Mastering SQL Engineering and Indexing in ClickHouse

Welcome to this comprehensive guide on optimizing SQL engineering and indexing in ClickHouse. As data volumes grow exponentially, the need for high-performance analytical databases becomes critical. ClickHouse stands out as a purpose-built solution for analytical workloads, offering remarkable query speeds at scale.

Throughout this presentation, we'll explore the fundamental principles and advanced techniques to maximize your ClickHouse implementation's performance. From understanding its unique architecture to implementing optimal primary key designs and SQL query patterns, you'll gain the knowledge needed to achieve peak performance for your analytical workloads.

 **by Shiv Iyer**

Agenda

1

ClickHouse Architecture Fundamentals

Column-oriented storage and sparse indexing model

Primary Key Design Principles

Optimizing your sort order and indexing strategy

SQL Query Optimization

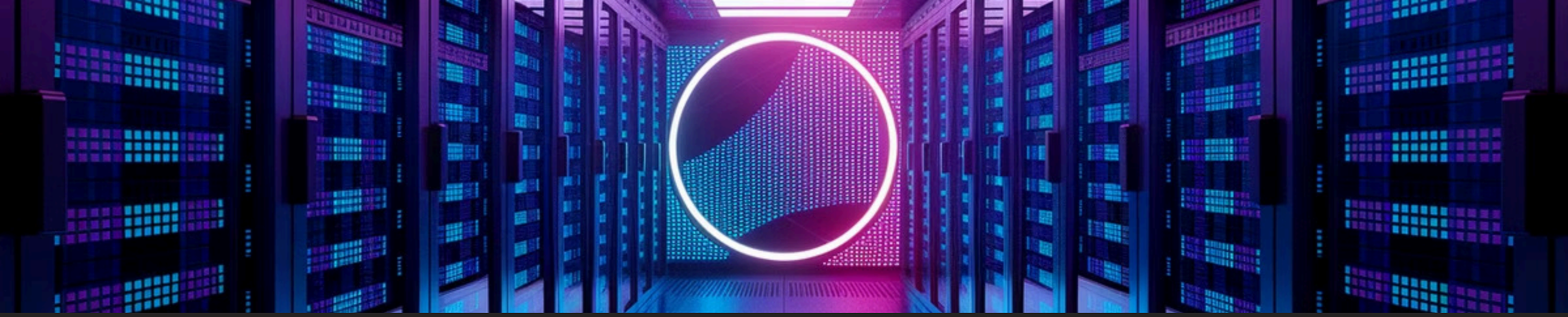
Best practices for writing efficient queries

4

Performance Monitoring and Tuning

Tools and techniques for ongoing optimization

Our journey will begin with a deep dive into ClickHouse's architecture to establish a solid foundation of knowledge. We'll then explore how to design optimal primary keys, write efficient SQL queries, and implement ongoing monitoring and tuning practices to ensure sustained performance.



What Makes ClickHouse Different?

True Column-Oriented Storage

Designed from the ground up for analytical workloads with columnar data organization that enables exceptional query performance

Sparse Primary Indexing

Unique approach to indexing that reduces memory footprint while maintaining fast data access

Advanced Data Compression

Highly efficient compression algorithms specific to each data type, reducing storage requirements by up to 10x

Vectorized Query Execution

Processes data in chunks rather than row-by-row, resulting in dramatically faster analytics

These fundamental differences give ClickHouse significant performance advantages over traditional row-oriented databases when handling analytical workloads. Understanding these unique characteristics is essential for effective optimization.

Column-Oriented Architecture

How it Works

In ClickHouse, data is organized by columns rather than rows. Each column is stored in separate files, allowing the system to read only the specific columns needed for a query.

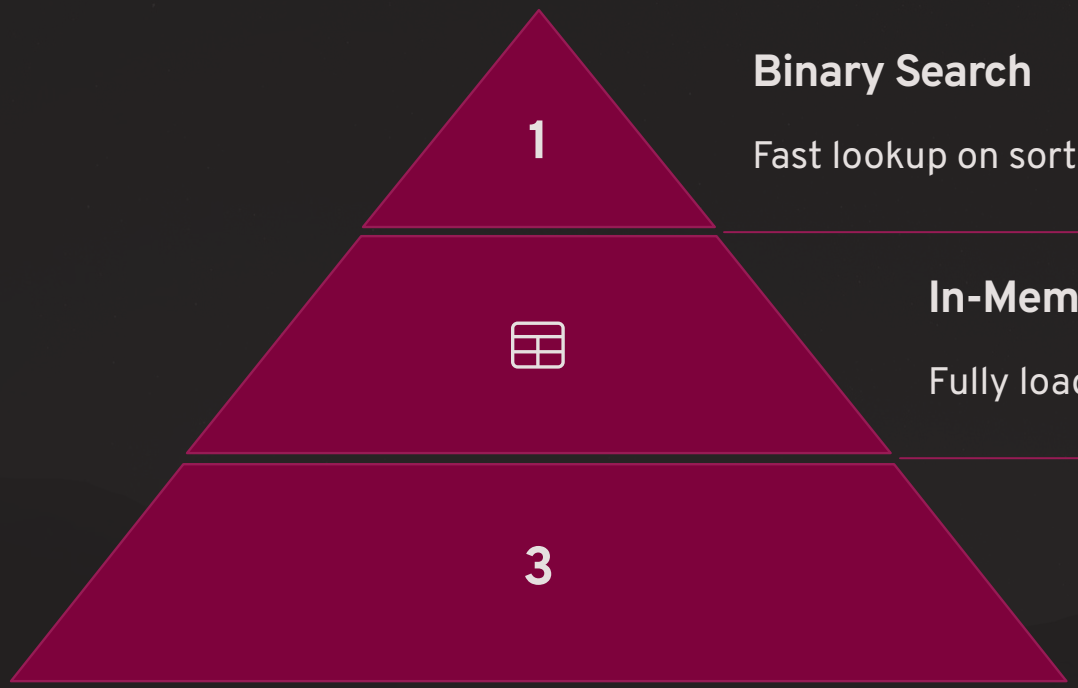
This approach stands in contrast to row-oriented databases where all column values for a single row are stored together, requiring the system to read unnecessary data when only specific columns are needed.

Key Benefits

- Dramatically reduced I/O for analytical queries
- Better compression ratios for similar data types
- Efficient vectorized processing on column chunks
- Improved cache locality for better CPU utilization

This column-oriented organization is the foundation of ClickHouse's performance advantages for analytical workloads. It enables the system to minimize the amount of data read from disk while maximizing processing efficiency.

Sparse Primary Indexing: A Different Approach



Binary Search

Fast lookup on sorted data

In-Memory Index

Fully loaded for rapid access

Granule-Level Indexing

One index entry per data granule

ClickHouse uses a fundamentally different indexing approach compared to traditional databases. Instead of maintaining a B-tree index that points to individual rows, ClickHouse creates a sparse index with one entry per granule (typically 8,192 rows). This dramatically reduces the memory footprint of the index while still providing fast data access.

The index is fully loaded into RAM at query time, enabling rapid binary search operations. Data is physically sorted on disk according to the primary key, allowing the system to quickly identify which granules need to be read for a given query condition.

How Sparse Indexing Works

Data Organized by Primary Key

All data is physically sorted on disk according to the columns in the primary key, creating ordered data blocks

Granule Creation

Data is divided into granules (default 8,192 rows), with one index entry created for the first row in each granule

Index Loading

When a query is executed, the entire sparse index is loaded into memory for fast lookups

Granule Selection

The system uses binary search on the index to determine which granules may contain relevant data

Mark Files

Special mark files help locate the exact position of granules on disk for efficient reading

This sparse indexing approach provides an excellent balance between memory efficiency and query performance. It allows ClickHouse to handle massive datasets while maintaining fast query response times.

Primary Key vs. Order By: Important Distinction

PRIMARY KEY Definition

In ClickHouse, the PRIMARY KEY determines what columns are used to build the sparse index. It defines the lookup mechanism but doesn't guarantee uniqueness.

- Creates sparse index entries
- Used for data skipping
- Does not enforce uniqueness

ORDER BY Definition

The ORDER BY clause determines how data is physically sorted on disk. This physical ordering is critical for performance.

- Determines physical data layout
- Affects compression efficiency
- Cannot be changed after table creation

Default Behavior

If PRIMARY KEY is not specified, it defaults to the same value as ORDER BY. In most cases, these should be the same, but there are scenarios where they might differ.

- Separate for specialized use cases
- Same for most implementations

Understanding this distinction is crucial for optimal table design in ClickHouse. While they often share the same columns, recognizing their different roles helps in designing more effective schemas.

Primary Key Design: Column Ordering Principles

Match Query Patterns

Design primary keys to support your most common query patterns. Columns used frequently in WHERE clauses should be considered for inclusion in the primary key.

Order by Cardinality (Ascending)

Place lower-cardinality columns (fewer unique values) first, followed by higher-cardinality columns. This improves both data compression and query performance.

Prioritize First Column

The first column enables efficient binary search operations, while secondary columns use less efficient generic exclusion methods. Choose this column carefully.

Balance Selectivity

Aim for a primary key that provides good selectivity - not too broad or too narrow - to optimize the number of granules that need to be scanned.

Following these principles will help you create primary keys that significantly enhance query performance by reducing the amount of data that needs to be scanned for each query.

Primary Key Column Ordering Example

Sub-Optimal Design

```
CREATE TABLE events (
  URL String,
  UserID UInt64,
  IsRobot UInt8,
  EventTime DateTime
) ENGINE = MergeTree()
ORDER BY (URL, UserID, IsRobot);
```

This design places a high-cardinality column (URL) first, resulting in less efficient compression and potentially more granules to scan.

Optimized Design

```
CREATE TABLE events (
  URL String,
  UserID UInt64,
  IsRobot UInt8,
  EventTime DateTime
) ENGINE = MergeTree()
ORDER BY (IsRobot, UserID, URL);
```

This improved design places columns in order of increasing cardinality: IsRobot (lowest) first, followed by UserID, then URL (highest). This improves both compression and query performance.

The optimized design allows ClickHouse to skip more granules when filtering on IsRobot, resulting in better performance for queries that filter on this column. Additionally, similar values are grouped together, improving data compression.

Handling Multiple Query Patterns

Secondary Tables

Create additional tables with different primary keys to support diverse query patterns

- Manual synchronization required
- Highest flexibility for different schemas

Materialized Views

Automatically maintain synchronized copies with different sort orders

- Automatic synchronization
- Separate storage with different indexes

Projections

Let ClickHouse automatically choose the best index for each query

- Most transparent option
- Automatic selection by query optimizer

When different query patterns require different primary key organizations, these approaches provide ways to optimize for multiple access patterns. Each has trade-offs in terms of maintenance overhead, storage requirements, and query transparency.

Using ClickHouse Projections

What Are Projections?

Projections allow you to maintain alternative data organizations optimized for different query patterns. They're like materialized views but with automatic selection by the query optimizer.

Each projection can have its own:

- Column subset
- Primary key order
- Aggregation or transformation

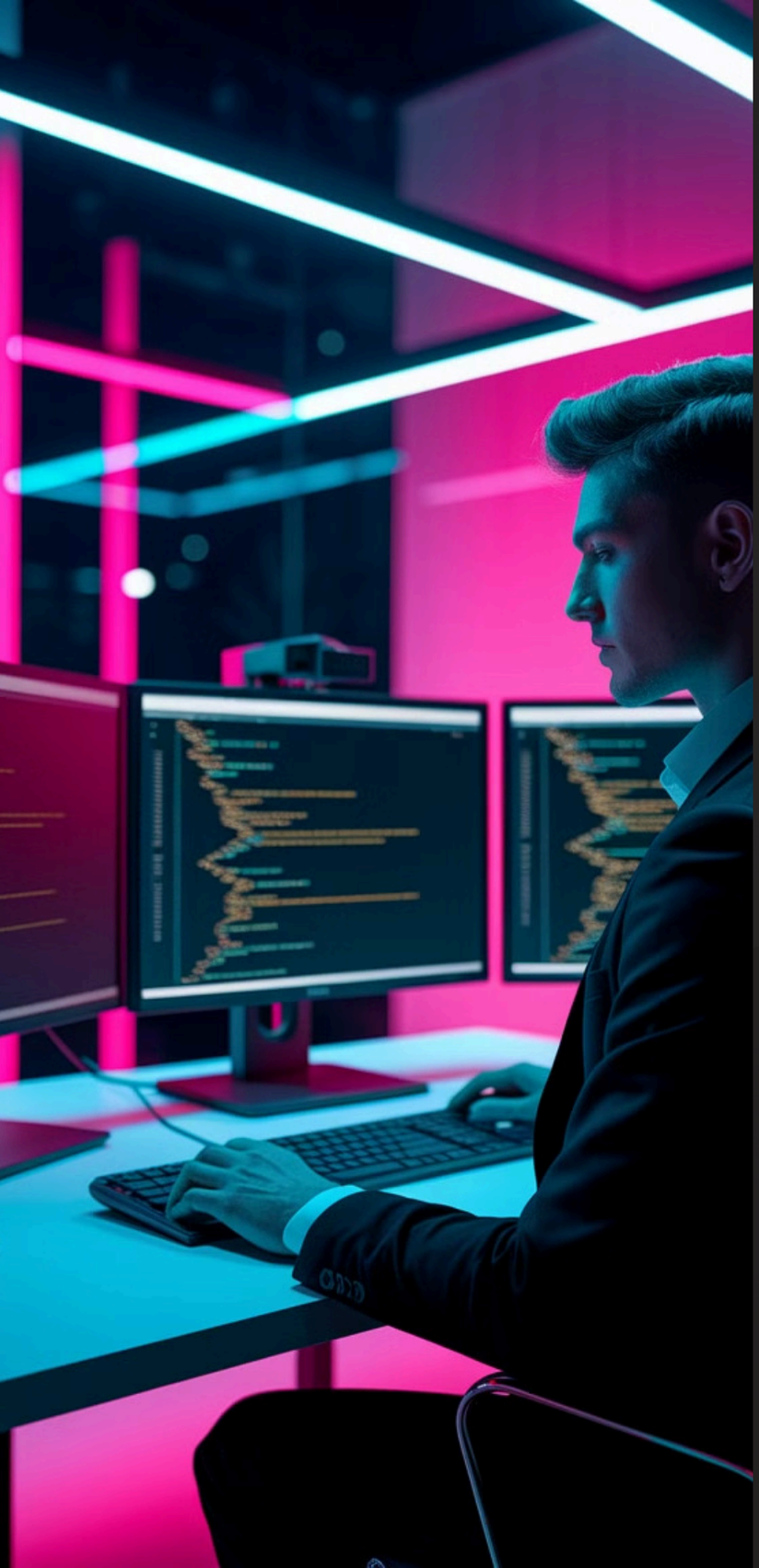
Implementation Example

```
CREATE TABLE web_events (
  EventDate Date,
  UserID UInt64,
  URL String,
  RegionID UInt32
) ENGINE = MergeTree()
ORDER BY (EventDate, UserID)
SETTINGS index_granularity = 8192;

ALTER TABLE web_events
ADD PROJECTION region_projection (
  SELECT * ORDER BY (RegionID, EventDate)
);

-- Materialize the projection
ALTER TABLE web_events
MATERIALIZE PROJECTION region_projection;
```

Projections provide a powerful way to optimize for multiple query patterns without manual table maintenance. The query optimizer automatically selects the most appropriate projection based on the query conditions.



SQL Query Optimization: General Principles

Filter Early and Effectively

Apply WHERE conditions that leverage the primary key to reduce data scanned as early as possible in your query

Select Only Required Columns

Avoid SELECT * by explicitly listing only the columns needed by your application

Optimize JOINs

Keep JOIN operations efficient by using matching types and filtering data before joining

Use Bulk Operations

Insert data in large batches rather than small, frequent operations for better performance

These fundamental principles form the foundation of efficient query design in ClickHouse. By focusing on these aspects, you can significantly improve the performance of your analytical workloads and reduce resource consumption.

Leveraging the Primary Key in Queries

Optimal Query Pattern

Queries that filter on the first column of your primary key will perform best, as they can leverage binary search.

```
-- Efficient query using primary key
SELECT COUNT(*)
FROM events
WHERE IsRobot = 0
  AND UserID = 12345;
```

Sub-Optimal Query Pattern

Queries that don't filter on indexed columns will result in full table scans, which can be extremely slow on large tables.

```
-- Inefficient query missing primary key
SELECT COUNT(*)
FROM events
WHERE URL LIKE '%product%';
```

The performance difference between these query patterns can be dramatic - often orders of magnitude. When designing your queries, always consider how they interact with your primary key to ensure optimal performance. When possible, include conditions on at least the first column of your primary key to enable efficient data skipping.

Query Filtering Best Practices

Use Equality Conditions on First Primary Key Column

Filtering with equality conditions (=, IN) on the first primary key column enables the most efficient binary search on the sparse index.

```
WHERE department = 'Engineering'
```

Range Conditions Work Best on Last Primary Key Column

Range conditions (>, <, BETWEEN) are most efficient when used on the last column of your primary key after equality conditions on preceding columns.

```
WHERE department = 'Engineering'
  AND hire_date BETWEEN '2020-01-01' AND '2020-12-31'
```

Avoid Functions on Indexed Columns

Applying functions to indexed columns prevents the optimizer from using the index effectively. Move functions to the right side of conditions when possible.

```
-- Inefficient: WHERE LOWER(status) = 'active'
-- Better: WHERE status = 'ACTIVE'
```

Understanding how different types of filter conditions interact with ClickHouse's sparse indexing model is crucial for writing efficient queries. Properly structured filter conditions can dramatically reduce the amount of data that needs to be scanned.

Optimizing Complex Filtering

Sub-Optimal Pattern

```
SELECT
  EventDate,
  COUNT(*)
FROM web_events
WHERE
  URL LIKE '%checkout%'
  AND EventDate BETWEEN
    '2023-01-01' AND '2023-01-31'
GROUP BY EventDate;
```

This query has a condition on EventDate (indexed) but also includes a non-indexed LIKE filter on URL which may prevent effective use of the index.

Optimized Pattern

```
SELECT
  EventDate,
  COUNT(*)
FROM web_events
WHERE
  EventDate BETWEEN
    '2023-01-01' AND '2023-01-31'
  AND URL LIKE '%checkout%'
GROUP BY EventDate;
```

Placing the indexed column condition first clarifies intent, though ClickHouse's optimizer should reorder conditions regardless. The key improvement would be creating a secondary index on URL if this pattern is common.

For consistently better performance with text searches, consider implementing a specialized secondary index such as a bloom filter or n-gram index on the URL column. This would significantly improve filtering performance for the LIKE condition.

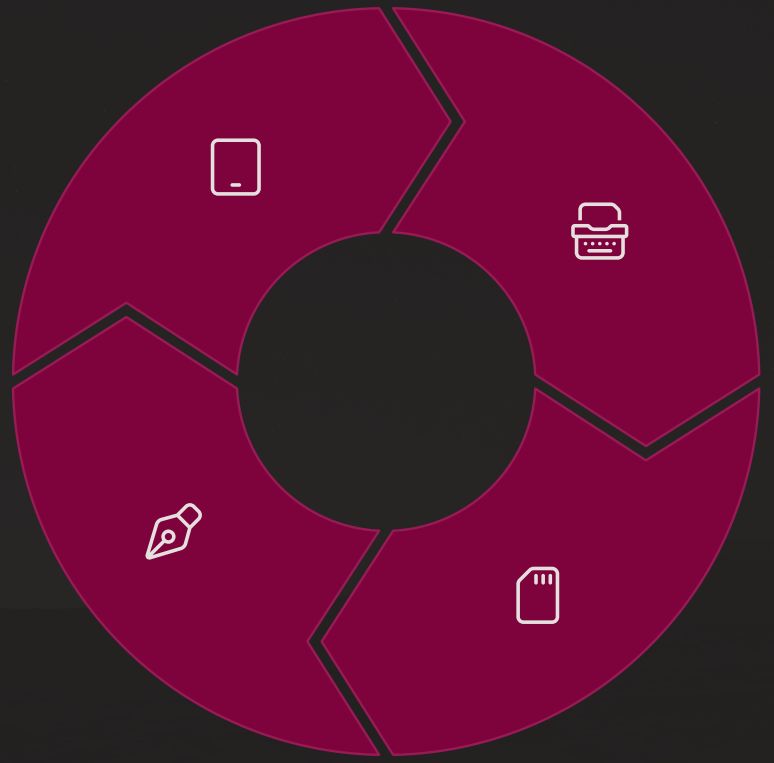
JOIN Optimization Strategies

Filter Before Joining

Apply WHERE clauses to each table before the JOIN to reduce the data volume in the join operation

Match Data Types

Ensure join columns use identical data types to avoid type conversion overhead



Evaluate Denormalization

Sometimes replacing JOINs with denormalized tables improves performance dramatically

Consider Join Engine

Use the appropriate join method based on table sizes and memory constraints

JOIN operations can be resource-intensive in any database system, and ClickHouse is no exception. By following these optimization strategies, you can significantly improve the performance of queries that require joining multiple tables.

Remember that ClickHouse is primarily designed for analytical workloads and often performs best with denormalized data models that minimize the need for complex joins.

JOIN Optimization Example

Unoptimized JOIN

```
SELECT
  u.username,
  COUNT(e.event_id) as event_count
FROM events e
JOIN users u ON e.user_id = u.id
WHERE
  e.event_date >= '2023-01-01'
GROUP BY u.username;
```

This query joins the entire events and users tables before filtering, potentially processing unnecessary data.

Optimized JOIN

```
SELECT
  u.username,
  COUNT(e.event_id) as event_count
FROM
  (
    SELECT user_id, event_id
    FROM events
    WHERE event_date >= '2023-01-01'
  ) e
JOIN users u ON e.user_id = u.id
GROUP BY u.username;
```

This query filters the events table first, reducing the amount of data involved in the JOIN operation.

The optimized query can perform significantly better, especially when the filter on event_date substantially reduces the number of rows from the events table. Always try to minimize the amount of data involved in JOIN operations by applying filters before joining.

Denormalization for Performance

When to Consider Denormalization

ClickHouse often performs best with denormalized data models, particularly for:

- High-frequency analytical queries
- Data with relatively stable dimensions
- Scenarios where JOIN performance is a bottleneck

Implementation Approaches

Several methods to implement denormalization:

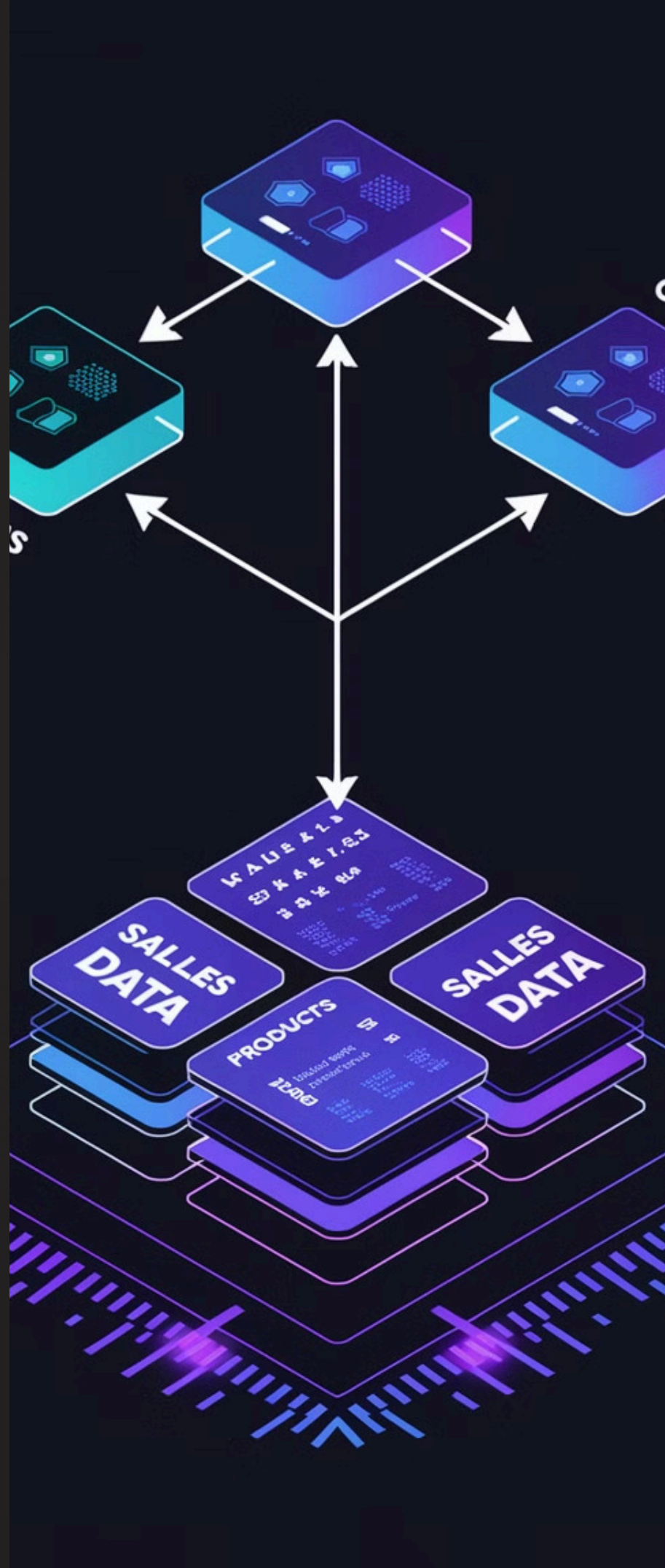
- Pre-join data during ETL processes
- Use materialized views to maintain denormalized copies
- Leverage dictionary tables for efficient lookups

Trade-offs to Consider

Denormalization has both benefits and costs:

- Increased storage requirements
- More complex data update processes
- Potential data consistency challenges

Unlike traditional OLTP databases where normalization is prioritized, ClickHouse's analytical focus often makes denormalization a better choice. The performance gains from avoiding complex JOINs can outweigh the added storage requirements and management complexity.



Dictionaries: A Powerful Alternative to JOINs

What Are ClickHouse Dictionaries?

Dictionaries are specialized data structures for storing relatively small, static reference data in memory for ultra-fast lookups. They provide an efficient alternative to JOINs for dimension tables.

Key characteristics:

- Stored entirely in RAM for fast access
- Support automatic updates from various sources
- Optimized for key-value or key-object lookups

Implementation Example

```
CREATE DICTIONARY product_dict (
  product_id UInt32,
  name String,
  category String
)
PRIMARY KEY product_id
SOURCE(CLICKHOUSE(
  HOST 'localhost'
  PORT 9000
  TABLE 'products'
  USER 'default'
))
LIFETIME(MIN 300 MAX 360)
LAYOUT(HASHED());

-- Usage in query
SELECT
  e.event_id,
  dictGet('product_dict', 'name', e.product_id)
    AS product_name
FROM events e
WHERE e.event_date = today();
```

Dictionaries can provide orders of magnitude better performance than JOINs for dimension lookups, especially for frequently used reference data like product catalogs, user attributes, or geographic information.

Secondary Indexes for Non-Primary Key Columns

Data Skipping Indexes

ClickHouse offers specialized secondary indexes designed to skip data blocks that don't match query conditions, even for columns not in the primary key

Index Types

Multiple index types available: minmax (ranges), set (distinct values), bloom filter (probabilistic membership), ngrambf_v1 (text search), and tokenbf_v1 (token search)

Granularity Configuration

Indexes can be configured with different granularities, allowing for trade-offs between index size and precision

Implementation Approach

Add indexes to existing tables with ALTER TABLE or define them during table creation for best performance on non-primary key columns

Secondary indexes in ClickHouse work differently from traditional database indexes. Rather than pointing to individual rows, they store aggregate information about data granules to enable efficient data skipping. This approach maintains ClickHouse's performance focus while providing additional filtering capabilities.

Implementing Secondary Indexes

Bloom Filter Index Example

Ideal for high-cardinality columns with equality checks:

```
ALTER TABLE web_events
ADD INDEX url_index url TYPE bloom_filter(0.01)
GRANULARITY 1024;
```

This creates a bloom filter index on the URL column, configured with a 1% false positive rate and a granularity of 1024 granules. It's effective for queries using equality conditions on URL.

MinMax Index Example

Ideal for range queries on non-primary key columns:

```
ALTER TABLE web_events
ADD INDEX timestamp_idx
  event_timestamp TYPE minmax
GRANULARITY 4;
```

This creates a minmax index on the event_timestamp column, storing the minimum and maximum values for every 4 granules. It helps optimize range queries like BETWEEN or >, < operations.

After adding indexes to an existing table, you need to populate them with: **ALTER TABLE web_events MATERIALIZE INDEX url_index**. For new data, indexes are built automatically during insertion. Use the proper index type based on your query patterns for optimal performance.

N-gram Indexes for Text Search

What Are N-gram Indexes?

N-gram indexes break text into small overlapping segments (typically 3 characters) and create bloom filters to efficiently check for their presence. This enables faster LIKE and substring searches on text columns.

- Optimized for pattern matching in text
- Works with wildcards and partial matches
- More efficient than full column scans

Implementation Example

```
ALTER TABLE web_events
ADD INDEX url_ngram url
TYPE ngrambf_v1(3, 512, 3, 0)
GRANULARITY 4;

ALTER TABLE web_events
MATERIALIZE INDEX
url_ngram;
```

This creates an n-gram index with 3-character grams, 512 hash functions, 3 parts per granule, and no additional processing.

When To Use

Consider using n-gram indexes when:

- Frequently searching with LIKE '%pattern%'
- Working with URL paths or text content
- Needing partial text matching

N-gram indexes can dramatically improve the performance of text search operations, which are otherwise particularly expensive in columnar databases. They're especially valuable for log analysis and URL pattern matching scenarios.

Partitioning Strategy

Define Partition Key

Choose a column that distributes data evenly and matches query patterns, often time-based for analytical data

Balance Granularity

Too many partitions increase management overhead, too few reduce the benefit; aim for partition sizes of 10-100GB

Leverage in Queries

Include partition key in query filters to enable partition pruning, dramatically reducing scan volume

Implement Lifecycle

Use TTL expressions to automate data retention policies at the partition level

Partitioning is a crucial performance optimization technique in ClickHouse. It allows the system to work with smaller, more manageable chunks of data and to completely skip irrelevant partitions during query execution. For time-series data, partitioning by date or month typically provides an excellent balance of manageability and query performance.

Partitioning Implementation

Table Definition with Partitioning

```
CREATE TABLE web_events (
  EventDate Date,
  EventTime DateTime,
  UserID UInt64,
  URL String,
  RegionID UInt32
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(EventDate)
ORDER BY (EventDate, UserID)
SETTINGS index_granularity = 8192;
```

This table is partitioned by year and month using the toYYYYMM function on EventDate. Each month's data will be stored in a separate partition on disk.

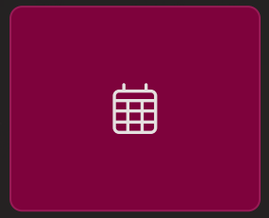
Partition-Aware Query

```
-- This query will only scan January 2023 data
SELECT
  COUNT(*) AS hits,
  uniq(UserID) AS users
FROM web_events
WHERE EventDate BETWEEN
  '2023-01-01' AND '2023-01-31'
  AND URL LIKE '%checkout%';
```

The optimizer will detect that this query only needs to read the January 2023 partition, completely skipping all other months' data.

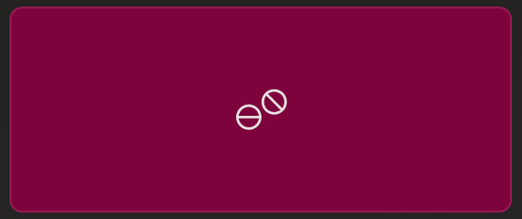
Partitioning provides one of the most significant performance boosts for analytical workloads with time-based data. Combined with a proper primary key design, it enables ClickHouse to minimize the amount of data read from disk, resulting in dramatic query speed improvements.

Data Lifecycle Management with TTL



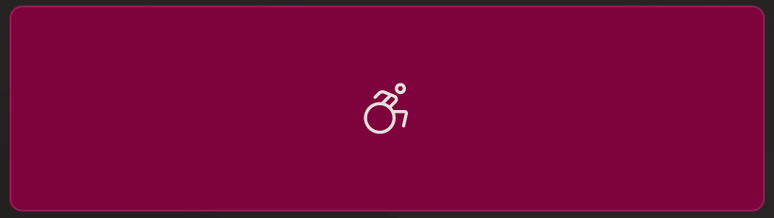
Column-Level TTL

Set expiration for individual column values



Row-Level TTL

Delete entire rows after specified time period



Move TTL

Migrate aging data to different storage tiers

Partition-Level Management

Drop or archive entire partitions based on age

ClickHouse provides sophisticated options for automating data lifecycle management. Using TTL (Time To Live) expressions, you can implement automatic data aging policies that maintain optimal performance by managing data volume while preserving accessibility according to business requirements.

These capabilities are especially valuable for time-series data where historical information gradually becomes less valuable and can be either deleted or moved to less expensive storage.

TTL Implementation Examples

Table with Multiple TTL Rules

```
CREATE TABLE web_events (
  EventDate Date,
  EventTime DateTime,
  UserID UInt64,
  URL String,
  UserAgent String,
  FullSessionData String
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(EventDate)
ORDER BY (EventDate, UserID)
TTL
  -- Delete rows after 2 years
  EventDate + INTERVAL 2 YEAR,
  -- Clear detailed session data after 3 months
  EventDate + INTERVAL 3 MONTH
    TO DISK 'cold',
  -- Remove UserAgent after 6 months
  EventDate + INTERVAL 6 MONTH
    DELETE UserAgent, FullSessionData;
```

Altering Existing Table to Add TTL

```
-- Add TTL to move old partitions to cold storage
ALTER TABLE web_events
  MODIFY TTL EventDate + INTERVAL 1 YEAR
  TO VOLUME 'cold_volume';

-- Add column-level TTL to clear detailed data
ALTER TABLE web_events
  MODIFY COLUMN FullSessionData
  TTL EventDate + INTERVAL 3 MONTH;
```

TTL rules are evaluated during merges, so you may need to trigger a merge manually using OPTIMIZE TABLE if you want the rules to apply immediately.

TTL expressions provide a powerful, automated approach to managing data lifecycle in ClickHouse. They allow you to implement sophisticated data retention and storage tiering policies without manual intervention.

Materialized Columns

What Are Materialized Columns?

Materialized columns store pre-computed values derived from other columns. They are calculated once during insertion rather than each time a query runs.

- Store complex expression results
- Computed and stored on data insertion
- Trade storage for query performance

Implementation

```
CREATE TABLE web_events (
  EventTime DateTime,
  URL String,
  ParsedPath String
    MATERIALIZED
extractURLPathAndQuery(URL)
) ENGINE = MergeTree()
ORDER BY (EventTime);
```

The ParsedPath column is automatically populated by applying the extractURLPathAndQuery function to the URL column during insertion.

Best Use Cases

Ideal for:

- Frequently used complex calculations
- Parsing or extraction operations
- When the derived value is smaller than the original data

Materialized columns significantly improve query performance for commonly used expressions by trading increased storage space for reduced computation time. They're especially valuable for complex parsing or transformation operations that would otherwise be repeated in multiple queries.

Avoiding Common Pitfalls

Avoid SELECT *

Always specify only the columns you need to reduce I/O and memory usage. This is especially important in a columnar database like ClickHouse.

Beware Functions on Indexed Columns

Applying functions to indexed columns prevents index usage. Restructure queries to apply functions to constants instead.

Monitor Disk I/O

ClickHouse is often I/O bound. Ensure your storage system provides sufficient throughput for your workload.

Use Proper ORDER BY in Queries

Without an ORDER BY clause, results may arrive in unpredictable order, requiring additional sorting. Specify ordering when needed.

Avoiding these common pitfalls can help prevent unexpected performance issues in your ClickHouse deployment. Many performance problems stem from query patterns that don't align well with ClickHouse's architecture and optimization strategies.

Monitoring Query Performance

Query Log Tables

ClickHouse maintains detailed query logs in system.query_log containing execution time, read rows, memory usage, and more. Use these logs to identify slow or resource-intensive queries.

Processes Tables

The system.processes table shows currently running queries, allowing you to monitor long-running operations and potentially kill problematic queries.

Metrics Tables

System tables like system.metrics and system.asynchronous_metrics provide comprehensive performance indicators about your ClickHouse instance.

Events Tables

The system.events table tracks various events and counters, helping you understand system behavior and identify potential bottlenecks.

ClickHouse provides rich introspection capabilities through its system tables. Regular monitoring of these tables can help you identify performance issues early and understand the resource usage patterns of your queries.

Analyzing Slow Queries

Finding Slow Queries

```
SELECT
  query_duration_ms,
  query,
  read_rows,
  read_bytes,
  memory_usage
FROM system.query_log
WHERE type = 'QueryFinish'
  AND query_duration_ms > 1000
  AND event_date = today()
ORDER BY query_duration_ms DESC
LIMIT 10;
```

This query identifies the ten slowest queries from today that took more than 1 second to execute, showing their duration, the number of rows read, bytes processed, and memory usage.

Key Metrics to Analyze

- **read_rows**: High values indicate insufficient data skipping
- **read_bytes**: Amount of data read from disk
- **memory_usage**: Peak memory usage during execution
- **result_rows**: Number of rows in the result set
- **query_duration_ms**: Total execution time

Look for queries reading many more rows than they return, as these are prime candidates for optimization through better indexing or query restructuring.

Regular analysis of slow queries is essential for maintaining optimal performance in ClickHouse. By understanding which queries consume the most resources and why, you can focus your optimization efforts on the areas that will provide the greatest benefits.

Using EXPLAIN for Query Analysis

1

EXPLAIN Syntax Formats

ClickHouse offers multiple EXPLAIN formats for different levels of detail

2

Query Pipeline Analysis

Understand how data flows through processing steps

3

Optimization Decision Review

See which indexes and optimizations are being applied

The EXPLAIN command is a powerful tool for understanding how ClickHouse executes your queries. It helps you identify potential optimization opportunities by showing which indexes are being used, how tables are being read, and how data flows through the query pipeline.

```
-- Basic syntax examples:
EXPLAIN syntax SELECT ... FROM ...;
EXPLAIN pipeline SELECT ... FROM ...;
EXPLAIN indexes SELECT ... FROM ...;
EXPLAIN SETTINGS analyze_expressions=1 SELECT ... FROM ...;
```

Understanding the query execution plan is crucial for diagnosing performance issues and validating that your optimization techniques (like indexing and filtering) are working as expected.

EXPLAIN Pipeline Example

Query Example

```
EXPLAIN pipeline
SELECT
  EventDate,
  COUNT(*) AS hits,
  uniq(UserID) AS visitors
FROM web_events
WHERE
  EventDate BETWEEN
    '2023-01-01' AND '2023-01-31'
  AND URL LIKE '%checkout%'
GROUP BY EventDate
ORDER BY EventDate;
```

Understanding the Output

The pipeline output shows:

1. Reading from MergeTree with filters
2. Data filtration steps
3. Aggregation operations
4. Sorting of results
5. Result formation

Look for full table scans, excessive reads, or missing index usage as indicators of potential optimization opportunities.

The EXPLAIN pipeline command provides a detailed view of how data flows through the query execution pipeline. It can help you understand where time is being spent during query execution and identify bottlenecks such as excessive data reads or inefficient operations that might benefit from additional indexing or query restructuring.

EXPLAIN Indexes Example

Query Example

```
EXPLAIN indexes
SELECT COUNT(*)
FROM web_events
WHERE
  EventDate = '2023-01-15'
  AND RegionID = 42
  AND URL LIKE '%checkout%';
```

Understanding the Output

The indexes output shows:

1. Which primary key columns are used for filtering
2. How conditions are applied to the index
3. Which secondary indexes might be engaged
4. The estimated data read reduction from indexing

Pay attention to conditions marked as "NONE" (not using an index) as these may be candidates for additional indexing or query restructuring.

The EXPLAIN indexes command specifically focuses on how ClickHouse uses indexes to optimize data access for your query. It helps you confirm that your primary key design and secondary indexes are effectively reducing the amount of data that needs to be read for query execution.

Query Profiling with EXPLAIN analyze

EXPLAIN analyze Syntax

```
EXPLAIN analyze
SELECT ...
FROM ...
WHERE ...
```

The analyze format provides detailed profiling information about query execution, including time spent in each execution stage.

Interpreting Results

Key metrics in the output:

- Execution time for each stage
- Rows processed at each step
- Memory usage during execution
- CPU and real time for operations

Optimization Opportunities

Look for:

- Steps with disproportionate execution time
- Excessive row processing before filtering
- High memory usage operations
- Unexpected data volume in intermediate steps

The analyze mode provides the most detailed profiling information about query execution, making it invaluable for pinpointing performance bottlenecks in complex queries. Use it when you need to understand exactly where time is being spent during query execution and which operations might benefit from optimization.

Memory Usage Optimization

Common Memory Issues

ClickHouse operations that often consume significant memory:

- GROUP BY on high-cardinality columns
- DISTINCT operations on large datasets
- ORDER BY without a LIMIT clause
- Large JOIN operations
- Complex analytical functions

Mitigation Strategies

Techniques to reduce memory consumption:

- Use approximate aggregation functions (e.g., uniqHLL12 instead of uniq)
- Apply LIMIT clauses when ordering
- Leverage external sorting with max_bytes_before_external_sort
- Enable partial aggregation with partial_aggregation_memory_efficient=1
- Filter data before joins or aggregations

Memory optimization is crucial for ClickHouse performance, especially when dealing with large datasets. By understanding which operations consume memory and applying appropriate settings and query patterns, you can avoid out-of-memory errors and keep your system running efficiently even under heavy load.

Data Type Optimization

Use Specific Numeric Types

Choose the narrowest numeric type that can accommodate your data range (UInt8/16/32/64, Int8/16/32/64) to improve compression and performance

Date vs. DateTime

Use Date for dates without time components to save storage and improve filtering performance, as Date consumes 2 bytes vs 4 bytes for DateTime

String Alternatives

Consider FixedString for fixed-length strings, Enum for limited sets of string values, and LowCardinality for columns with many repeated values

LowCardinality Decorator

Apply LowCardinality to columns with fewer than 10,000 unique values to dramatically improve query performance and compression

Choosing appropriate data types is a low-effort, high-impact optimization that improves both storage efficiency and query performance. ClickHouse offers specialized types that can provide significant benefits when matched correctly to your data characteristics.

LowCardinality Data Type

10x

Query Performance

Potential speedup for operations on columns with repeated values

2-10x

Compression Ratio

Improved storage efficiency compared to regular String types

<10K

Ideal Distinct Values

Optimal number of unique values for maximum benefit

The LowCardinality data type is a special decorator in ClickHouse that implements dictionary encoding for string and numeric columns with a limited number of distinct values. It stores unique values in a dictionary and references them with small integers, dramatically reducing storage requirements and improving query performance.

```
-- Example implementation
CREATE TABLE web_events (
  EventDate Date,
  URL String,
  Browser LowCardinality(String),
  OS LowCardinality(String),
  RegionName LowCardinality(String)
) ENGINE = MergeTree()
ORDER BY (EventDate, URL);
```

For columns like browser names, operating systems, country names, or status codes, LowCardinality can provide dramatic performance improvements with minimal implementation effort.

Compression Strategies

Codec Selection

ClickHouse offers multiple compression codecs: LZ4 (default), ZSTD, Delta, Gorilla, DoubleDelta, and more. Each codec performs differently depending on data characteristics.

Column-Level Settings

Compression can be configured at the column level, allowing you to optimize each column based on its data pattern. This granular approach maximizes overall compression efficiency.

Codec Combinations

Codecs can be combined in sequence for better results. For example, Delta encoding followed by ZSTD often works well for numeric time series data with small changes.

Performance Trade-offs

Higher compression ratios typically come with increased CPU usage. Balance storage savings against processing overhead based on your workload characteristics.

Effective compression strategy can significantly reduce storage requirements and improve I/O performance by reducing the amount of data that needs to be read from disk. ClickHouse's columnar storage makes it particularly amenable to high compression ratios since similar values are stored together.

Compression Implementation Examples

Table Creation with Codecs

```
CREATE TABLE sensor_readings (
  sensor_id UInt32,
  timestamp DateTime,

  -- For sequential integers with small
  -- differences, Delta works well
  measurement_id UInt32 CODEC(Delta, ZSTD),

  -- For floating-point time series,
  -- Gorilla is effective
  temperature Float64 CODEC(Gorilla, ZSTD),

  -- For high-compression needs
  humidity Float64 CODEC(ZSTD(3)),

  -- For categorical string data
  status LowCardinality(String)
) ENGINE = MergeTree()
ORDER BY (sensor_id, timestamp);
```

Altering Existing Columns

```
-- Modify compression for an existing column
ALTER TABLE sensor_readings
MODIFY COLUMN temperature
CODEC(Gorilla, ZSTD);

-- Check compression ratio
SELECT
  column,
  formatReadableSize(data_compressed_bytes) AS
compressed,
  formatReadableSize(data_uncompressed_bytes) AS
uncompressed,
  round(data_uncompressed_bytes /
data_compressed_bytes, 2) AS ratio
FROM system.columns
WHERE table = 'sensor_readings'
ORDER BY ratio DESC;
```

Choosing the right compression codec for each column based on its data characteristics can significantly improve both storage efficiency and query performance. Monitor compression ratios and query performance to find the optimal balance for your specific workload.

ClickHouse Insert Performance

Insert in Batches

ClickHouse performs best with larger batch inserts typically between 1,000 and 1,000,000 rows, which amortize overhead across more rows

Use INSERT ... SELECT

For data transformations, perform them during insertion rather than as separate steps to reduce overhead and improve performance

Buffer Tables

Consider Buffer table engine for accumulating small inserts before writing to the main table, improving insert throughput for frequent small operations

Async Inserts

Enable async_insert=1 and wait_for_async_insert=0 for non-blocking inserts that improve throughput for high-frequency insert workloads

ClickHouse is optimized for analytical workloads, which typically involve bulk data loading followed by complex queries. Its insert performance can be excellent when following these best practices, but it's important to align your data ingestion patterns with ClickHouse's strengths.

Buffer Tables for Insert Optimization

How Buffer Tables Work

Buffer tables temporarily store inserted data in memory and periodically flush it to the target table in larger batches. This approach dramatically improves performance for frequent small inserts.

Key characteristics:

- In-memory data accumulation
- Configurable flush conditions
- Automatic batch formation
- Transparent queries across buffer and target

Implementation Example

```
CREATE TABLE events_buffer AS events
ENGINE = Buffer(default, events, 16, 10, 100, 10000,
1000000, 10000000, 100000000);

-- Insert directly to buffer table
INSERT INTO events_buffer VALUES (...);

-- Queries automatically combine data from buffer
-- and main table
SELECT COUNT(*) FROM events_buffer
WHERE EventDate = today();
```

The buffer parameters control the number of shards and various thresholds for flushing based on time, rows, and bytes.

Buffer tables provide an excellent solution for scenarios requiring frequent small inserts while maintaining the analytical query performance benefits of ClickHouse. They're particularly valuable for real-time data ingestion pipelines that need to handle many small batches of incoming data.

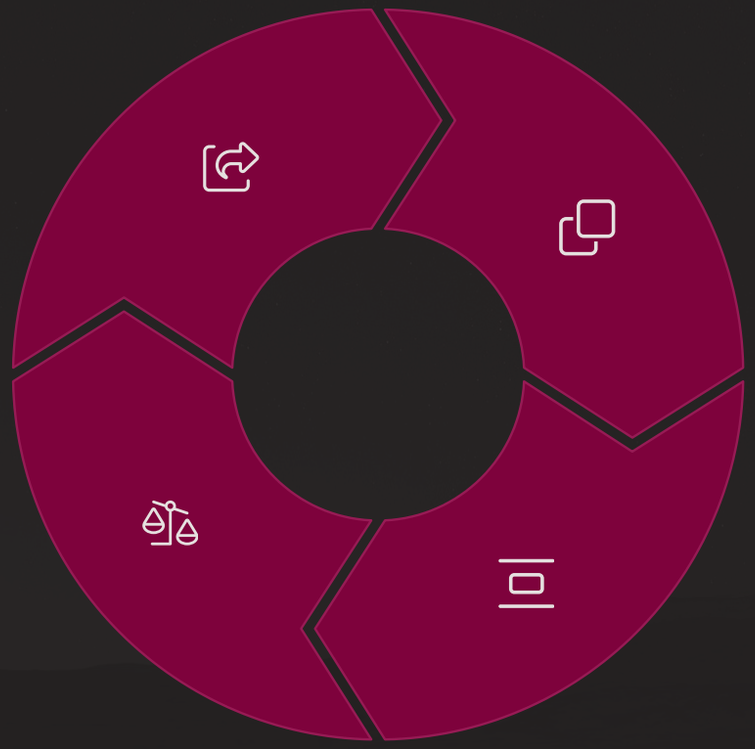
Distributed Table Design

Sharding Strategy

Distribute data across shards based on access patterns and data locality

Replication Design

Ensure data reliability with appropriate replication factor



Load Balancing

Configure for even workload distribution across cluster nodes

Distributed Tables

Create abstraction layer for transparent querying across shards

When scaling ClickHouse across multiple servers, proper distributed table design is essential for performance and reliability. ClickHouse uses a shared-nothing architecture where each shard operates independently but can be queried together through distributed tables.

The key to effective distributed performance is choosing an appropriate sharding key that distributes data evenly while allowing most queries to target a minimal number of shards. Distributing by date is common for time-series data but can lead to hotspots on recent dates.

Distributed Table Implementation

Configuration Steps

1. Define cluster in config.xml
2. Create local tables on each shard
3. Create distributed table as virtual view
4. Insert data through distributed table

The distributed table acts as a view across all shards, automatically routing queries and inserts to the appropriate servers based on the sharding scheme.

Implementation Example

```
-- On each shard, create local table
CREATE TABLE events_local ON CLUSTER
'cluster_name' (
  EventDate Date,
  UserID UInt64,
  EventType String
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(EventDate)
ORDER BY (EventDate, UserID);

-- Create distributed table
CREATE TABLE events_distributed ON CLUSTER
'cluster_name' (
  EventDate Date,
  UserID UInt64,
  EventType String
) ENGINE = Distributed(
  cluster_name,
  default,
  events_local,
  cityHash64(UserID)
);

-- Insert through distributed table
INSERT INTO events_distributed VALUES (...);
```

The Distributed engine provides a transparent interface for working with sharded data. It automatically routes queries to the appropriate shards and merges results, while distributing inserted data according to the specified sharding key function.

Aggregation Performance Optimization

Aggregate Function Data Types

Store pre-aggregated states using AggregateFunction data type to efficiently update aggregations incrementally without reprocessing all data.

- Enables incremental aggregation
- Significantly reduces computation for frequent updates
- Works with most aggregate functions

Approximate Aggregation

Use approximation functions for high-cardinality aggregations with large data volumes.

- uniqHLL12() instead of uniq()
- quantileTDigest() instead of quantile()
- Trades small accuracy for major performance gains

Materialized Views for Pre-Aggregation

Create materialized views that pre-compute common aggregations for faster query responses.

- Automatically updated on data insertion
- Dramatically speeds up repeated aggregation queries
- Perfect for dashboards and recurring reports

Aggregation queries can be particularly resource-intensive, especially with large datasets. These optimization techniques can significantly improve performance for analytical workloads that rely heavily on aggregation operations.

AggregateFunction Implementation

Materialized View with AggregateFunction

```
-- Create materialized view with aggregate states
CREATE MATERIALIZED VIEW events_hourly_stats
ENGINE = AggregatingMergeTree()
PARTITION BY toYYYYMM(hour)
ORDER BY (hour, event_type)
AS SELECT
  toStartOfHour(EventTime) AS hour,
  EventType AS event_type,
  countState() AS events,
  uniqState(UserID) AS users
FROM events
GROUP BY hour, event_type;

-- Query using -Merge functions to finalize
SELECT
  hour,
  event_type,
  countMerge(events) AS event_count,
  uniqMerge(users) AS user_count
FROM events_hourly_stats
WHERE hour >= yesterday()
GROUP BY hour, event_type
ORDER BY hour;
```

Benefits of This Approach

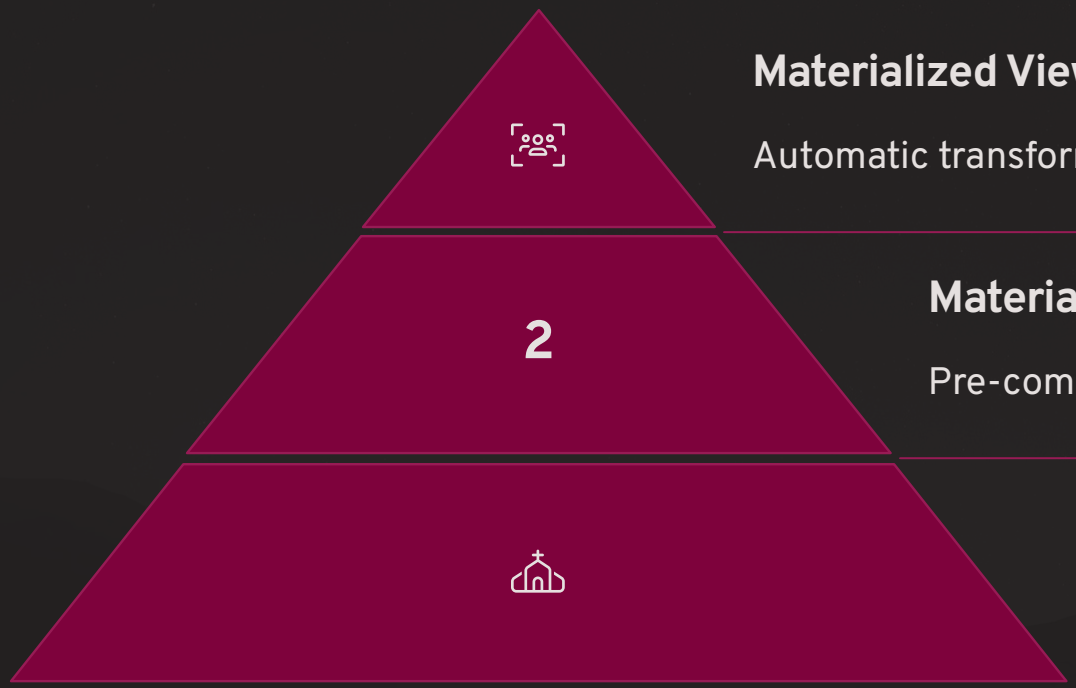
This implementation provides significant advantages:

- Pre-computed aggregation states dramatically speed up queries
- Incremental updates avoid full recalculation when new data arrives
- Storage is much more efficient than storing final aggregation results
- Supports re-aggregation at query time with different grouping

The AggregatingMergeTree engine automatically merges states for the same keys during background merges, keeping the aggregation states up-to-date and efficient.

AggregateFunction data types and the AggregatingMergeTree engine provide a powerful solution for scenarios requiring both frequent data ingestion and fast aggregation queries, such as real-time analytics dashboards.

Materializing Complex Calculations



Materialized Views

Automatic transformation and aggregation

Materialized Columns

Pre-computed expressions stored with data

Query Caching

Session-level caching for repeated queries

When working with complex calculations or transformations that are frequently used in queries, materializing the results can significantly improve performance. ClickHouse offers several approaches, each with different trade-offs in terms of storage, update complexity, and query flexibility.

Materialized views automatically transform and potentially aggregate data as it's inserted, creating derived tables optimized for specific query patterns. Materialized columns pre-compute expressions at the row level during insertion. Query caching provides temporary performance improvements for repeated identical queries within the same session.

Schema Evolution and Maintenance

Adding New Columns

ClickHouse makes adding columns very efficient, as they're stored separately on disk. New columns can be added with default values without rewriting existing data.

Managing Table Merges

Background merges can impact query performance. Schedule major operations during off-peak hours and monitor merge progress through system tables.

Data Mutations

UPDATE and DELETE operations are implemented as mutations that rewrite data parts asynchronously. Monitor and manage these operations carefully.

Partition Management

Use DETACH, ATTACH, DROP, and REPLACE PARTITION commands for efficient data lifecycle management, particularly for time-series data.

ClickHouse is designed primarily for append-only analytical workloads, but it does provide mechanisms for schema evolution and data maintenance. Understanding these operations' performance implications and monitoring their progress is essential for maintaining optimal performance during maintenance activities.

Replication and High Availability

ReplicatedMergeTree Engine

Built-in replication mechanism that ensures data consistency across multiple servers. Provides automatic failover and recovery capabilities while maintaining the performance advantages of MergeTree.

Synchronization Mechanism

Uses ZooKeeper to coordinate insert operations and metadata changes across replicas. Provides the option for synchronous or asynchronous inserts depending on durability requirements.

Monitoring Replication

System tables provide detailed visibility into replication status, including lag, queue size, and potential problems. Regular monitoring is essential for ensuring system health.

Load Balancing

Distribute read queries across replicas to improve throughput and reduce load on individual servers. Write operations can be directed to any replica with synchronization handled automatically.

ClickHouse replication is designed to provide high availability while maintaining excellent query performance. It handles the challenges of distributed consistency without significant overhead, making it suitable for demanding analytical workloads that require both performance and reliability.

Replication Implementation

Replicated Table Creation

```
CREATE TABLE events_replicated (
  EventDate Date,
  EventTime DateTime,
  UserID UInt64,
  EventType String
) ENGINE = ReplicatedMergeTree(
  '/clickhouse/tables/{shard}/events',
  '{replica}'
)
PARTITION BY toYYYYMM(EventDate)
ORDER BY (EventDate, UserID);
```

The two string parameters specify the ZooKeeper path for table metadata and the replica identifier. These ensure proper coordination across all replica servers.

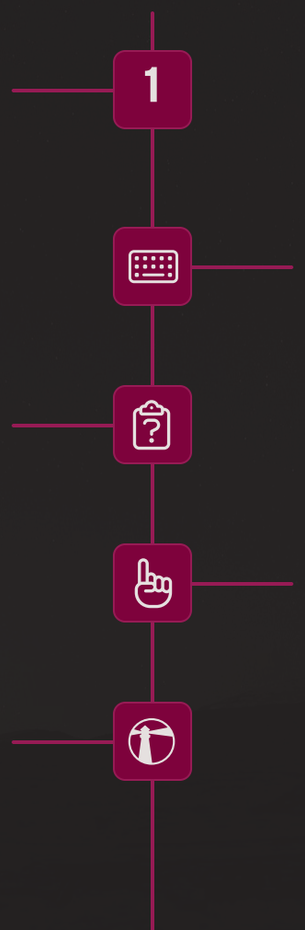
Monitoring Replication Status

```
-- Check for replication delays
SELECT
  database,
  table,
  is_leader,
  total_replicas,
  active_replicas,
  future_parts,
  parts_to_check,
  queue_size,
  inserts_in_queue,
  merges_in_queue
FROM system.replicas
WHERE active_replicas < total_replicas
  OR queue_size > 20
ORDER BY queue_size DESC;
```

Regular monitoring of replication metrics helps identify potential issues before they impact system availability or performance.

ClickHouse replication is a robust solution for ensuring data availability and durability across multiple servers. The ReplicatedMergeTree engine handles all aspects of keeping data synchronized, allowing you to focus on your analytical workloads rather than complex replication management.

Key Takeaways: Optimizing ClickHouse Performance



Leverage ClickHouse Architecture

Design with column-oriented storage and sparse indexing in mind

Optimize Primary Key Design

Order columns by cardinality and match query patterns

Write Efficient Queries

Filter early, select only required columns, and optimize JOINs

Implement Secondary Indexes

Use specialized indexes for non-primary key columns

Monitor and Analyze

Continuously monitor performance and use EXPLAIN to optimize queries

Optimizing ClickHouse performance requires understanding its unique architecture and applying specific design principles. By following these best practices for schema design, indexing strategy, and query patterns, you can achieve exceptional performance for your analytical workloads.

Remember that optimization is an ongoing process. Continuously monitor your system's performance, analyze query patterns, and refine your approach as your data volume and query requirements evolve.