An Architectural Analysis of the ClickHouse OLAP Database Management System

Introduction: The Architectural Philosophy of a High-Performance OLAP DBMS

The design and implementation of any high-performance database system are fundamentally a response to a specific set of computational and business challenges. ClickHouse, an open-source Online Analytical Processing (OLAP) Database Management System (DBMS), is no exception. Its architecture is meticulously engineered to address five key challenges that define modern, large-scale analytical data management: the ability to handle enormous and rapidly growing datasets with high ingestion rates; the capacity to execute many simultaneous queries with low latency; seamless integration with diverse landscapes of data stores and formats; the provision of an expressive and convenient SQL dialect with sophisticated performance introspection tools; and the assurance of industry-grade robustness and versatile deployment options.[1] To meet these demanding requirements, ClickHouse is built upon a set of core architectural principles that permeate every layer of its design, from data storage to query execution.

The Columnar Paradigm as a First Principle

At its core, ClickHouse is a true column-oriented DBMS.[2] This is not merely a storage optimization but the foundational axiom from which its performance characteristics are derived. In a columnar system, data is not stored in rows but by columns. All values for a single column are stored contiguously on disk. This approach has profound implications for analytical queries, which typically access a subset of columns from a wide table. By storing data column by column, the system can read only the required columns from disk, dramatically reducing I/O compared to a row-oriented database that would have to read entire rows to access the few necessary values.[3] This principle extends beyond physical storage into in-memory processing. During query execution, data is processed in arrays, or vectors, of column data, a structure that directly enables the system's primary performance mechanism: vectorized query execution.[2]

Vectorized Query Execution: The Engine of Performance

ClickHouse employs a vectorized query execution model, a technique that is central to its high performance.[2] Instead of processing data one value at a time in a tight loop (e.g.,

for each row, do operation), operations are dispatched on arrays or "chunks" of column data.[2] This batch-oriented processing model significantly improves CPU efficiency in several ways. It dramatically reduces the overhead of function call dispatches, as a single function call can operate on thousands of values. More importantly, it allows the system to leverage modern CPU architectures, particularly Single Instruction, Multiple Data (SIMD) capabilities.[3] SIMD instructions allow a single CPU instruction to be applied to multiple data points simultaneously, leading to massive parallelism at the hardware level. This approach ensures that the CPU's computational power, not I/O or function call overhead, is the primary factor in query execution speed.

System Layering and Core Abstractions

The ClickHouse engine is logically divided into three main layers: the storage layer, the query processing layer, and the integration layer.[1] These layers interact through a set of core in-memory data abstractions. The primary abstraction for column data is the

IColumn interface, which represents a chunk of a column in memory.[2] Operations on

IColumn objects are typically immutable; they create a new, modified column rather than altering the original. While the system is optimized for columnar operations, it also provides a mechanism to work with individual values through the Field object, a discriminated union that can hold various scalar types. However, this is deliberately inefficient for bulk processing and is used sparingly.[2] During query execution, chunks of columns are grouped into a container called a

Block, which serves as the unit of data that flows through the query processing pipeline.[3]

A defining characteristic of ClickHouse's design is its philosophy of "leaky abstractions".[2]
While the

IColumn interface provides generic methods for data manipulation, the system encourages performance-critical functions to bypass this generic interface. Developers are expected to cast an IColumn to its specific implementation (e.g., ColumnUInt64) and operate directly on its internal, contiguous memory array. This deliberate design choice sacrifices strict software engineering encapsulation for raw performance. It is this "leakiness" that allows specialized routines to be written that can fully exploit the underlying memory layout and CPU capabilities, such as SIMD instructions. This reveals a core aspect of the system's philosophy: ClickHouse is engineered to be as close to the hardware as possible, functioning not just as a high-level OLAP system but as a performance-engineering framework for data processing. This design choice results in a higher performance ceiling but also a steeper learning curve for developers contributing to the system, as it requires a deep understanding of low-level optimization techniques.

Deployment Philosophy: A Self-Contained Native Binary

Further reinforcing its focus on bare-metal performance, ClickHouse is built in C++ and distributed as a single, statically-linked binary with no external runtime dependencies.[1] This simplifies deployment and, critically, avoids the performance overheads associated with managed runtimes like the Java Virtual Machine (JVM), such as garbage collection pauses and just-in-time compilation latencies. This design allows ClickHouse to deliver predictable, high performance on any hardware, from a developer's laptop to a massive server cluster, fulfilling its mandate of versatile and robust deployment.[1]

Section 1: The Storage Layer - A Deep Dive into the MergeTree Engine Family

The MergeTree engine family is the cornerstone of ClickHouse's data storage and retrieval capabilities, providing the features necessary for resilience, high-performance data retrieval, and high-volume ingestion.[4] These engines are the most robust and commonly used in ClickHouse, designed to handle the immense scale and velocity of modern analytical workloads.[5] Their architecture is fundamentally based on the principles of a Log-Structured Merge-Tree (LSM Tree), a design that optimizes for write throughput by treating inserts as append-only operations.

1.1 The Core MergeTree Engine: An LSM-Tree for Analytics

The base MergeTree engine is the default choice for single-node ClickHouse instances due to its versatility and practicality.[4] Its design is centered around the concept of immutable data parts. When data is inserted into a

MergeTree table, it is written to a new, self-contained directory on disk known as a "data part".[6] This process is extremely fast because it is an append-only operation, avoiding the costly read-modify-write cycles common in transactional databases.

Over time, as numerous small inserts create many small data parts, a background process asynchronously and continuously merges these smaller parts into larger, more efficient ones.[6] This merging process is the origin of the name

MergeTree. It compacts the data, improves compression, and keeps the number of files on disk manageable. Once a merge is complete, the original smaller parts are marked as inactive and are deleted after a configurable interval.[6] This LSM-Tree approach defers the expensive work of data organization to the background, thereby sustaining very high ingestion rates. To minimize the overhead of creating and merging an excessive number of small parts, it is a critical best practice to insert data in large batches (e.g., tens of thousands of rows at once) or to use asynchronous inserts, which buffer data on the server side before creating a part.[6]

The lifecycle of a data part involves several steps upon insertion: the rows are first sorted by the table's sorting key, then split into individual columns, each of which is compressed and written to a binary file within the new data part directory.[6] This self-contained nature means each part includes all the metadata necessary for its interpretation, including indexes, column statistics, and checksums.[6]

1.2 Physical Data Organization: From Partitions to Granules

Data within a MergeTree table is organized in a multi-level hierarchy to facilitate efficient management and querying.

Partitions

At the highest level, a table's data can be logically divided into partitions using the PARTITION BY clause in the CREATE TABLE statement.[7] This clause can accept an arbitrary expression, though it is most commonly used with a date or timestamp column to partition data by month, day, or another time-based interval (e.g.,

PARTITION BY toYYYYMM(event_date)).[5] Partitioning serves two primary purposes. First, it is a powerful tool for data lifecycle management; operations like

DROP PARTITION are extremely fast metadata operations that simply delete a directory, making it easy to implement retention policies for aging out old data.[8] Second, it is a crucial mechanism for query optimization. If a query's

WHERE clause contains a filter on the partitioning key, the query planner can perform "partition pruning," completely ignoring all partitions (and their corresponding directories on disk) that do not match the filter, thus drastically reducing the scope of the data scan.[5]

Data Parts

Within each partition directory, data is stored in one or more data parts. As described previously, each part is a directory containing all the data and metadata for a specific batch of inserted rows.[6] Parts belonging to different partitions are never merged together.[5]

Columns and Compression

Inside a data part's directory, the columnar storage principle is physically realized. Each column of the table is stored in its own separate, compressed binary file (typically with a .bin extension).[6] This physical separation is what allows ClickHouse to read only the specific columns required by a query. ClickHouse also supports a

Compact part format in addition to the default Wide format. In the Compact format, all columns are stored in a single file. This format is designed to increase the performance of small, frequent inserts by reducing the number of files that need to be opened and written to.[5]

Granules: The Unit of Data Processing

The data within each column file is further logically divided into **granules**. A granule is the smallest indivisible unit of data that ClickHouse reads from disk during query execution.[5] By default, a granule consists of 8192 rows.[10] This concept is fundamental to ClickHouse's indexing strategy. When processing a query, ClickHouse does not read individual rows; it reads entire granules into memory for processing. This block-based approach is highly efficient for analytical workloads that scan large amounts of data.

Mark Files

To locate granules within the large column files, ClickHouse uses **mark files** (with a .mrk2 extension).[8] For each column, a mark file stores the physical offset in the

.bin file corresponding to the beginning of each granule. The primary index, which is small enough to fit in memory, contains pointers to these "marks." This allows the query engine to seek directly to the start of a required granule in a column file without having to scan the file from the beginning.

1.3 The Sparse Primary Index: The Key to Efficient Data Pruning

ClickHouse's primary index is one of its most critical and unique performance features. It is a "sparse" index that enables the engine to efficiently prune large swaths of data from a query, minimizing disk I/O.

The ORDER BY Key is the Primary Key

A crucial aspect of schema design in ClickHouse is that the primary index is defined by the ORDER BY clause, not a separate PRIMARY KEY clause.[8] The

ORDER BY expression determines the physical sort order of data within each data part. If a

PRIMARY KEY clause is specified, it must be a prefix of the ORDER BY key; its primary function is to provide a unique key for specialized engines like ReplacingMergeTree, not to define the main index.[9] This tight coupling of physical data order and the primary index makes the choice of the

ORDER BY key the single most important decision when designing a MergeTree table. The data sorting improves data compression, as similar values are grouped together, and it is the prerequisite for the sparse index to function effectively.[5]

Sparsity Explained

The index is termed "sparse" because, unlike a traditional B-Tree index in an OLTP database, it does not contain an entry for every row. Instead, it stores an index entry—the value of the ORDER BY key—for only the *first row* of each granule.[8] Given the default granule size of 8192 rows, this means the index is over 8000 times smaller than a dense index. This extreme sparseness ensures that the entire primary index for a data part (stored in the

primary.idx file) can easily fit into memory, even for tables with trillions of rows.[10]

Mechanism of Pruning

The data pruning process leverages this in-memory sparse index to dramatically reduce the amount of data read from disk. When a query contains a WHERE clause that filters on the columns of the ORDER BY key, the following occurs [10]:

1. ClickHouse loads the small primary.idx file for each relevant data part into memory.
2. It performs a fast binary search on this in-memory index to identify the ranges of granules whose first-row key values indicate that they *might* contain data matching the query's conditions. For example, if the key is a timestamp and the query asks for data from a specific hour, ClickHouse can quickly find the range of granules that cover that hour.
3. For the identified granule ranges, ClickHouse consults the mark files (.mrk2) to get the physical offsets of those granules within the on-disk column files (.bin).
4. Finally, the engine seeks directly to these offsets and reads only the required granules from disk, completely skipping all other granules.

This mechanism is exceptionally efficient for range queries and queries that filter on a prefix of the ORDER BY key.[12] To maximize its effectiveness, a common best practice is to order the columns in the

ORDER BY key from lowest cardinality to highest cardinality. This creates longer, more consistent runs of values for the initial columns in the key, allowing the index to prune data more effectively on those columns.[8]

1.4 Secondary Data Skipping Indexes

While the primary index is extremely powerful, it can only accelerate queries that filter on the ORDER BY key. To provide data skipping capabilities for other columns, ClickHouse offers secondary, or "data skipping," indexes.[13] These indexes store aggregate metadata for blocks of granules (where the block size is defined by the index's

GRANULARITY). During query planning, ClickHouse checks this metadata to determine if a block of granules can be skipped entirely.

Several types of data skipping indexes are available, each suited for different data types and query patterns [14]:

- **minmax:** This index stores the minimum and maximum values of the indexed expression for each block. It is very lightweight and ideal for accelerating range queries (>, <, BETWEEN) on numeric or date columns that are not in the primary key but may have some correlation with it.[14]
- **set(N):** This index stores a set of all unique values within a block, up to a maximum of N values. It is effective for equality or IN queries on columns that have low cardinality within each block, even if the overall column cardinality is high.[15]
- **bloom_filter:** This index uses a Bloom filter, a probabilistic data structure, to test for the presence of a value within a block. It is highly space-efficient and excellent for accelerating equality and IN queries on high-cardinality columns like UserID or IPAddress, where the goal is to find a "needle in a haystack".[13] A query can efficiently skip blocks where the Bloom filter guarantees the value is not present.
- **tokenbf_v1 and ngrambf_v1:** These are specialized Bloom filter variants designed for text search. tokenbf_v1 splits strings into tokens (words) and indexes them, accelerating hasToken() and LIKE queries for whole words. ngrambf_v1 splits strings into n-grams (substrings of length n), enabling efficient substring searches.[16]

While powerful, data skipping indexes add overhead to data ingestion and consume storage. They should be applied judiciously after the primary key has been carefully designed and optimized.[14]

1.5 Specialized MergeTree Variants: Handling Data Mutation and Aggregation

The base MergeTree engine is designed for immutable, append-only data. However, many real-world analytical use cases require semantics for updating, deleting, or pre-aggregating data. To accommodate this without sacrificing ingestion performance, ClickHouse provides a family of specialized MergeTree engines. These engines extend the base functionality by applying additional logic during the background merge process.[4]

This design pattern of deferring mutation logic to the background merge process is a cornerstone of ClickHouse's architecture. It avoids the performance penalty of immediate, synchronous read-modify-write operations that would be prohibitive in a columnar store. Instead, the *intent* of a mutation is captured in a new, quickly appended row (e.g., a row with Sign = -1 to signify a deletion). The actual mutation is then executed lazily and asynchronously during a merge. This creates a model of eventual consistency. Data in these tables can exist in an "unsettled" state—duplicates may be visible in a ReplacingMergeTree table, or canceled rows may appear in a CollapsingMergeTree table—until a merge has processed them. For queries that require absolute, up-to-the-second consistency, the FINAL modifier can be used in the FROM clause. This forces ClickHouse to perform a final merge of the data in the background before executing the query, guaranteeing a correct result at the cost of query performance.[18] The user must be aware of and actively manage this trade-off between ingestion speed, query performance, and data consistency.

The primary specialized variants are detailed in Table 1.

Table 1: The MergeTree Engine Family

Engine Name	Core Function	Key Mechanism	Ideal Use Case
MergeTree	Base engine for high-throughput analytics.	Background merges of immutable, sorted data parts.	General-purpose storage for time-series, logs, and event data. [5]
ReplicatedMergeT	Adds high	Replicates data and	Production

ree	availability to any MergeTree engine.	coordinates merges across nodes via ClickHouse Keeper.	deployments requiring fault tolerance and data redundancy. [4]
ReplacingMergeTree	Removes duplicate entries based on the sorting key.	During a merge, for rows with the same sorting key, it keeps only the last inserted row or the one with the maximum value in an optional ver column.	Deduplicating event streams or implementing upsert logic for dimension tables. [4]
SummingMergeTree	Automatically sums numeric data during merges.	During a merge, it replaces rows with the same sorting key with a single row where specified numeric columns are summed.	Creating simple, pre-aggregated summary tables for reporting. [4]
AggregatingMergeTree	Incrementally combines aggregate function states.	Stores intermediate aggregation states using the AggregateFunction data type and merges these states during background processing.	Building materialized views for complex aggregations (e.g., avg, uniq, quantiles) for dashboards. [4]
CollapsingMergeTree	Asynchronously collapses pairs of state/cancellation rows.	During a merge, it removes pairs of rows with the same sorting key but opposite Sign values (1 and -1).	Tracking the state of objects that change over time, where only the final state is important. [4]

VersionedCollapsingMergeTree	An enhanced version of CollapsingMergeTree that handles out-of-order state changes.	Uses an additional version column alongside the Sign column to correctly collapse rows, regardless of their insertion order.	More robust state tracking in distributed systems where event ordering is not guaranteed. [4]

Section 2: The Query Execution Pipeline - From SQL to Result Set

The transformation of a user's SQL query into a result set within ClickHouse is a multi-stage compilation and execution process designed for performance and transparency. This pipeline takes a high-level declarative SQL statement and converts it into a highly optimized, parallelized data flow graph that can be executed efficiently by the engine. Understanding this pipeline is crucial for diagnosing and optimizing query performance.

2.1 Query Lifecycle: Parsing and Semantic Analysis

The query lifecycle begins when a client application sends a SQL string to the server, which is received by the TCP handler.[25]

1. **Parsing (Lexical and Syntactic Analysis):** The raw SQL query is first passed to a parser. A lexical analyzer breaks the string into a sequence of fundamental units called tokens (e.g., keywords like SELECT, identifiers, operators). Following this, a syntactic analyzer constructs an **Abstract Syntax Tree (AST)** from the stream of tokens. The AST is a hierarchical, tree-based representation of the query's logical structure, capturing the relationships between its components.[25] This initial, unvalidated structure can be inspected using the
   EXPLAIN AST command.[27]
2. **Analyzer and Query Tree:** The AST is then handed off to the **Analyzer**. This is a critical component that performs semantic analysis. It validates the query by checking for the existence of databases, tables, and columns; verifying data types; and resolving identifiers like aliases and wildcards (*). The Analyzer transforms the purely syntactic AST into a more detailed and semantically rich **Query Tree**. This new structure has resolved

references to the underlying storage and has undergone initial logical optimizations.[26] ClickHouse is transitioning from an older analyzer to a new, more powerful architecture that is enabled by default.[26] The state of the query after this stage can be viewed with EXPLAIN QUERY TREE.[27]

2.2 Planning and Optimization

Once a valid Query Tree is constructed, it is passed to the **Planner**. The Planner's role is to convert the logical representation of the query (the *what*) into a concrete execution plan (the *how*).[26] This stage involves applying a series of advanced optimization rules, including:

- **Query Rewriting:** The query may be syntactically rewritten for more efficient execution. For example, JOINs might be reordered, or filters might be transformed.
- **Predicate Pushdown:** WHERE clause conditions are pushed down as close to the data source as possible to filter data early and reduce the amount of data processed in later stages.
- **Index Selection:** The planner analyzes the query's filters against the available primary and secondary indexes of the target tables. It determines which indexes can be used to prune data parts and granules, a critical step for performance.

The output of this stage is a **Query Plan**, which is a sequence of logical steps that the database will follow to produce the result. This plan can be inspected with EXPLAIN PLAN. Using the setting indexes=1 with this command is particularly valuable, as it reveals precisely which indexes were used and provides statistics on how many parts and granules were read versus the total available, offering direct feedback on the effectiveness of the schema design.[25]

2.3 The Execution Pipeline

The final stage of query compilation is the construction of the **Query Execution Pipeline** from the Query Plan. This pipeline is a directed graph of **Processors** (or IBlockInputStream objects in the older execution model) that stream data from one to another.[26] Each processor in the graph represents a specific operation in the query plan, such as:

- ReadFromMergeTree: Reads data blocks (granules) from the physical storage of a MergeTree table.
- Filter: Applies WHERE or HAVING conditions.

- ExpressionTransform: Applies calculations or transformations to columns (e.g., the expressions in the SELECT list).
- AggregatingTransform: Performs the GROUP BY aggregation.
- MergeSorting: Merges sorted blocks for an ORDER BY clause.

Execution within this pipeline is **vectorized**. Processors do not operate on individual rows; they consume and produce entire Blocks of data at a time.[2] This batch processing model is the key to leveraging CPU cache and SIMD instructions for high throughput. The pipeline is also inherently parallel; ClickHouse will instantiate multiple instances of pipeline segments to distribute the workload across available CPU cores, merging the results at the end.[25] The structure of this final data flow can be visualized using

EXPLAIN PIPELINE, which can produce either a textual description or a graphical representation in the DOT language.[26]

2.4 Introspection with EXPLAIN

The multi-stage compilation process in ClickHouse is not a black box. The system provides an extensive family of EXPLAIN statements that offer a window into each stage of the query's transformation. This level of introspection is a deliberate design choice, reflecting a philosophy of transparency that empowers expert users to understand and influence the query execution process at a granular level. While many databases provide a single EXPLAIN command, ClickHouse's differentiated tools allow a user to trace a query's evolution from raw SQL to final execution graph. This transforms performance tuning from a trial-and-error process into a systematic investigation. A user can see how their SQL is parsed (AST), how it is rewritten (SYNTAX), what logical plan is formed (QUERY TREE), how indexes are applied (PLAN), and how data will physically flow (PIPELINE). This empowers the user to become a partner in the optimization process, making informed decisions about schema design, indexing strategies, and query structure based on direct feedback from the engine's internal workings.

Table 2 provides a summary of the EXPLAIN statement types and their primary use cases in the optimization workflow.

Table 2: EXPLAIN Statement Types and Their Purpose

EXPLAIN Type	Output Represents	Primary Use Case	Key Settings
AST	Raw Abstract	Debugging	N/A

	Syntax Tree from the parser.	complex SQL syntax to understand how ClickHouse initially interprets the query.	
SYNTAX	The SQL query after AST-level optimizations and rewrites.	Understanding how ClickHouse normalizes or rewrites the query before semantic analysis.	run_query_tree_passes=1
QUERY TREE	Internal query tree structure after semantic analysis by the Analyzer.	Inspecting the logical query structure with resolved identifiers and initial optimizations.	run_passes=1
PLAN	The sequence of logical steps in the query execution plan.	**Verifying index usage and data pruning effectiveness.** This is the most common tool for performance tuning.	indexes=1, description=1
PIPELINE	The graph of data-processing nodes (processors) for query execution.	Identifying bottlenecks in the physical data flow and understanding parallelism.	graph=1, compact=1
ESTIMATE	Estimated number of rows, marks, and parts to be read.	A quick, pre-execution check to gauge the scope of a query's data scan on	N/A

		MergeTree tables.	

Section 3: Distributed Architecture - Scaling and High Availability

While ClickHouse delivers exceptional performance on a single server, its architecture is designed from the ground up to scale out to massive, multi-petabyte clusters. The distributed system capabilities are built on two orthogonal concepts: **replication** for high availability and data durability, and **sharding** for horizontal scaling of storage and compute resources. This design exhibits a "shared-nothing" architecture, where each node is independent and self-sufficient, communicating with others over the network without sharing disk or memory. A thin coordination layer, provided by ClickHouse Keeper, is used to manage consensus for replicated operations.

3.1 Replication with ReplicatedMergeTree

Replication in ClickHouse is the mechanism for achieving high availability and fault tolerance. It is implemented at the individual table level through the Replicated*MergeTree family of table engines (e.g., ReplicatedMergeTree, ReplicatedSummingMergeTree).[4]

- **Asynchronous Multi-Master Replication:** The replication model is asynchronous and multi-master. This means that INSERT and ALTER queries can be sent to any available replica in the cluster.[20] The server that receives the query first writes the data to its local disk and then adds a task to a replication queue. Other replicas watch this queue and pull the data to apply it locally. This asynchronous nature ensures that write operations are not blocked by slow or unavailable replicas, maintaining high ingestion throughput.
- **Data Consistency and Deduplication:** Replication operates on blocks of inserted data. Each block is assigned a unique identifier. The system automatically performs block-level deduplication, so if the same data block is sent to multiple replicas (for instance, due to a client-side retry), it will only be processed and stored once.[4] This makes INSERT statements idempotent, which is a crucial property for building reliable data pipelines. While data replication is asynchronous, the background merges of data parts are coordinated across all replicas to ensure that they are performed identically and in the same order, leading to bit-for-bit identical data parts across the cluster over time.
- **Scope of Replication:** It is important to note that replication applies only to the data

within tables using a Replicated*MergeTree engine. DDL (Data Definition Language) statements like CREATE TABLE or DROP TABLE are not replicated by the engine itself. Such statements must be executed on all nodes of a cluster, typically using the ON CLUSTER clause, which instructs the initiating node to forward the DDL query to all other nodes in the specified cluster.[20] For automating DDL replication, ClickHouse provides the Replicated database engine, which writes DDL logs to ZooKeeper/Keeper for execution on all database replicas.[29]

3.2 The Role of Clickhouse Keeper/ZooKeeper

To manage the state and coordination required for replication, ClickHouse relies on a distributed consensus system. While historically this role was filled by Apache ZooKeeper, the recommended and natively integrated solution is **ClickHouse Keeper**, a C++ implementation of the Raft consensus algorithm that is compatible with the ZooKeeper client protocol.[20]

The coordination system is the central nervous system for a replicated cluster and performs several critical functions:

- **Metadata Storage:** It stores the metadata for each replicated table, including the list of replicas, their health status, and paths to their data in the coordination system.[20]
- **Replication Log:** For each shard, it maintains a shared log of operations (e.g., insert a block, merge parts). Replicas use this log to determine which actions they need to perform to catch up with their peers.
- **Leader Election:** It facilitates leader election among replicas for tasks that require coordination, such as assigning background merges.
- **Distributed DDL Coordination:** When using the Replicated database engine, it stores the log of DDL queries to be executed across replicas.

For each INSERT into a replicated table, a small number of metadata entries are written to Keeper to log the operation.[20] However,

SELECT queries do not interact with the coordination system at all. This design ensures that read performance is completely unaffected by the overhead of replication.[20] A production-grade ClickHouse cluster's availability is directly tied to the availability of its Keeper ensemble, which should consist of an odd number of nodes (typically 3 or 5) spread across different fault domains.[31]

3.3 Sharding with the Distributed Engine

Sharding is ClickHouse's strategy for horizontal scaling, allowing a dataset to be partitioned across multiple servers (shards). This enables a cluster to handle data volumes and query loads far beyond the capacity of a single machine.[32]

Unlike replication, sharding is not implemented as a property of the storage engine. Instead, it is managed by a special, virtual table engine called the **Distributed** engine.[33] A table created with the

Distributed engine does not store any data itself. It acts as a transparent proxy or a distributed query router that forwards requests to the underlying local tables on the various shards.

The mechanism works as follows:

- **Cluster Definition:** A cluster, which is a collection of shards and their replicas, is defined in the server's configuration file.
- **Distributed Table Creation:** A Distributed table is created on one or more nodes, pointing to this cluster definition and the name of the underlying local tables on the shards.
- **INSERT Queries:** When data is inserted into the Distributed table, the engine uses a **sharding key** specified in the table definition (e.g., rand() for random distribution or intHash64(UserID) for consistent hashing) to determine which shard each row should be sent to. It then transparently forwards the rows to the appropriate shard.[33]
- **SELECT Queries:** When a SELECT query is executed against the Distributed table, the engine rewrites the query and sends it in parallel to all shards in the cluster (or a subset, if optimizations allow). Each shard executes the query on its local data. The initiating node then receives the partial results from all shards and merges them to produce the final result set for the client.[32]

This separation of the sharding mechanism (Distributed engine) from the replication mechanism (ReplicatedMergeTree engine) is a key architectural choice. It provides immense flexibility, allowing for complex cluster topologies. For example, a user can have replication without sharding (a single, highly-available shard) or sharding without replication (not recommended for production). However, this flexibility also places the responsibility on the user to correctly configure and understand the interaction between these two layers. A common point of confusion is that an INSERT made directly to a local ReplicatedMergeTree table will be replicated within its shard but will *not* be sharded across the cluster. To both shard and replicate data, the INSERT must be directed to the Distributed table, which then routes the data to the correct shard, where the underlying ReplicatedMergeTree engine takes over to handle replication within that shard.[33] Operating ClickHouse at scale requires a deep understanding of this interaction.

3.4 High Availability Best Practices

Building a robust, production-ready ClickHouse cluster involves combining replication, sharding, and operational best practices to mitigate risks of data loss and downtime. Based on official documentation and industry experience, the following practices are recommended [31]:

- **Component Redundancy:** Deploy at least three replicas for each shard to tolerate the failure of a single replica while still having a quorum for recovery operations. Similarly, deploy a ClickHouse Keeper (or ZooKeeper) ensemble of at least three nodes.
- **Fault Domain Isolation:** Disperse replicas and Keeper nodes across separate physical hardware and, ideally, different availability zones (AZs) or data centers. This prevents a single hardware, power, or network failure from taking down multiple components and compromising the cluster's availability.
- **Low-Latency Networking:** Ensure low and consistent network latency (ideally under 20ms round-trip) between all ClickHouse replicas within a shard and between the replicas and the Keeper ensemble. Higher latency can delay write acknowledgments and impact performance.
- **Backup and Disaster Recovery:** Replication protects against node failure but not against logical errors like accidental data deletion (DROP TABLE) or data corruption. A comprehensive disaster recovery strategy must include regular backups using tools like clickhouse-backup, with backups stored in a remote, durable location like an S3 bucket.
- **Regular Testing:** High availability procedures, including failover and recovery from backup, should be regularly tested in a non-production environment to ensure they work as expected and that operational teams are proficient in executing them.

Section 4: Performance Optimization Levers

Beyond its core architecture, ClickHouse provides a powerful toolkit of advanced features that allow users to fine-tune performance. These are not default settings but rather explicit levers that, when applied correctly, can dramatically improve query speed, reduce storage footprint, and minimize resource consumption. An expert ClickHouse user must think like a systems architect, constantly diagnosing performance bottlenecks and applying the appropriate tool to manage the trade-offs between storage, insert-time CPU, and query-time CPU. These features form an integrated performance-tuning toolkit that requires a holistic understanding of the system's resource trade-offs.

4.1 Data Compression and Encoding

Data compression is a first-order concern for performance in an I/O-bound analytical system. ClickHouse's columnar storage format is inherently well-suited for compression, as values of the same type are stored contiguously, often exhibiting patterns that compression algorithms can exploit effectively.[35] The physical sorting of data dictated by the

ORDER BY key further enhances this effect by grouping identical or similar values together.

ClickHouse offers granular control over compression at the column level:

- **Compression Codecs:** Users can specify a compression algorithm for each column using the CODEC clause. While the default is LZ4, which is optimized for very high compression and decompression speeds, ZSTD often provides a significantly better compression ratio with a modest increase in CPU usage.[35] For many workloads, the I/O savings from
  ZSTD's higher compression outweigh the additional CPU cost.
- **Specialized Encodings:** Before applying a generic compression algorithm, ClickHouse can transform the data using a specialized, data-type-aware encoding to make it more compressible.[37] These encodings are applied as a stack within the
  CODEC clause (e.g., CODEC(DoubleDelta, ZSTD)). Key encodings include:
  - Delta: This encoding stores the difference between consecutive values. It is extremely effective for columns with monotonic or slowly changing sequences, such as timestamps or counters.[37]
  - DoubleDelta: Stores the difference of the deltas, which is ideal for data with a constant rate of change.
  - Gorilla: An encoding optimized for floating-point time-series data where values often remain constant for periods before changing.
  - T64: A unique ClickHouse encoding that analyzes the range of integer values within a block and strips unnecessary high-order bits, effectively compacting the data.

If a query is I/O bound (i.e., spending most of its time reading data from disk), applying more effective codecs and encodings is the primary optimization strategy. This trades a small amount of CPU at insert time for a significant reduction in I/O at query time.

4.2 Materialized Views for Pre-computation

For queries that are CPU-bound due to expensive aggregations, Materialized Views (MVs) are the primary optimization tool. MVs shift the computational cost from read time to write time by pre-calculating and storing the results of a query.[39] ClickHouse supports two distinct types of materialized views.

- **Incremental Materialized Views:** This is the most common and powerful type of MV in ClickHouse. It functions as an insert trigger on a source table. When a new block of data is inserted into the source table, the MV's defining SELECT query is executed *only on that new block*. The partial result of this query is then inserted into a separate target table.[39] This target table is almost always an AggregatingMergeTree or SummingMergeTree engine, which is designed to merge these incoming partial results in the background over time.[23] This mechanism provides a form of continuous, real-time aggregation, making it ideal for powering dashboards and other applications that require fast access to up-to-date summary data.
- **Refreshable Materialized Views:** This newer type is more analogous to materialized views in traditional databases like PostgreSQL. A refreshable MV re-executes its entire defining query over the full source dataset on a user-defined schedule (e.g., REFRESH EVERY 1 HOUR).[40] Each time it runs, it replaces the contents of its target table with the new result set. This approach is suitable for complex queries, such as those involving resource-intensive JOINs, that are not amenable to incremental processing. It is also useful when some degree of data staleness is acceptable and the cost of the full re-computation can be borne during off-peak hours.

4.3 Dictionaries for High-Speed Lookups

For queries that are CPU-bound due to JOIN operations, particularly those involving a large fact table and a smaller dimension or enrichment table, ClickHouse Dictionaries offer a highly performant alternative.

- **In-Memory Key-Value Stores:** A Dictionary is an in-memory data structure that maps a key to one or more attributes.[42] Dictionaries are loaded into RAM from a variety of sources, including another ClickHouse table, an external database, or a flat file, and are periodically refreshed to stay in sync with the source.[43]
- **High-Speed Lookups with dictGet():** Instead of writing a JOIN in a query, a user can enrich data using special functions like dictGet('dictionary_name', 'attribute_name', key_expression). Because the dictionary resides in a highly optimized in-memory hash map, these lookups are extremely fast—often orders of magnitude faster than executing a traditional JOIN plan, which involves more complex algorithms and potential disk I/O.[42]
- **Layouts and Use Cases:** ClickHouse provides various dictionary layouts optimized for

different use cases, such as flat and hashed for simple key-value lookups, range_hashed for matching against date ranges, and ip_trie for efficient IP prefix matching.[43] Using a dictionary effectively trades server RAM (to store the dictionary) for a massive reduction in query-time CPU and latency.

Section 5: Architectural Context and Design Trade-offs

To fully appreciate the internal design of ClickHouse, it is essential to place it in the context of the broader analytical database landscape. Its architecture represents a unique set of design choices and trade-offs when compared to other prominent systems like Apache Druid, Apache Pinot, and Snowflake. This comparative analysis highlights ClickHouse's specific strengths and clarifies its ideal use cases. The fundamental differences between these systems are summarized in Table 3.

Table 3: Architectural Comparison of Analytical Databases

Dimension	ClickHouse	Apache Druid / Pinot	Snowflake
Architecture	Coupled Storage/Compute (evolving in Cloud)	Decoupled, multi-component microservices with deep storage	Fully decoupled multi-cluster, shared data architecture
Primary Use Case	Real-time, user-facing interactive analytics	Real-time analytics on streaming/event data	Enterprise BI, data warehousing, high-throughput batch analytics
Ingestion Model	Optimized for micro-batch ingestion	Built for true real-time, event-by-event streaming ingestion	Optimized for large batch loads; supports micro-batching (Snowpipe)
Query	Rich SQL dialect	Limited SQL; JOIN	ANSI SQL with full,

Language/JOINs	with strong JOIN support	support is restricted or less performant	powerful JOIN support
Scalability Model	Add identical nodes (shards/replicas); manual rebalancing	Independent scaling of ingestion, query, and storage nodes	Instant, elastic scaling of isolated compute ("virtual warehouses")
Management Overhead	High; requires significant expertise in tuning and operations	Very High; complex multi-component deployment and management	Low; fully managed SaaS that abstracts away infrastructure

5.1 ClickHouse vs. Real-Time OLAP Systems (Druid, Pinot)

Apache Druid and Apache Pinot are OLAP systems designed specifically for real-time analytics on high-volume, streaming event data. While they share the goal of low-latency queries with ClickHouse, their architectural philosophies differ significantly.

- **Architectural Complexity:** Druid and Pinot feature complex, distributed, microservice-based architectures.[45] They are composed of multiple distinct node types with specialized roles (e.g., Broker nodes for query routing, Historical nodes for serving historical data, and Real-time nodes for ingestion).[47] This design relies on an external "deep storage" layer, such as HDFS or S3, for data persistence. In contrast, ClickHouse has a much simpler, more monolithic architecture, typically deployed as a single binary on a cluster of identical nodes, which are responsible for both storage and compute.[45] This makes ClickHouse easier to deploy and manage initially, but the modularity of Druid/Pinot allows for more granular, independent scaling of different system functions.
- **Ingestion Model:** Druid and Pinot are architected for true real-time, event-by-event ingestion, making data available for querying almost instantaneously.[46] ClickHouse, while capable of handling streaming data via its Kafka engine, performs best when data is ingested in micro-batches of at least a thousand rows at a time to optimize the creation of
MergeTree parts.[45] This makes Druid and Pinot a more natural fit for use cases where sub-second data freshness is an absolute requirement.
- **Query Capabilities:** ClickHouse offers a more mature and comprehensive SQL dialect, including robust support for complex, multi-table JOINs.[51] Druid and Pinot have

historically had more limited
JOIN capabilities, focusing primarily on high-performance filtering and aggregation on a single, denormalized fact table.[49] This makes ClickHouse more versatile for exploratory analytics and workloads that require combining different datasets.

5.2 ClickHouse vs. Cloud Data Warehouses (Snowflake)

Snowflake represents the modern, fully managed cloud data warehouse, and its design philosophy contrasts sharply with that of ClickHouse.

- **Core Architectural Difference: Coupled vs. Decoupled:** The most fundamental distinction is the separation of storage and compute. Snowflake pioneered the multi-cluster, shared data architecture, where all data resides in a central object storage layer (like S3), and multiple, independent "virtual warehouses" (compute clusters) can access this data concurrently.[55] This allows for seamless, independent scaling of storage and compute resources. Traditional open-source ClickHouse, in contrast, follows a coupled, shared-nothing architecture where storage (local disk) and compute reside on the same nodes. It is important to note, however, that the managed ClickHouse Cloud offering is evolving this model by also leveraging object storage, which begins to blur this architectural line.[55]
- **Performance and Use Case Focus:** This architectural difference dictates their ideal use cases. ClickHouse is a performance-engineered engine optimized for extreme low-latency, sub-second queries. Its design is tailored for powering interactive, user-facing analytical applications where responsiveness is paramount.[55] Snowflake is optimized for high-throughput, large-scale Business Intelligence (BI) and data warehousing queries. It excels at handling complex, long-running queries and high user concurrency, where query latencies of several seconds or even minutes are often acceptable.[50]
- **Management and Tuning:** ClickHouse is a "system for engineers," exposing a vast array of configuration settings, engine choices, and indexing strategies that require significant expertise to tune for optimal performance.[56] Snowflake is a fully managed SaaS platform that abstracts away nearly all of this complexity. It provides simple "T-shirt sized" compute warehouses and handles most optimization automatically, making it far easier to operate but offering less granular control.[50]

These comparisons reveal the unique niche that ClickHouse occupies. It is neither a pure stream-processing engine like Druid nor a managed, high-throughput BI warehouse like Snowflake. It is best described as a "user-facing data warehouse" engine. Its architecture is tailored for applications that need to execute complex analytical queries, including JOINs, over massive, rapidly ingested datasets and return results in milliseconds to power an interactive user experience. This demanding and growing market segment sits squarely between traditional BI and pure stream processing, and it is here that ClickHouse's specific set of architectural trade-offs provides a compelling solution.

Conclusion: Synthesizing the Internals for Optimal Use

The internal architecture of ClickHouse is a masterclass in purpose-built design for high-performance analytical query processing. Its efficacy stems from a cohesive set of foundational principles that are consistently applied across every layer of the system. The strict adherence to a columnar data model, from physical disk storage to in-memory representation, enables both remarkable data compression and the system's primary performance driver: vectorized query execution. This model, which processes data in batches rather than row-by-row, maximizes the use of modern CPU capabilities and is the key to its sub-second query speeds.

The MergeTree engine family stands as the core of the storage layer, its LSM Tree-based design providing exceptionally high ingestion throughput by deferring the cost of data organization to an asynchronous background process. This core design is ingeniously extended by a family of specialized engines that introduce mutation-like semantics—such as deduplication, pre-aggregation, and state-change tracking—into an append-only, immutable world, albeit with a trade-off of eventual consistency that users must actively manage.

A deep understanding of these internals is not merely an academic exercise; it is the prerequisite for unlocking the full potential of ClickHouse. The analysis reveals several critical takeaways for architects and engineers:

- **Schema Design is Paramount:** The tight coupling of the physical data sort order with the primary index means that the choice of the ORDER BY key is the single most impactful decision in schema design. A well-chosen key, ordered from low to high cardinality, is the foundation of all query performance.
- **Optimization is a Multi-faceted Discipline:** Performance tuning in ClickHouse is a systematic process of diagnosing bottlenecks and applying the appropriate lever. The comprehensive EXPLAIN toolkit provides the necessary transparency to determine if a query is bound by I/O (addressable with compression codecs), CPU-intensive aggregations (addressable with Materialized Views), or complex JOINs (addressable with Dictionaries).
- **Distributed Operations Require Nuanced Understanding:** Scaling ClickHouse effectively requires a clear comprehension of the distinct and orthogonal roles of replication (via ReplicatedMergeTree for high availability) and sharding (via the Distributed engine for horizontal scale). The interaction between these two layers must

be correctly configured to build a robust and performant cluster.

In essence, ClickHouse's architectural philosophy is one of performance through transparency and tune-ability. It provides powerful automated optimizations but also gives expert users the visibility and control to fine-tune every aspect of its operation. By mastering these internal mechanisms, users can build analytical systems that meet the most demanding requirements for speed, scale, and real-time responsiveness.

Works cited

1. Architecture Overview | ClickHouse Docs, accessed August 16, 2025, https://clickhouse.com/docs/academic_overview
2. Architecture Overview | ClickHouse Docs, accessed August 16, 2025, https://clickhouse.com/docs/development/architecture
3. Understanding ClickHouse®: Products, architecture, tutorial and ..., accessed August 16, 2025, https://www.instaclustr.com/education/clickhouse/understanding-clickhouse-products-architecture-tutorial-and-alternatives/
4. MergeTree Engine Family | ClickHouse Docs, accessed August 16, 2025, https://clickhouse.com/docs/engines/table-engines/mergetree-family
5. MergeTree | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/engines/table-engines/mergetree-family/mergetree
6. Table parts | ClickHouse Docs, accessed August 16, 2025, https://clickhouse.com/docs/parts
7. ClickHouse Table Engine Overview - Cloud Service Help Center, accessed August 16, 2025, https://doc.hcs.huawei.com/usermanual/mrs/mrs_01_24105.html
8. ClickHouse Basic Tutorial: Keys & Indexes - DEV Community, accessed August 16, 2025, https://dev.to/hoptical/clickhouse-basic-tutorial-keys-indexes-5d7a
9. The Power of Sparse Indexes in ClickHouse | by Sanjeev Singh ..., accessed August 16, 2025, https://medium.com/@sjksingh/the-power-of-sparse-indexes-in-clickhouse-d4ab6b05c420
10. Primary indexes | ClickHouse Docs, accessed August 16, 2025, https://clickhouse.com/docs/primary-indexes
11. Primary indexes | ClickHouse Docs, accessed August 16, 2025, https://clickhouse.com/docs/zh/primary-indexes
12. How to choose a primary key in ClickHouse® - Propel Data, accessed August 16, 2025, https://www.propeldata.com/blog/how-to-choose-a-primary-key-in-clickhouse
13. How Data Skipping Indexes are implemented in ClickHouse?, accessed August 17, 2025, https://chistadata.com/how-data-skipping-indexes-are-implemented-in-clickhouse/
14. Use data skipping indices where appropriate | ClickHouse Docs, accessed August 17, 2025,

https://clickhouse.com/docs/best-practices/use-data-skipping-indices-where-appropriate
15. Understanding ClickHouse Data Skipping Indexes | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/optimize/skipping-indexes
16. Mastering ClickHouse Data Skipping Indexes: A Guide to Optimizing Query Performance | by sathish kumar srinivasan | Medium, accessed August 17, 2025, https://medium.com/@sathishdba/mastering-clickhouse-data-skipping-indexes-a-guide-to-optimizing-query-performance-013101168352
17. Using Bloom filter indexes for real-time text search in ClickHouse®, accessed August 17, 2025, https://www.tinybird.co/blog-posts/using-bloom-filter-text-indexes-in-clickhouse
18. ReplacingMergeTree | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/engines/table-engines/mergetree-family/replacingmergetree
19. CollapsingMergeTree | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/engines/table-engines/mergetree-family/collapsingmergetree
20. Data Replication | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/engines/table-engines/mergetree-family/replication
21. ReplacingMergeTree | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/guides/replacing-merge-tree
22. SummingMergeTree | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/engines/table-engines/mergetree-family/summingmergetree
23. clickhouse.com, accessed August 17, 2025, https://clickhouse.com/docs/engines/table-engines/mergetree-family/aggregatingmergetree
24. VersionedCollapsingMergeTree | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/engines/table-engines/mergetree-family/versionedcollapsingmergetree
25. Comprehensive Guide to ClickHouse EXPLAIN - ChistaDATA, accessed August 16, 2025, https://chistadata.com/comprehensive-guide-clickhouse-explain/
26. Understanding Query Execution with the Analyzer | ClickHouse Docs, accessed August 16, 2025, https://clickhouse.com/docs/guides/developer/understanding-query-execution-with-the-analyzer
27. EXPLAIN Statement | ClickHouse Docs, accessed August 16, 2025, https://clickhouse.com/docs/sql-reference/statements/explain
28. ClickHouse Query Execution Pipeline - clickhouse-presentations, accessed August 16, 2025, https://presentations.clickhouse.com/meetup24/5.%20Clickhouse%20query%20execution%20pipeline%20changes/
29. Replicated | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/engines/database-engines/replicated
30. ClickHouse Keeper | ClickHouse Docs, accessed August 17, 2025,

https://clickhouse.com/docs/guides/sre/keeper/clickhouse-keeper

31. ClickHouse® High Availability Architecture - Altinity® Documentation, accessed August 16, 2025, https://docs.altinity.com/operationsguide/availability-and-recovery/availability-architecture/
32. jaeger-clickhouse/guide-sharding-and-replication.md at main - GitHub, accessed August 17, 2025, https://github.com/jaegertracing/jaeger-clickhouse/blob/main/guide-sharding-and-replication.md
33. Clickhouse — sharding and replication | by Sairam Krish | Medium, accessed August 17, 2025, https://sairamkrish.medium.com/clickhouse-sharding-and-replication-95b3275c873e
34. Special Table Engines | ClickHouse Docs, accessed August 16, 2025, https://clickhouse.com/docs/engines/table-engines/special
35. Compression in ClickHouse | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/data-compression/compression-in-clickhouse
36. Compression Modes | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/data-compression/compression-modes
37. New Encodings to Improve ClickHouse® Efficiency - Altinity | Run ..., accessed August 17, 2025, https://altinity.com/blog/2019-7-new-encodings-to-improve-clickhouse
38. Demystifying Data Compression in ClickHouse | ChistaDATA Blog, accessed August 17, 2025, https://chistadata.com/data-compression-in-clickhouse/
39. Incremental materialized view | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/materialized-view/incremental-materialized-view
40. Use Materialized Views | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/best-practices/use-materialized-views
41. Refreshable materialized view | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/materialized-view/refreshable-materialized-view
42. Join me if you can: ClickHouse vs. Databricks vs. Snowflake - Part 2, accessed August 17, 2025, https://clickhouse.com/blog/join-me-if-you-can-clickhouse-vs-databricks-snowflake-part-2
43. Dictionaries | ClickHouse Docs, accessed August 17, 2025, https://clickhouse.com/docs/sql-reference/dictionaries/
44. Polygon Dictionaries in ClickHouse - YouTube, accessed August 17, 2025, https://www.youtube.com/watch?v=FyRsriQp46E
45. In-depth: ClickHouse vs Druid - PostHog, accessed August 17, 2025, https://posthog.com/blog/clickhouse-vs-druid
46. ClickHouse vs Druid: 10 Feature-By-Feature Comparison (2025) - Chaos Genius, accessed August 17, 2025, https://www.chaosgenius.io/blog/clickhouse-vs-druid/
47. Comparison of the Open Source OLAP Systems for Big Data ..., accessed August 17, 2025, https://leventov.medium.com/comparison-of-the-open-source-olap-systems-for

-big-data-clickhouse-druid-and-pinot-8e042a5ed1c7
48. Apache Pinot, Druid, and Clickhouse Comparison | StarTree, accessed August 17, 2025, https://startree.ai/resources/a-tale-of-three-real-time-olap-databases
49. ClickHouse vs. Apache Druid: A Detailed Comparison - CelerData, accessed August 17, 2025, https://celerdata.com/glossary/clickhouse-vs-apache-druid
50. ClickHouse vs Snowflake—7 Reasons for Choosing One (2025) - Chaos Genius, accessed August 17, 2025, https://www.chaosgenius.io/blog/clickhouse-vs-snowflake/
51. Apache Druid vs ClickHouse: A Comprehensive Comparison for B2B Analytics Solutions, accessed August 17, 2025, https://www.ksolves.com/blog/big-data/apache-druid-vs-clickhouse
52. Join me if you can: ClickHouse vs. Databricks vs. Snowflake - Part 1, accessed August 17, 2025, https://clickhouse.com/blog/join-me-if-you-can-clickhouse-vs-databricks-snowflake-join-performance
53. Druid vs ClickHouse (2024) - Firebolt, accessed August 17, 2025, https://www.firebolt.io/comparison/druid-vs-clickhouse
54. Apache Pinot vs. ClickHouse Comparison - SourceForge, accessed August 17, 2025, https://sourceforge.net/software/compare/Apache-Pinot-vs-ClickHouse/
55. ClickHouse vs Snowflake: Key Differences, Performance & Pricing ..., accessed August 17, 2025, https://estuary.dev/blog/clickhouse-vs-snowflake/
56. Clickhouse vs Snowflake | Performance & Pricing: Comparison Guide, accessed August 17, 2025, https://www.firebolt.io/comparison/snowflake-vs-clickhouse
57. ClickHouse vs Snowflake, accessed August 17, 2025, https://clickhouse.com/comparison/snowflake