR` (210) / `SOCKET_TIMEOUT` (209)	Connectivity failures between nodes or to clients.	Chapter 11 — pair with retransmit stats at the OS; a flaky NIC looks like "random" query failures.
`CHECKSUM_DOESNT_MATCH` (40)	Part data failed verification on read or fetch.	Chapter 10 §fetches — treat as data-integrity: isolate the host, check hardware (RAM/disk).

SATURATION SIGNAL	SOURCE	MEANING	HEALTHY	RED FLAG
Error-log rate	`text_log` level ≤ Error per minute	The server narrating its own problems.	**≈ baseline noise**	**Step change** — read the top messages before any dashboard; ClickHouse log lines are unusually specific.
Crash log	`system.crash_log`	Fatal signals with stacks.	**Empty, always**	**Any row** — capture and version-match against known issues before restarting into the same crash.
Startup warnings	`system.warnings`	Misconfigurations the server itself flags at boot.	**Empty**	**Ignored warnings** — free audit findings; clear them as config hygiene (Ch. 14).

CHAPTER 13

Monitoring Integration and the Starter Alert Set

Everything in Chapters 3–12 exports cleanly. The reference stack ChistaDATA deploys: the built-in Prometheus endpoint for metrics, `query_log`/`metric_log` for forensics, Grafana on top, and a small, opinionated alert set that pages on state and tickets on trend.

13.1 Export paths

PATH	HOW	USE FOR
Embedded Prometheus endpoint	`<prometheus>` server config: port, expose metrics + events + asynchronous_metrics + errors	The default: every gauge/counter in this book scraped directly, no exporter sidecar needed.
System-table dashboards	Grafana ClickHouse datasource querying `system.*` and `metric_log`	Anything per-table or per-query-pattern (parts per partition, top patterns) — richer than pre-aggregated metrics.
Built-in `/dashboard`	HTTP endpoint over `system.dashboards`	Zero-install triage UI; first look at an unfamiliar cluster.
Logs & traces	`text_log` to your log pipeline; `opentelemetry_span_log` for tracing	Error-rate alerting; end-to-end latency attribution across services.

13.2 The golden dashboard set

Six panels answer the four-question triage of Chapter 1 at a glance: **(1)** QPS + p50/p95/p99 by query kind; **(2)** running queries and memory tracking vs. limits; **(3)** parts per partition (top tables) + merge pool saturation; **(4)** insert rate, delayed/rejected inserts; **(5)** max replication delay + readonly replicas + Keeper latency; **(6)** host vitals — load vs. cores, page-cache hit ratio, disk free, disk latency. Everything else is drill-down.

> **Note:** Scrape at 15–30 s intervals; keep `metric_log` at its 1 s default regardless. Prometheus answers "is it healthy now"; `metric_log` answers "what exactly happened at 03:12:47" — you need both, and they cost almost nothing together.

13.3 Starter alert thresholds

Page = wake a human now; Ticket = fix this week. Calibrate values to your baseline after two weeks of data.

ALERT	EXPRESSION (CONCEPT)	SEVERITY	RATIONALE / CHAPTER
Readonly replica	`ReadonlyReplica > 0` for 2 min	**Page**	Inserts failing on that replica; Keeper or metadata issue. Ch. 10.
Replication delay	`ReplicasMaxAbsoluteDelay > 300 s` for 5 min	**Page**	Stale reads and growing recovery debt. Ch. 10.
Keeper session expirations	expirations > 0 in 10 min	**Page**	Each one flips replicas readonly. Ch. 10 §10.2.
Rejected inserts	`RejectedInserts` delta > 0	**Page**	Error 252 is already user-visible. Ch. 4–5.
Disk free	any disk < 15% (page < 10%)	**Ticket/Page**	Merges fail below ~15%; readonly risk below 10%. Ch. 8.
Memory pressure	`MemoryTracking > 85%` of limit for 10 min	**Ticket**	241-kill territory approaching. Ch. 6.
Parts per partition	max > 60% of `parts_to_delay_insert`	**Ticket**	Days of warning before the 252 page. Ch. 4.
Delayed inserts	`DelayedInserts` delta > 0 sustained 15 min	**Ticket**	The early-warning stage of the parts story. Ch. 5.
Merge pool saturation	pool task = pool size for 30 min	**Ticket**	Invisible queue forming. Ch. 7 §7.2.
Failed queries rate	`FailedQuery` delta > N/min vs. baseline	**Ticket**	Errors before latency: retries multiply load. Ch. 12.
Crash log	any new row in `crash_log`	**Page**	Always. Ch. 12.
Mutation stuck	mutation age > 1 h with fail reason	**Ticket**	Blocks all later mutations on the table. Ch. 4 §4.2.
Distributed queue backlog	pending bytes > threshold or broken > 0	**Ticket**	Undelivered data on the initiator. Ch. 11.
p95 latency regression	p95 > 2× 7-day baseline for 15 min	**Ticket**	Catch-all for what the specific alerts miss. Ch. 3.

Tip: Alert on *state* (readonly, rejected, crash) with pages and on *trend* (parts, delay, memory) with tickets. Clusters that page on trends train their operators to ignore pages — the most expensive monitoring failure mode there is.

CHAPTER 14

Configuration Audit Checklist

The settings worth explicitly signing off in every ClickHouse audit — server level first, then query profiles. Compare against `system.server_settings`, `system.settings` (with `changed = 1`), and `system.merge_tree_settings`.

14.1 Server and MergeTree settings

SETTING	CONTROLS	STARTING POINT	AUDIT NOTE
`max_server_memory_usage_to_ram_ratio`	Server memory ceiling as share of RAM/cgroup.	**0.8–0.9**	Verify it derives from the **cgroup** limit on Kubernetes; leave real room for page cache (Ch. 6/8).
`mark_cache_size`	Mark cache (Ch. 9).	**≥ 5 GB; size to marks**	Most undersized setting on busy clusters with many tables/parts.
`max_concurrent_queries`	Admission control.	**100–200**	A protection mechanism, not a performance knob — raising it under pile-up makes pile-ups worse (Ch. 12, error 202).
Background pool sizes merges, fetches, schedule, moves	Concurrency of all background machinery (Ch. 7 §7.2).	**Defaults, then evidence**	Raise only against measured pool saturation with CPU/disk headroom to spend.
`parts_to_delay_insert` / `parts_to_throw_insert`	The parts backpressure thresholds (Ch. 4).	**Defaults**	Raising them without fixing ingestion converts backpressure into a bigger, later outage — a recurring field anti-pattern.
`max_bytes_to_merge_at_max_space_in_pool`	Largest merge the scheduler will attempt.	**Default (~150 GB)**	Lower on small disks (1.5× rule); check against Ch. 4 giant-merge findings.
Log-table config query_log, part_log, metric_log, text_log	The flight recorder (Ch. 2 §2.2).	**All on, TTL 14–30 d**	Missing `part_log` or TTLs is a fail — you cannot debug backward without them.
Prometheus endpoint	Metrics export (Ch. 13).	**Enabled, all sections**	Confirm `errors` and `asynchronous_metrics` are included in the exposition.

14.2 Profiles, users, and the host

SETTING	CONTROLS	STARTING POINT	AUDIT NOTE
`max_memory_usage` **(per profile)**	Per-query memory ceiling.	**10–30 GB analytics**	Different per workload class; one global value means dashboards and batch fight in the same budget (Ch. 6).
`max_bytes_before_external_group_by` / `_sort`	Graceful spill to disk before the memory cliff.	**≈ 50–70% of limit**	Unset on analytics profiles = binary success/death; watch Ch. 3 spill events for cost.
`max_threads` **(per profile)**	Per-query parallelism.	**4–8 interactive; high batch**	Default = all cores; three dashboard users can oversubscribe the host 3× (Ch. 7).
`max_execution_time`	Runaway guard.	**Set per class**	Alerting on 159 rates (Ch. 12) tells you when limits are masking regressions.
`join_algorithm`, `insert_distributed_sync`	Join strategy; distributed insert durability.	**Deliberate**	Both are "changed once during an incident, never revisited" candidates — every non-default needs a stated reason.
Readonly & quotas for ad-hoc users	Blast-radius control for humans.	**Quotas on**	An analyst with server-default limits is the most common "mystery load" source in field engagements.

14.3 Host checklist

ITEM	TARGET	WHY
Swap	**Off (or swappiness ≤ 1)**	Swapping a columnar database is strictly worse than killing a query.
CPU governor / turbo	**performance**	Latency jitter under powersave shows up directly in p99.
Filesystem & mount	**ext4/xfs, noatime**	atime updates add writes on every read path.
Clock sync	**chrony/NTP healthy**	Replicated-log ordering and cross-node forensics depend on it.
ulimits (nofile)	**≥ 500k for the service**	Parts × columns × connections = file descriptors; exhaustion looks like random I/O errors.
Keeper placement	**Dedicated hosts/disks**	Ch. 10 — the most common cluster-wide root cause in our incident reviews.

Note: `SELECT * FROM system.settings WHERE changed` is the audit in one query: every deviation from defaults, with current values. Ask for the reason behind each row — "nobody remembers" is itself a finding, and usually the oldest ones date to an incident three versions ago.

CHAPTER 15

Troubleshooting Matrix

Symptom → likely causes (ordered by field frequency) → first checks. First move in every incident: capture `system.processes`, the last 30 minutes of `metric_log`, and `system.errors` ordered by `last_error_time` — then diagnose.

SYMPTOM	LIKELY CAUSES (ORDERED)	FIRST CHECKS	RESOLUTION PATH
SELECTs slow, cluster-wide	1. Page-cache hit ratio fell (working set / RSS creep) 2. Merge storm eating CPU/IO 3. Concurrency pile-up behind one heavy query 4. Mark cache thrash	Cache ratio (Ch. 8); `system.merges`; `processes` by elapsed; `MarkCacheMisses` delta	Kill/gate the head-of-line query; if cache-driven, restore page-cache headroom (Ch. 6) before touching query settings.
One query pattern slow	1. Predicate off the sorting key 2. Partition scan 3. JOIN order/algorithm 4. Spill to disk began as data grew	`EXPLAIN indexes=1`; read amplification and spill events for its `normalized_query_hash` (Ch. 3)	Fix key alignment or add projection/skipping index; pre-aggregate with an MV if it is a dashboard query.
INSERTs delayed or failing (252)	1. Micro-batch producers 2. Merge capacity behind 3. Partition key too fine 4. MV fan-out multiplying parts	Rows/part in `part_log`; parts per partition; merge pool saturation; `query_views_log`	Batch (client or async inserts); repartition coarser; scale merge pool only with hardware headroom. Never just raise the throw threshold.
Replication delay growing	1. Fetch pool saturated 2. Network between replicas 3. One giant merge replaying 4. Keeper latency gating the queue	`system.replicas` queue ages; fetch pool metrics; `replication_queue.postpone_reason`; Keeper latency	The queue entry's exception text names the blocker; fix that entry — do not blind-restart replicas ahead of diagnosis.
Replica readonly	1. Keeper session expired 2. Keeper quorum unhealthy 3. Replica metadata diverged after crash	Session expirations in `text_log`; Keeper `mntr`; `replicas.last_queue_update_exception`	Fix Keeper first (Ch. 10 §10.2); then `SYSTEM RESTART REPLICA`; restore-replica procedure only if it will not converge.

SYMPTOM	LIKELY CAUSES (ORDERED)	FIRST CHECKS	RESOLUTION PATH
Memory-limit kills (241)	1. One unbounded GROUP BY/JOIN 2. Total-scope pressure from many medium queries 3. Fixed consumers grew (dictionary, caches) 4. Fragmentation gap	Error scope in message; `processes` top holders; dictionary bytes; tracked-vs-RSS gap (Ch. 6)	Query-scope: fix the pattern or set spill. Total-scope: gate concurrency or shrink fixed consumers — the reported query is often just the last straw.
OOM kill (process dies)	1. Cgroup limit below server ceiling 2. Tracker gap (untracked allocations) 3. Co-tenant process on the host	`dmesg`/journal OOM record; `CGroupMemoryUsed` history in `asynchronous_metric_log`; RSS vs. tracked	Align `max_server_memory_usage` with the real cgroup limit minus page-cache headroom; isolate co-tenants.
High CPU, low QPS	1. Merge/mutation storm 2. One scan-heavy pattern 3. Decompression churn (bad codec/locality) 4. High system time (THP, syscall storms)	`trace_log` CPU sampling top frames; merge activity; CPU per pattern; user/system split (Ch. 7)	The profiler names it in minutes: merges → Ch. 4; a pattern → Ch. 3; kernel → host tuning (Ch. 14 §14.3).
Disk filling fast	1. Log tables without TTL 2. Inactive parts not clearing 3. Compression regressed after schema change 4. TTL moves/deletes stopped	Size by table incl. `system.*`; inactive parts age; per-column ratios (Ch. 4 §4.3); move-pool errors	The offender is almost always visible in a size-by-table query; fix TTLs and lifecycle machinery before adding disk.
Queries pile up (202)	1. One slow head-of-line query holding slots 2. Client retry storm 3. Genuine load growth	`processes` by elapsed; QPS by client; failed-then-retried patterns in `query_log`	Kill the blocker and gate its pattern; fix client retry policy (no retry on timeout without backoff); capacity only for cause 3.
Slow after upgrade	1. Setting default changed 2. Query plan/analyzer behavior change 3. Cache-cold restart effects judged too early	Diff `system.settings WHERE changed` across versions; release notes; compare `query_log` pattern metrics before/after	Pin the regressed pattern with A/B settings (`SETTINGS` clause) to isolate the changed default; report upstream if genuine.

Important: Restarting a ClickHouse node destroys the two things you need most — `system.processes` state and cumulative counters — while log tables survive. Capture first, restart second. The 60-second evidence kit: `processes`, `errors`, last-30-min `metric_log` aggregates, `replicas` summary, and parts-per-partition top 20.

APPENDIX A

The Field Query Pack

The collection queries referenced throughout this book — also automated by ChistaDATA's open-source auditor, `chaudit`. Run A1–A3 in the first minute of any incident.

```
-- A1 · What is running right now (the kill list)
SELECT query_id, user, elapsed, formatReadableSize(memory_usage) AS mem,
       read_rows, substring(query, 1, 90) AS q
FROM system.processes ORDER BY elapsed DESC;
```

```
-- A2 · Errors since start, most recent first
SELECT name, code, value, last_error_time, substring(last_error_message, 1, 120) AS msg
FROM system.errors ORDER BY last_error_time DESC LIMIT 20;
```

```
-- A3 · Parts per partition — the merge-health headline (Ch. 4)
SELECT database, table, partition, count() AS parts, sum(rows) AS rows,
       formatReadableSize(sum(bytes_on_disk)) AS size
FROM system.parts WHERE active
GROUP BY database, table, partition ORDER BY parts DESC LIMIT 20;
```

```
-- A4 · Top query patterns by cost, last 24 h (Ch. 3; re-order by any column)
SELECT normalized_query_hash AS h, count() AS execs,
       quantile(0.95)(query_duration_ms) AS p95_ms, sum(read_rows) AS rows_read,
       formatReadableSize(max(memory_usage)) AS peak_mem, any(substring(query,1,80)) AS sample
FROM system.query_log
WHERE type = 'QueryFinish' AND event_time > now() - INTERVAL 1 DAY
GROUP BY h ORDER BY rows_read DESC LIMIT 20;
```

```
-- A5 · Replication health, worst first (Ch. 10)
SELECT database, table, is_readonly, absolute_delay, queue_size,
       inserts_in_queue, merges_in_queue, last_queue_update_exception
FROM system.replicas ORDER BY absolute_delay DESC LIMIT 20;
```

```
-- A6 · Cache hit ratios from event deltas (Ch. 8–9; run twice, diff, or use metric_log)
SELECT event, value FROM system.events
WHERE event IN ('MarkCacheHits','MarkCacheMisses','OSReadBytes','OSReadChars',
                'CachedReadBufferReadFromCacheHits','CachedReadBufferReadFromCacheMisses');
```

```
-- A7 · Storage & compression audit per column (Ch. 4 §4.3)
SELECT table, name,
       formatReadableSize(data_compressed_bytes)  AS comp,
       round(data_uncompressed_bytes / nullIf(data_compressed_bytes,0), 1) AS ratio
FROM system.columns WHERE database NOT IN ('system')
ORDER BY data_compressed_bytes DESC LIMIT 25;
```

ChistaDATA

Observe deeply. Diagnose precisely. Fix what the evidence names.

ClickHouse tells you almost everything about itself — if you know which of its thousands of signals to read, what healthy looks like, and which subsystem a bad number indicts. This book is the matrix ChistaDATA engineers use to do exactly that: the system-table map, per-subsystem metric matrices with healthy targets and red flags, a starter alert set, a configuration audit checklist, and the symptom-driven troubleshooting matrix refined across production engagements.

About ChistaDATA Inc.

ChistaDATA Inc. provides enterprise-class 24×7 ClickHouse consultative support and managed services — performance audits and health checks, architecture and capacity engineering, streaming and real-time analytics platforms, and around-the-clock managed operations for ClickHouse fleets on bare metal, Kubernetes, and object storage. The matrices in this book power `chaudit`, ChistaDATA's open-source ClickHouse performance auditor, available from the ChistaDATA GitHub organization.

Engage ChistaDATA

If your ClickHouse estate needs a performance audit, an observability build-out to the standard in Chapter 13, or a second pair of eyes on a stubborn incident — our engineers run the exact methodology in this book, from evidence capture to verified remediation, with findings ranked by impact.

chistadata.com

ChistaDATA Technical Series · First Edition · July 2026