Arise
by INFINITAS

ClickHouse

Mastering ClickHouse: A Comprehensive Guide

Query **billions** of rows in milliseconds
Fastest and most resource efficient open-source database for real-time apps and analytics.

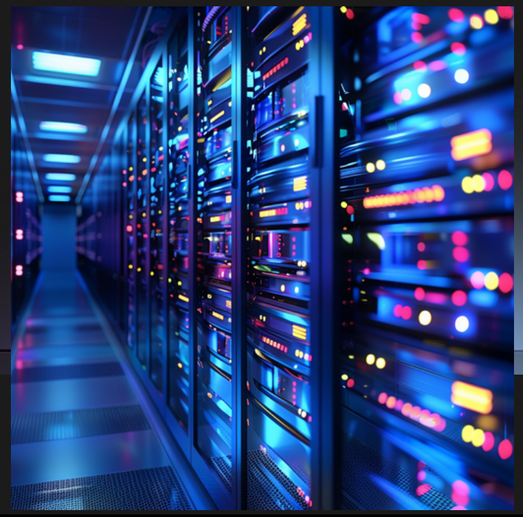
Apaichon Punopas
Technical Coach

ClickHouse

Arise by INFINITAS

Course Outline

Fastest and most resource efficient open-source database for real-time apps and analytics.



MODULE 1 : INTRODUCTION

1.1 Overview

- What is Clickhouse ?
- Use cases and advantages
- Comparison with other databases

1.2 Installation and Setup

- System requirements
- Installation methods

1.3 Basic Concepts

- Databases and Tables
- Data types and Compression
- Partitions and Sorting
- Replication and Sharding

ClickHouse

Arise by INFINITAS

Course Outline

Fastest and most resource efficient open-source database for real-time apps and analytics.

MODULE 2 : CLICKHOUSE QUERY LANGUAGE (CHQL)

2.1 Basic Queries

- Insert statement
- SELECT statement
- WHERE clause
- ORDER BY and LIMIT

2.2 Advanced Queries

- JOIN operations
- Subqueries
- Aggregation functions

2.3 Performance Optimization

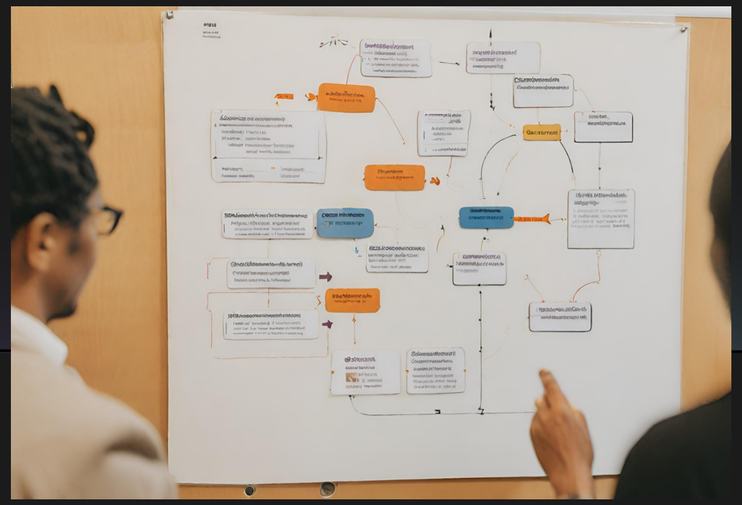
- Indexing
- Query profiling
- Using the EXPLAIN statement

ClickHouse

Arise by INFINITAS

Course Outline

Fastest and most resource efficient open-source database for real-time apps and analytics.



MODULE 3 : CLICKHOUSE DATA MODELING

3.1 Schema Design

- Choosing the right data types
- Designing efficient tables
- Normalization and denormalization

3.2 Integration Methods

- Using the ClickHouse client libraries
- Integrating with admin tools

3.3 Data Transformation

- Views
- Materialized views
- Aggregating tables
- Custom transformations using SQL functions
- Custom transformations using Other language

ClickHouse

Arise by INFINITAS

Course Outline

Fastest and most resource efficient open-source database for real-time apps and analytics.

MODULE 4 : CLICKHOUSE PERFORMANCE TUNING

4.1 Indexing Strategies
- Key indexing strategies
- Indexing best practices

4.2 Configuration Optimization
- Memory settings
- Disk settings
- Query parallelism

4.3 Monitoring and Diagnostics
- System tables
- Logs
- Profiling
- Using external monitoring tools

ClickHouse

Arise by INFINITAS

Course Outline

Fastest and most resource efficient open-source database for real-time apps and analytics.

MODULE 5 : CLICKHOUSE ADMINISTRATION

5.1 Backup and Recovery
- ClickHouse backup methods
- Point-in-time recovery

5.2 Security
- User authentication and authorization
- Encryption and SSL/TLS setup
- Securing ClickHouse clusters

5.3 High Availability and Scaling
- Setting up replication
- Sharding strategies
- Managing distributed clusters

ClickHouse

Course Outline

Fastest and most resource efficient open-source database for real-time apps and analytics.

Arise by INFINITAS

MODULE 6 : CLICKHOUSE ADVANCED FEATURES

6.1 Real-time Analytics
- Aggregating live data
- Integrating with streaming platforms

6.2 Custom Functions and UDFs
- Writing and using User Defined Functions
- Leveraging ClickHouse extensions

6.3 Integration with External Tools
- ClickHouse and popular BI tools
- Using ClickHouse with data visualization tools

ClickHouse

Arise by INFINITAS

Course Outline

Fastest and most resource efficient open-source database for real-time apps and analytics.

MODULE 7: CASE STUDIES AND BEST PRACTICES

7.1 Real-world Use Cases

- Analyzing large datasets
- Implementing ClickHouse in production environments

7.2 Best Practices

- Optimization techniques
- Troubleshooting common issues
- ClickHouse community and resources

Arise by INFINITAS

Repository

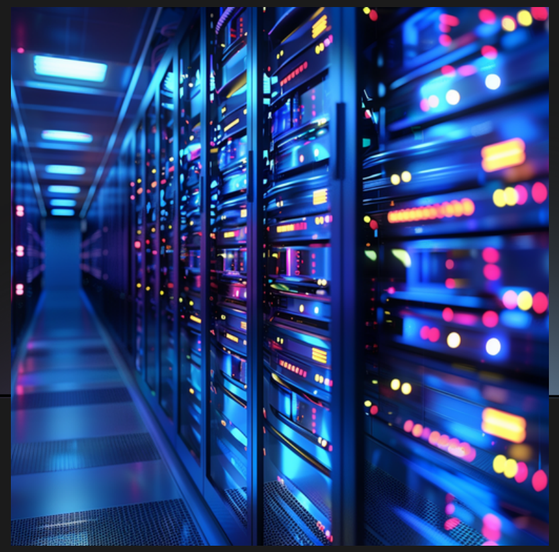
https://github.com/apaichon/clickhouse-tutorial

ClickHouse

Arise by INFINITAS

Module: 1

Introduction



MODULE 1 : INTRODUCTION

1.1 Overview

- What is Clickhouse ?
- Use cases and advantages
- Comparison with other databases

1.2 Installation and Setup

- System requirements
- Installation methods

1.3 Basic Concepts

- Databases and Tables
- Data types and Compression
- Partitions and Sorting
- Replication and Sharding

Arise by INFINITAS

What is Clickhouse ?

ClickHouse is an **open-source, column-oriented database management system (DBMS)** designed specifically for **online analytical processing (OLAP)**. This means it excels at handling and analyzing large datasets, often in real-time, to answer complex analytical questions.

OLAP

- Real-time responses
- Large datasets for complex analytical

COLUMN-ORIENTED

- Data is stored in columns
- Values from the same columns stored together.

REAL-TIME ANALYTICS

- Optimized for real-time data processing.
- Handle continuous data ingestion and querying.

SQL - QUERY LANGUAGE

- 50 years old
- Knowledge base
- Quick learning
- Popularity

INTEGRATIONS

S3, GCS, Kafka, BigQuery, Snowflake, Postgres, MySQL, dbt, Redshift ,etc

FORMATS

- Binary
- CSV and TSV
- JSON

Alexey Milovidov
Co-Founder & CTO

2009 : team start an experimental project determined to generate analytical reports in real-time from non-aggregated data which also grows in real-time.

2021: Series B funding round totals $250M and establishes a $2B valuation.

Arise by INFINITAS

Use cases and advantages

REAL-TIME ANALYTICS

vimeo | CLOUDFLARE
Microsoft | lyft
Contentsquare | highlight.io

LOGS, EVENTS, & TRACES

ROKT | Deutsche Bank
QuickCheck | NANO CORP.
TrillaBit | HIFI

BUSINESS INTELLIGENCE

Darwinium | Uber
ebay | BENOCS
resmo | zomato

MACHINE LEARNING & GENAI

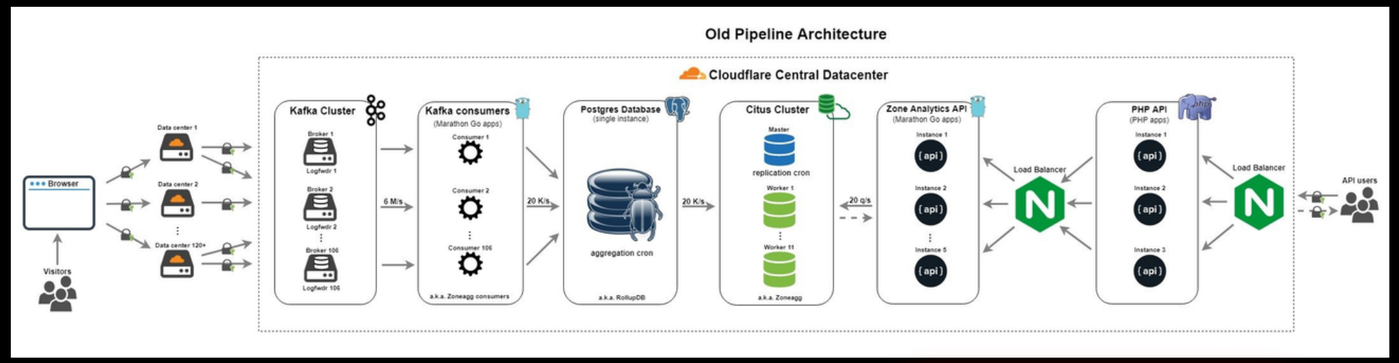
denic | DeepL
ADMIXER
ensemble

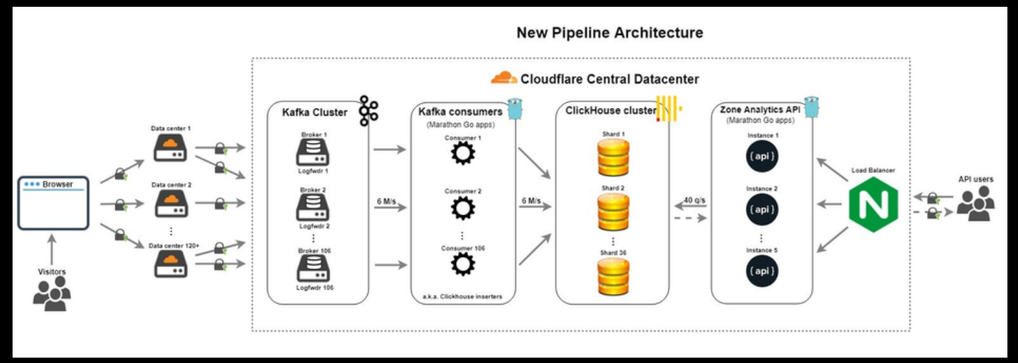
CLOUDFLARE

Arise by INFINITAS

Real-time analytics

HTTP Analytics for 6M requests per second using ClickHouse **100+ columns**





Metric	Cap'n Proto	Cap'n Proto (zstd)	ClickHouse
Avg message/record size	1630 B	360 B	36.74 B
Storage requirements for 1 year	273.93 PiB	60.5 PiB	18.52 PiB (RF x3)
Storage cost for 1 year	$28M	$6.2M	$1.9M

- Architecture of new pipeline is much simpler
- Fault-tolerant
- 7M+ customers' domains
- Totalling more than 2.5 billion monthly unique visitors
- Over 1.5 trillion monthly page views

ref : https://blog.cloudflare.com/http-analytics-for-6m-requests-per-second-using-clickhouse/o grows in real-time.

UBER

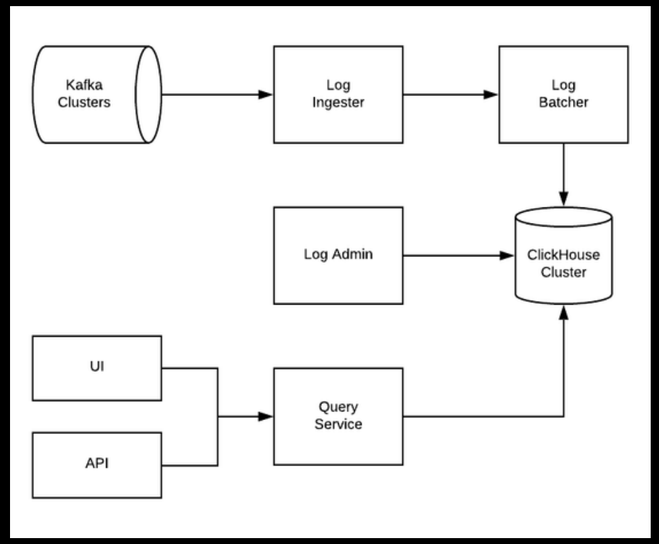
Arise by INFINITAS

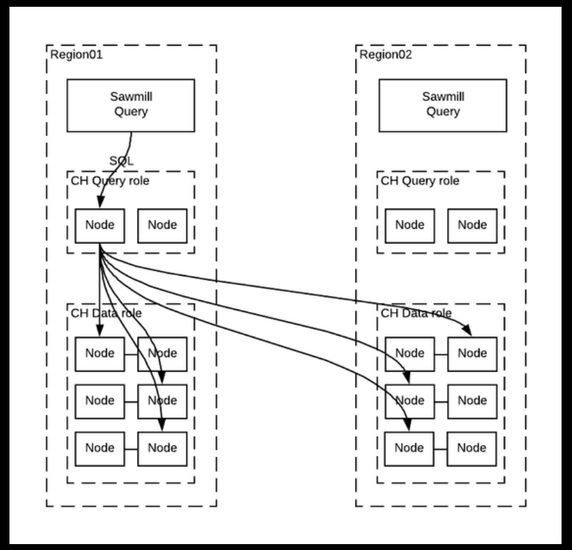
Logs, Events, & Traces

Fast and Reliable Schema-Agnostic Log Analytics Platform

Old Problems

- **Log schema** - Incompatible field types cause error in ES.
- **Operational cost** - run 20+ ES clusters, each region limit heavy queries, sometime cluster freeze, had to restart.
- **Hardware Cost** - Indexing fields is pretty costly in ES.
- **Aggregation -** more than 80% queries are aggregation but ES not designed to support fast aggreations across large datasets.





New Solutions

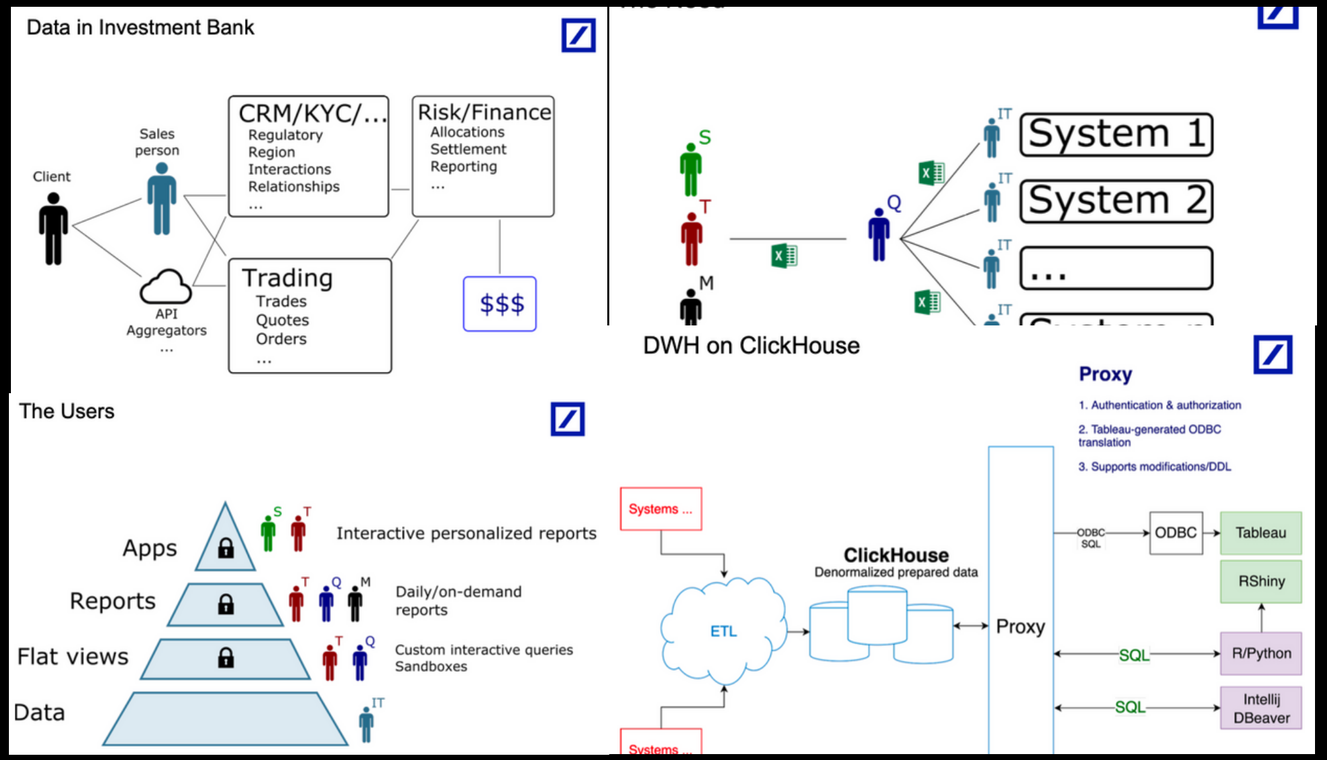
- **Functionalities**
  - **Schema-agnostic**
  - **Effiecient support of aggregation queries**
  - **Support muti-region and cross-tenant queries**
- **Efficiency and maintenance**
  - Support multi-tenant properly to consolidate deployment
  - Reduce cost and be able to handle 10X scale
  - Improve reliability and simplify operation
- **Transparent** migration from ELK

ref : https://www.uber.com/en-TH/blog/logging/

Deutsche Bank

Business Intelligence

How we built BI on Clickhouse with row-level security in Deutsche Bank Technology Centre



- **Data in Investment Bank**
  - Natural data silos and BI
- **Data warehouse**
  - ClickHouse
  - Data driven access control: ABAC
  - SQL hardening

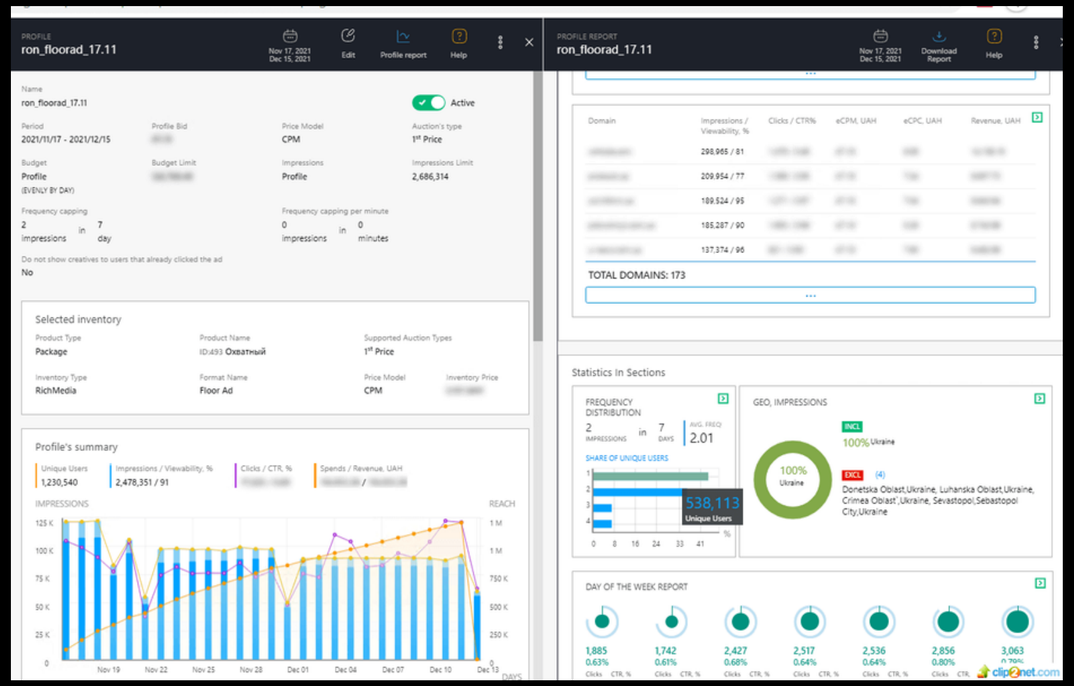
ref : https://blog.cloudflare.com/http-analytics-for-6m-requests-per-second-using-clickhouse/o grows in real-time.

ADMIXER

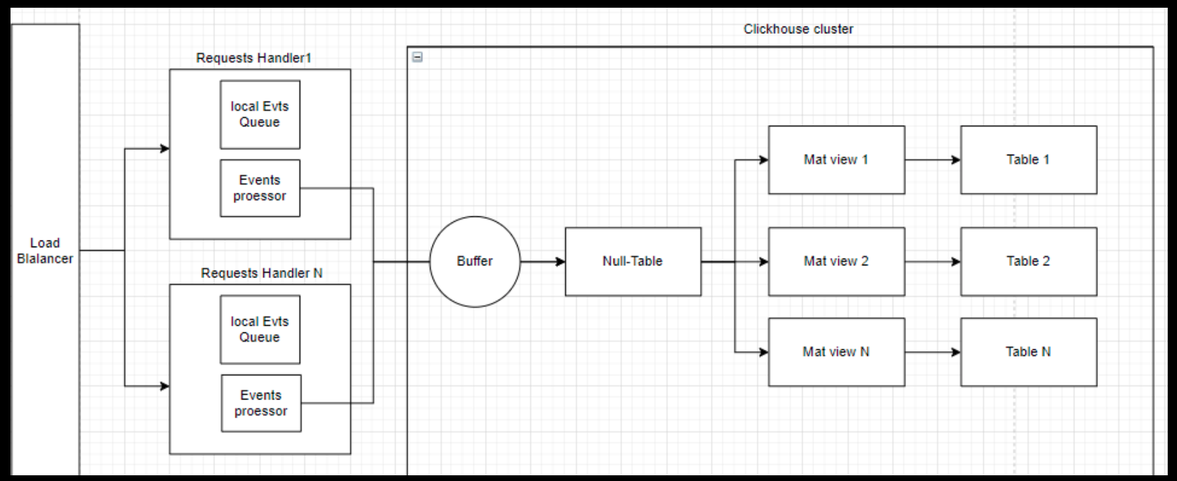
Arise by INFINITAS

Machine Learning & GenAI

Admixer Aggregates Over 1 Billion Unique Users a Day using ClickHouse



- Inserting around 100 billion records per day, over 1 million records per second
- Able to aggregate over 1 billion unique users a day
- Moved from MSSQL to Azure Table Storage to ClickHouse
- ClickHouse is deployed on 15 servers with 2 TB total RAM

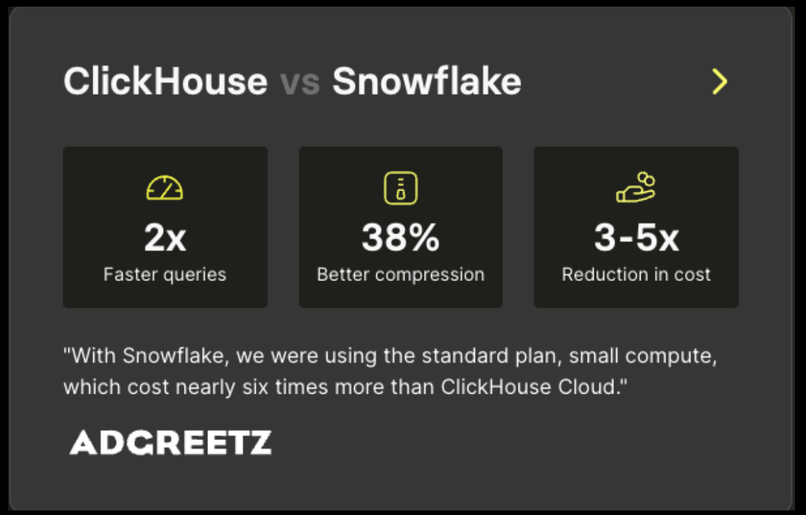


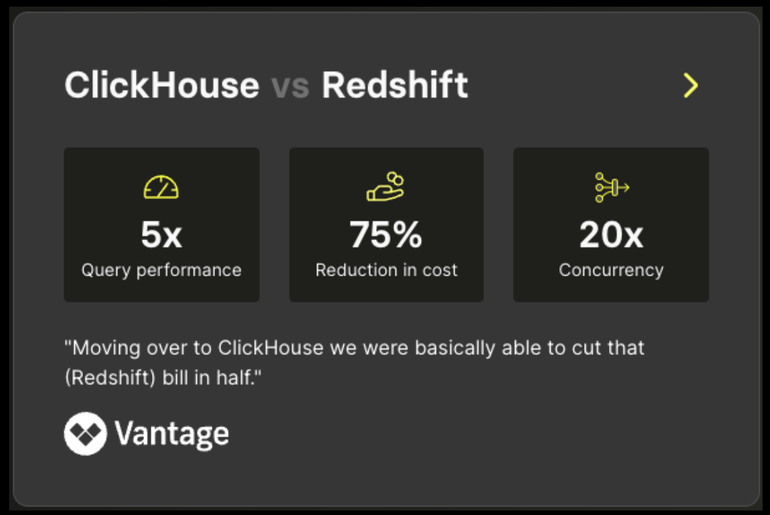
ref : https://clickhouse.com/blog/admixer-aggregates-over-1-billion-unique-users-a-day-using-clickhouse

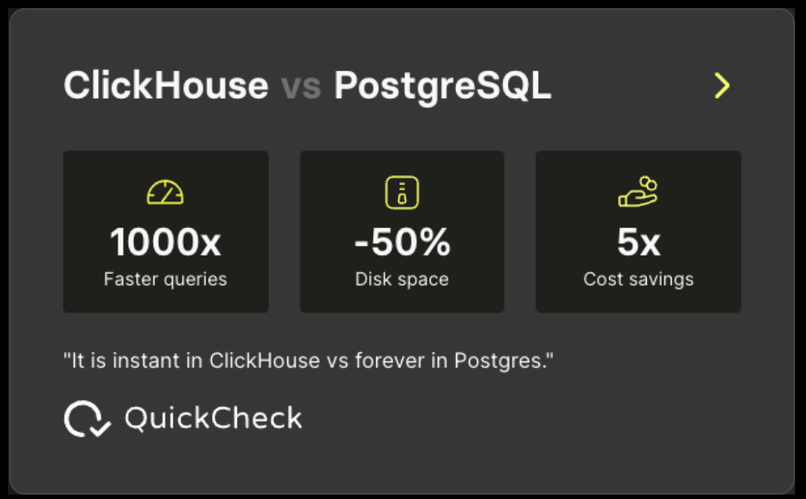
Arise by INFINITAS

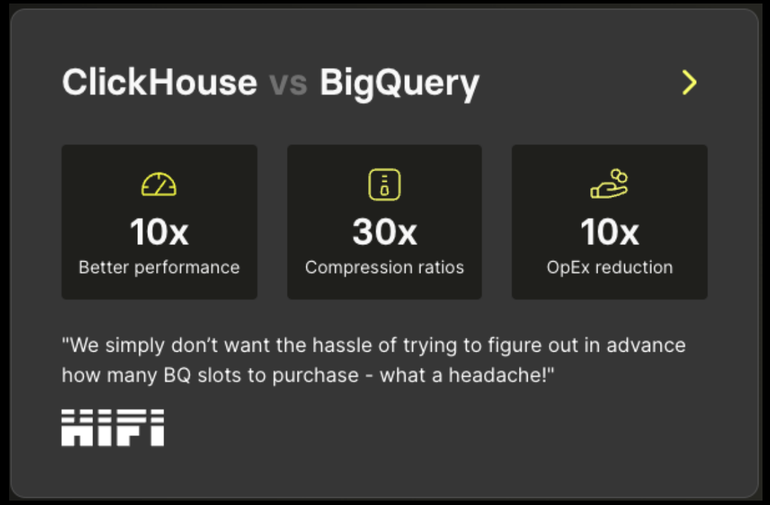
Comparison with other databases

In contrast, traditional warehouses and transactional databases lack the performance and cost efficiency that makes them viable for analytic workloads at scale.









ref : https://clickhouse.com/use-cases

Arise by INFINITAS

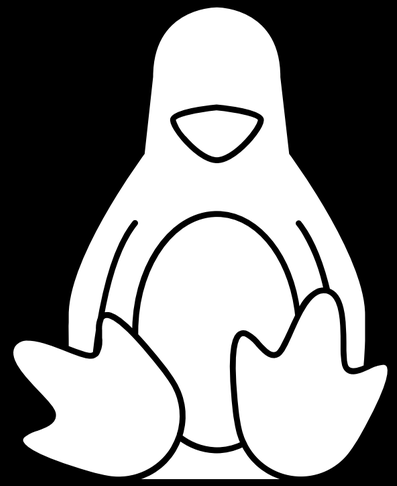
System requirements

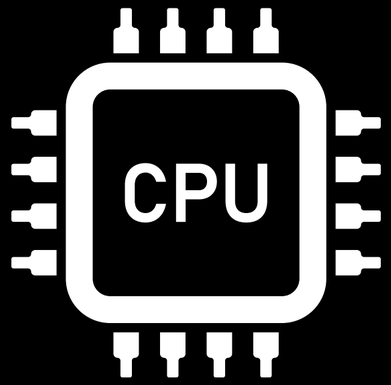
Operating System
- any Linux, FreeBSD, or macOS

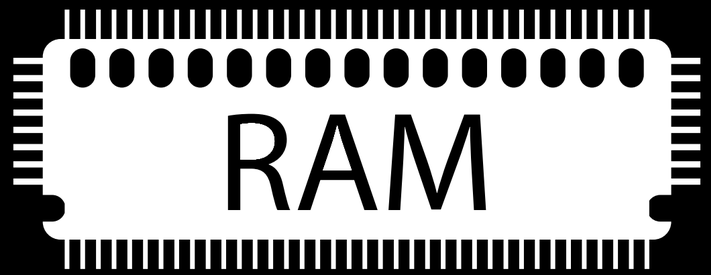
CPU
- x86-64, ARM

Minimum RAM
- 4 GB

File system
- Ex4







Installation methods

Clickhouse Cloud

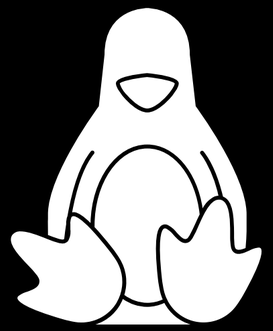
- https://clickhouse.com/cloud

Production Deployments

sudo apt-get install -y clickhouse-server clickhouse-client

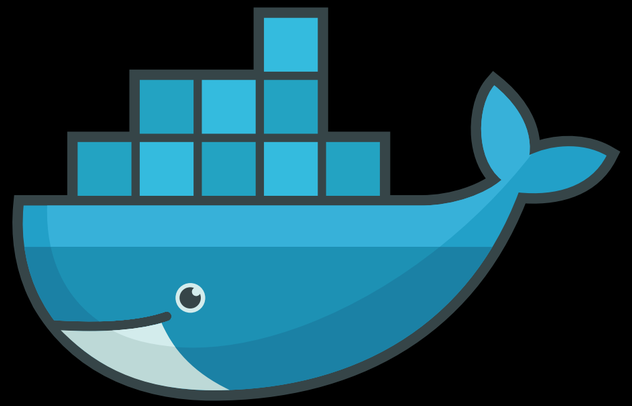
Quick Install

curl https://clickhouse.com/ | sh



Docker Image

docker pull clickhouse/clickhouse-server

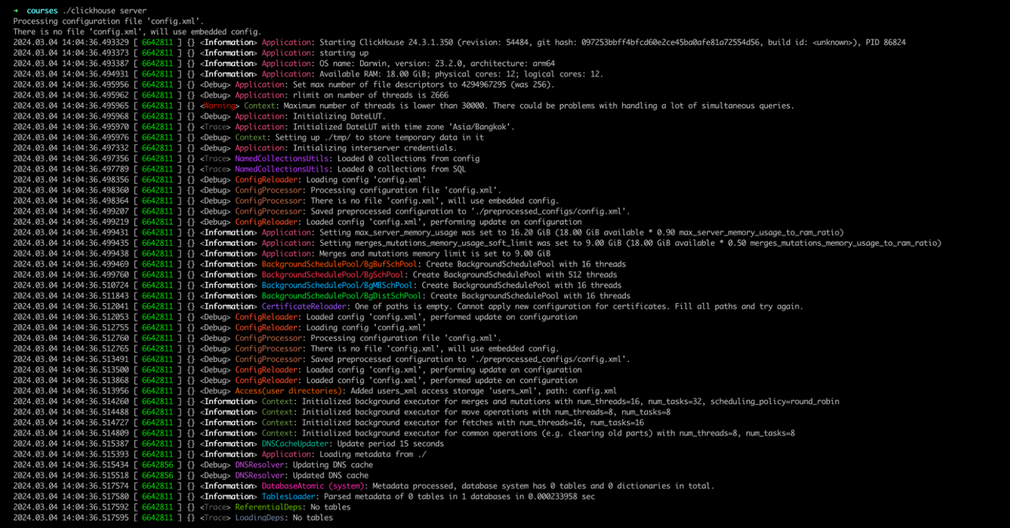


Qucik Install and Start

Quick Install

curl https://clickhouse.com/ | sh

```
➜  courses curl https://clickhouse.com/ | sh
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2822    0  2822    0     0   5301      0 --:--:-- --:--:-- --:--:--  5294

Will download https://builds.clickhouse.com/master/macos-aarch64/clickhouse into clickhouse

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 67.5M  100 67.5M    0     0  17.8M      0  0:00:03  0:00:03 --:--:-- 17.8M

Successfully downloaded the ClickHouse binary, you can run it as:
    ./clickhouse
```



Prepare data folder

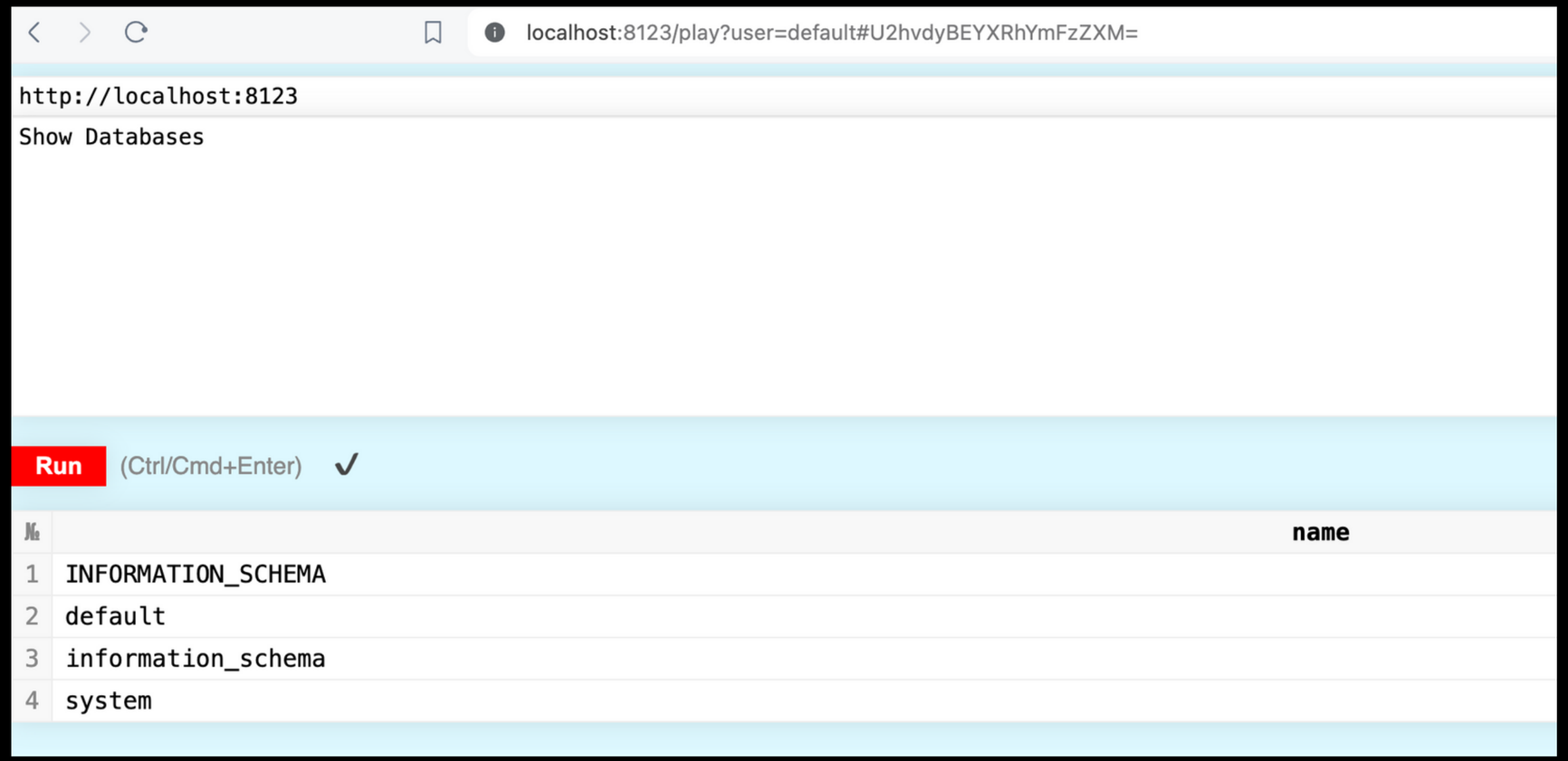
Start up

./clickhouse server

Qucik Install and Start

Test

- open browser
- goto http://localhost:8123/play

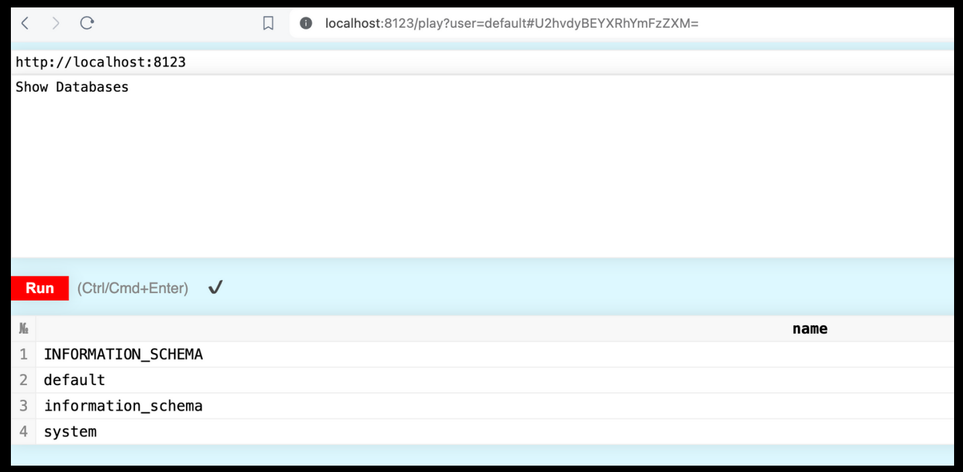


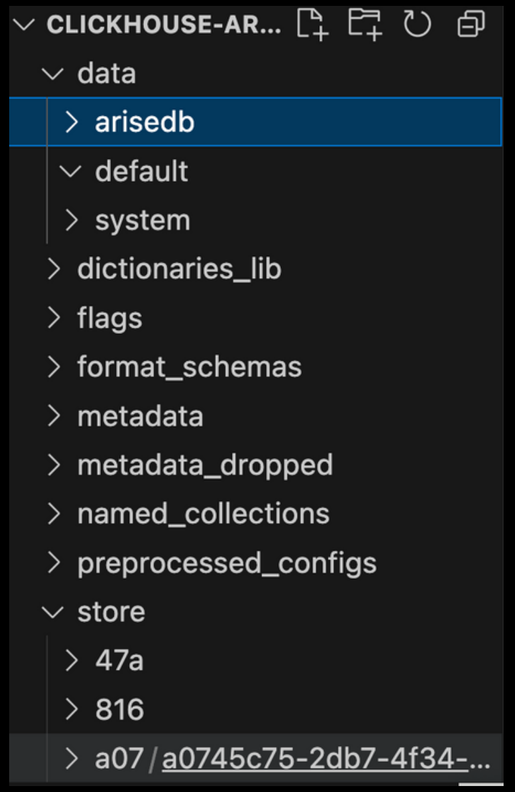
1.3 Basic Concepts

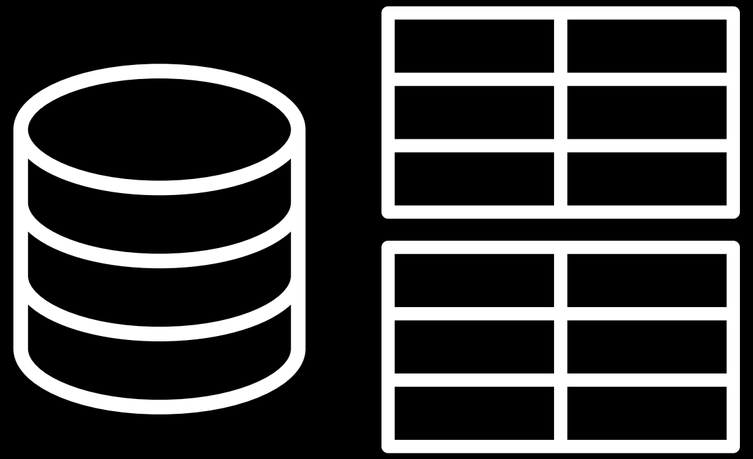
Databases and Tables

Databases
Like most databases, ClickHouse logically groups tables into databases.
Use the CREATE DATABASE command to create a new database in ClickHouse:

```
CREATE DATABASE IF NOT EXISTS arisedb
```



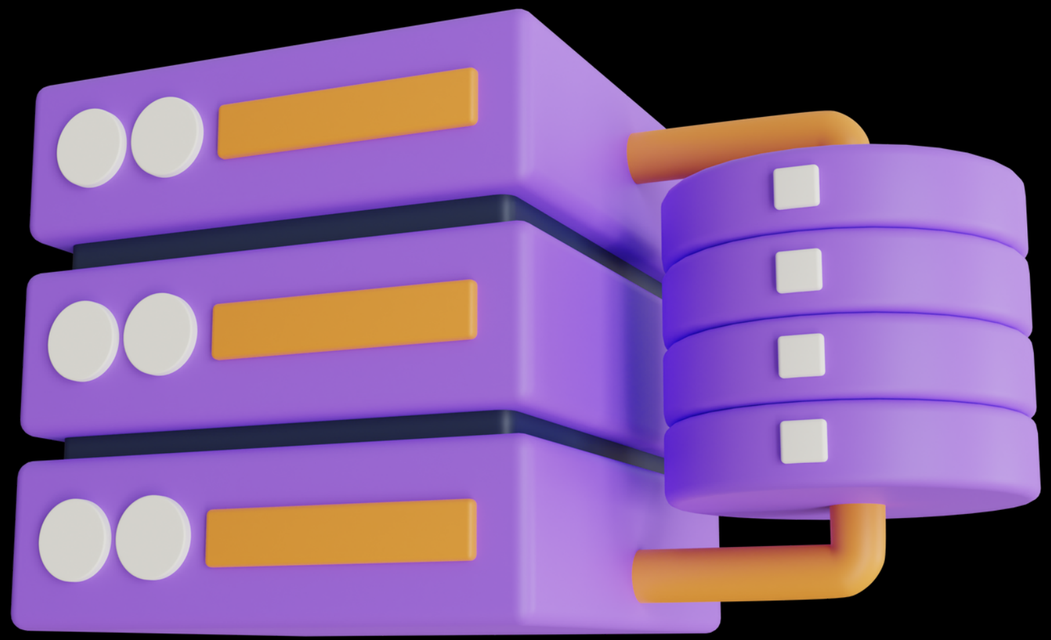




Databases and Tables

Tables

- Tables are structures within a database that store the actual data. They define the schema, which includes the columns, data types, and other properties of the data to be stored.
- ClickHouse supports different types of tables, and the choice of table type depends on the specific use case. Common table types include MergeTree (for time-series data), Distributed (for distributed data storage), and more



- Entities
- Attributes
- Relationships
- Normalization / De-normalization
- Data Types
- Constraints
- Audit Fields

Try it - Database Table

Tables

1. Clone tutorial repository. https://github.com/apaichon/clickhouse-tutorial
2. cd to <project folder> <clickhouse-tutorial/fake-data>
3. npm/yarn/bun install.
4. Create a table for keep people data by sql command.
5. Prepare sampling data by node js.

2

```
clickhouse-tutorial/fake-data
➜  fake-data git:(main) ✗
```

4

```
CREATE TABLE IF NOT EXISTS arisedb.people
(
    id LowCardinality(FixedString(13)),
    first_name String,
    last_name String,
    dateOfBirth Date,
    laser_id String
)
ENGINE = MergeTree
Primary Key (id, laser_id)
ORDER BY (id,laser_id, dateOfBirth);
```

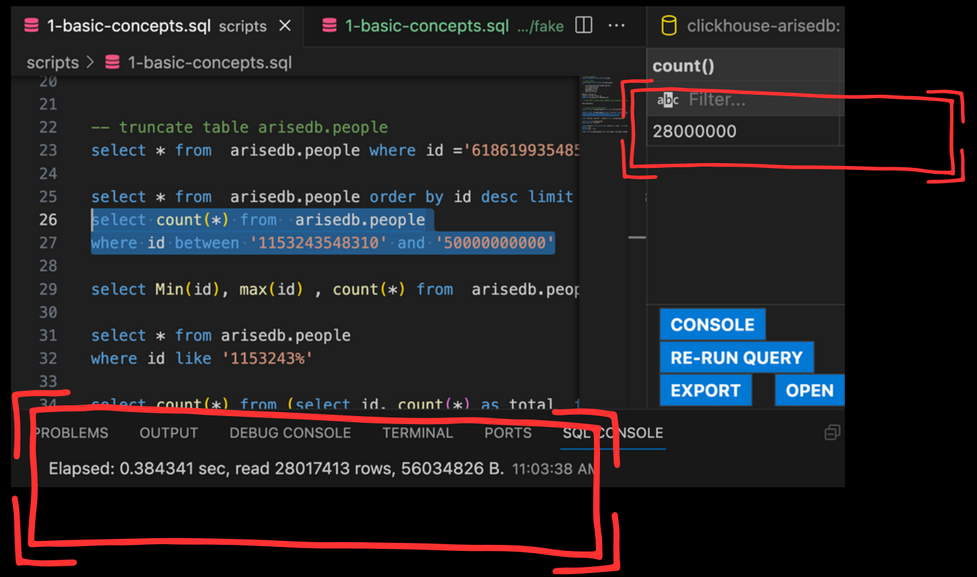
5

What have we found?

Arise by INFINITAS

INSERT 1 M ROWS TO FAST

```
generate data: 3.032s
insert data: 1.004s
Successfully inserted 1000000 people data into ClickHouse table.
```



Why in real world insert slowly ?

Arise by INFINITAS

Transfer
1...N requests and responses

1 Request Response Life cycle

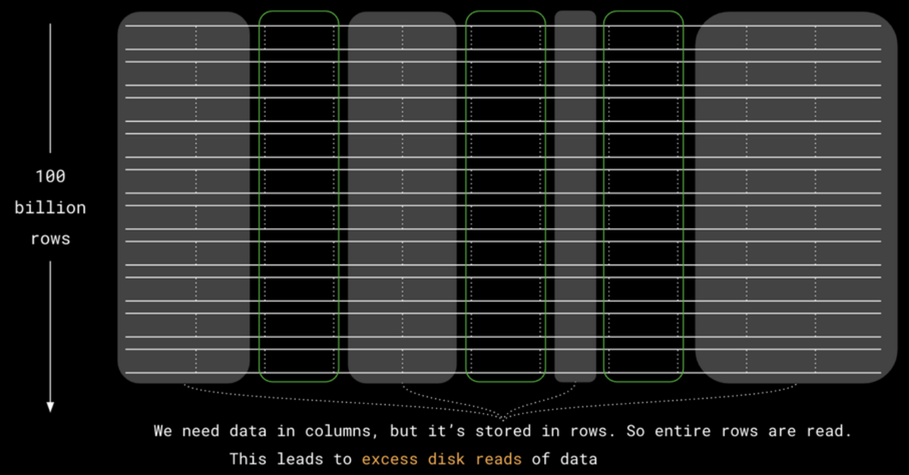
- Application Processing
- Network Latency
- Service Processing
- Database Connection
- Database Processing
- Database Close Connection

NOT INSERT BATCH !!

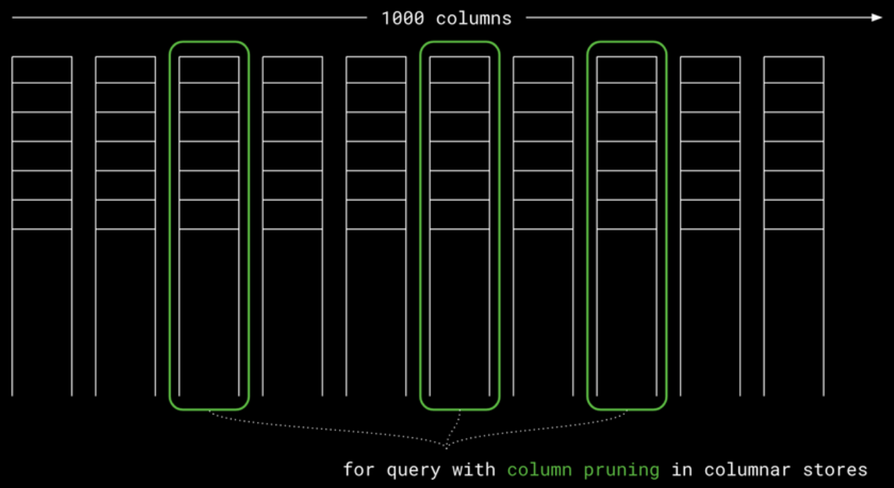
Why clickhouse so fast ?

Arise by INFINITAS

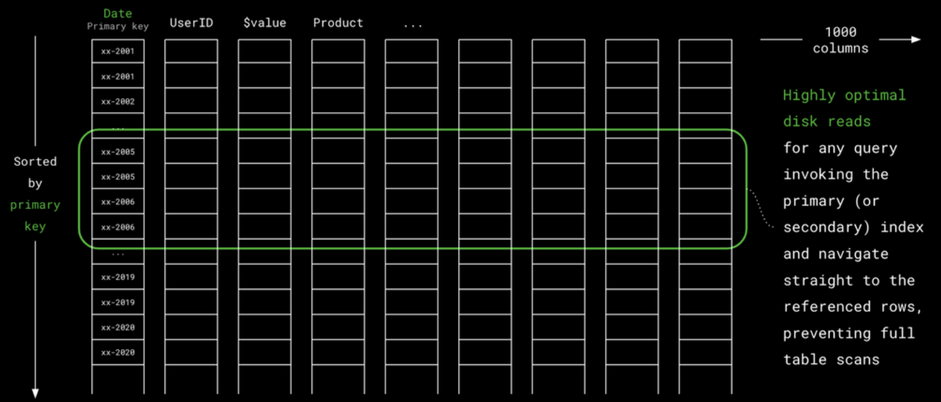
Row-based Store



Columnar Store



Primary index in ClickHouse



ref : https://youtu.be/CAS2otEoerM , https://chistadata.com/why-clickhouse-is-so-fast/

Data Types

Arise by INFINITAS

Standard data types in a database refer to the predefined categories of data that can be stored in tables.

Boolean

Numeric

- Int
  - Int8-256
  - UInt8-256
- Float
  - 32-64
- Decimal
  - 32-256

TEXT

- String
- FixedString(N)

DateTime

- Date
- DateTime
- DateTime64

Other

- UUID
- Enum
- LowCardinality
- Array
- AggregateFunction
- JSON
- Tuple
- Nullable
- Variant
- Domains
- Nested
- IPv4, IPv6
- Geo
- Map(key,value)
- SimpleAggregateFunction

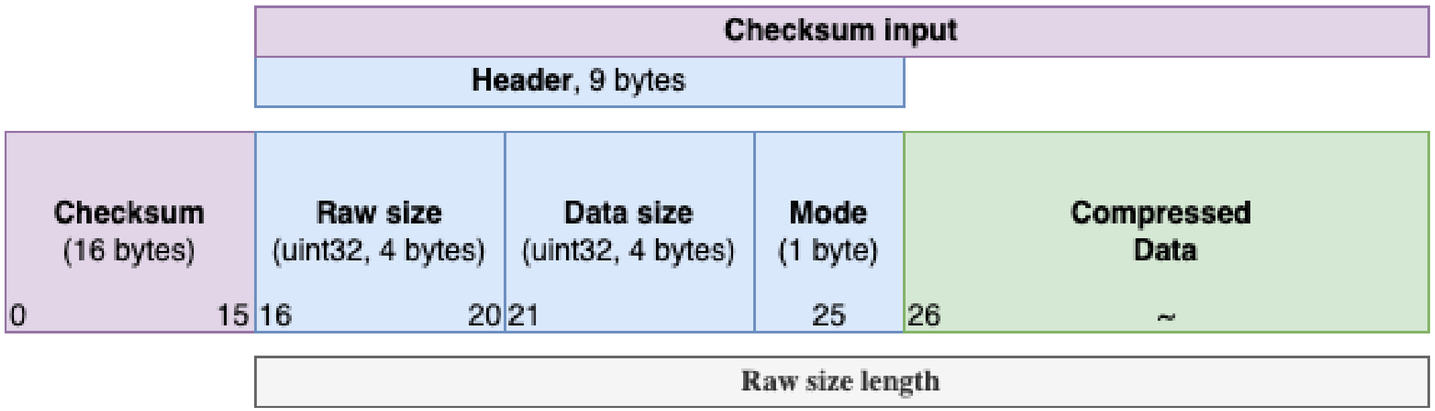
Compression

Arise by INFINITAS

ClickHouse protocol supports data blocks compression with checksums. Use LZ4 if not sure what mode to pick.

value	name	description
`0x02`	None	No compression, only checksums
`0x82`	LZ4	Extremely fast, good compression
`0x90`	ZSTD	Zstandard, pretty fast, best compression

name	ratio	encoding	decoding
zstd 1.4.5 -1	2.8	500 MB/s	1660 MB/s
lz4 1.9.2	2.1	740 MB/s	4530 MB/s

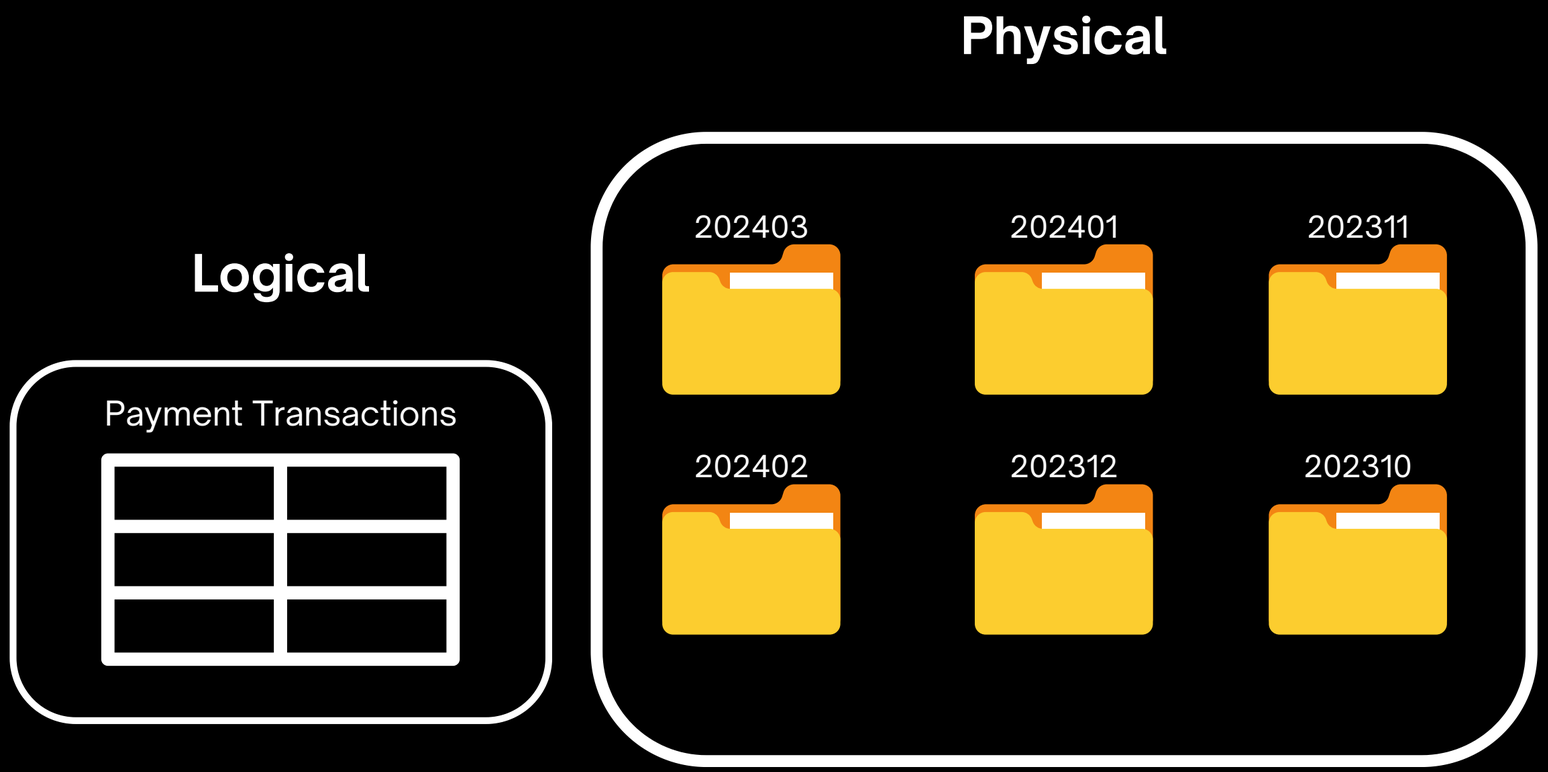


field	type	description
checksum	uint128	Hash of (header + compressed data)
raw_size	uint32	Raw size without header
data_size	uint32	Uncompressed data size
mode	byte	Compression mode
compressed_data	binary	Block of compressed data

Partition

A partition is a logical combination of records in a table by a specified criterion. You can set a partition by an arbitrary criterion, such as by month, by day, or by event type. Each partition is **stored separately** to simplify manipulations of this data. When accessing the data, ClickHouse uses the **smallest subset of partitions possible**. Partitions **improve performance for queries containing a partitioning key** because ClickHouse will filter for that partition before selecting the parts and granules within the partition.

```
Create table if NOT EXISTS payment
(
    payment_id UInt64,
    amount Float64,
    payment_date DateTime,
    customer_id UInt32
)
ENGINE = MergeTree
Partition By toYYYYMM(payment_date)
order by (payment_date, payment_id, customer_id )
```

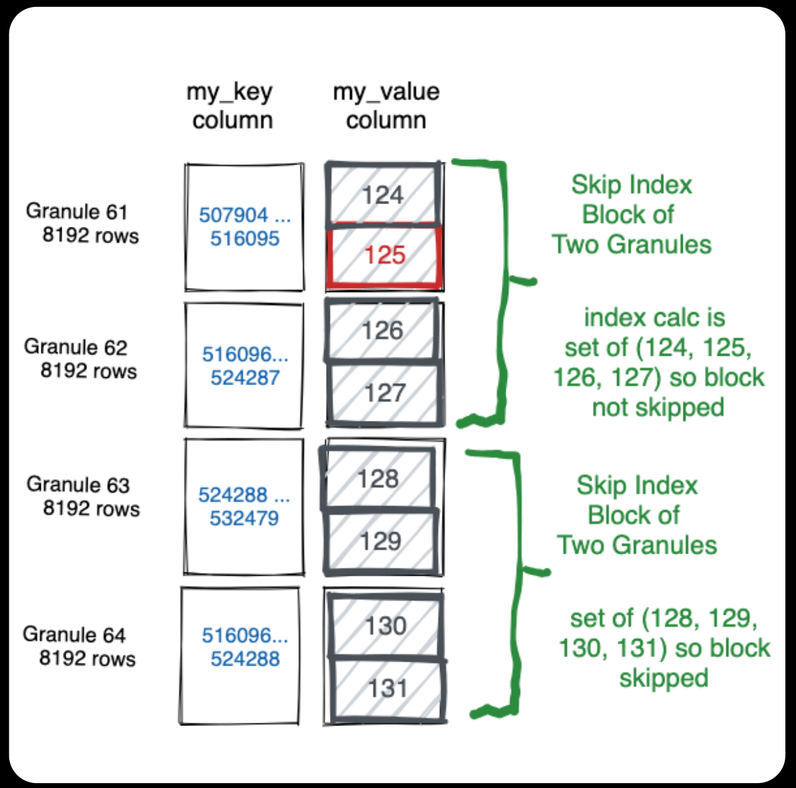


Sorting

Arise by INFINITAS

Sorting in ClickHouse refers to the order in which data is physically stored within each partition of a table. The sorting key determines the order of the data, and it is used to organize the data for efficient retrieval.

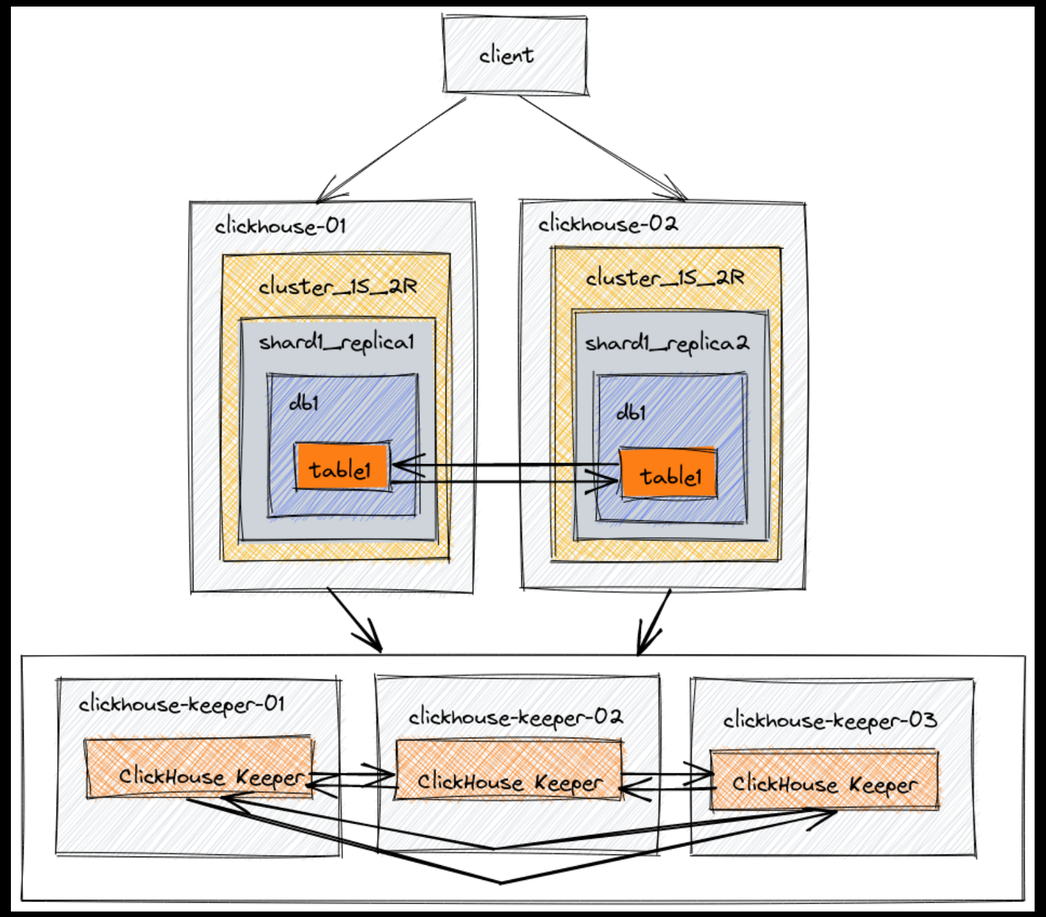
```
Create table if NOT EXISTS payment
(
    payment_id UInt64,
    amount Float64,
    payment_date DateTime,
    customer_id UInt32
)
ENGINE = MergeTree
Partition By toYYYYMM(payment_date)
order by (payment_date, payment_id, customer_id )
```



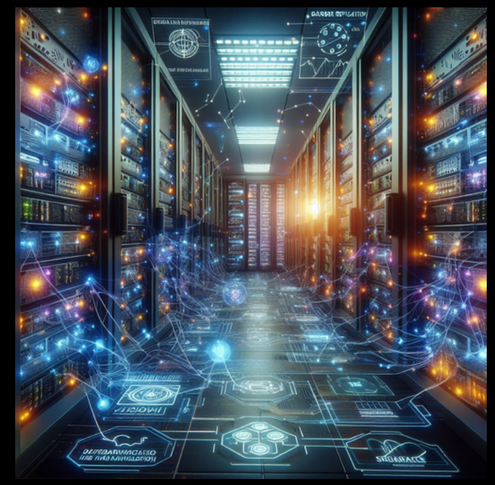
Physical

Replication

In ClickHouse, replication is a mechanism for creating and maintaining **copies of data on multiple servers to ensure data availability**, fault tolerance, and load distribution. Replication is crucial for high-availability configurations and for protecting against data loss in the event of server failures. ClickHouse supports both synchronous and asynchronous replication modes.



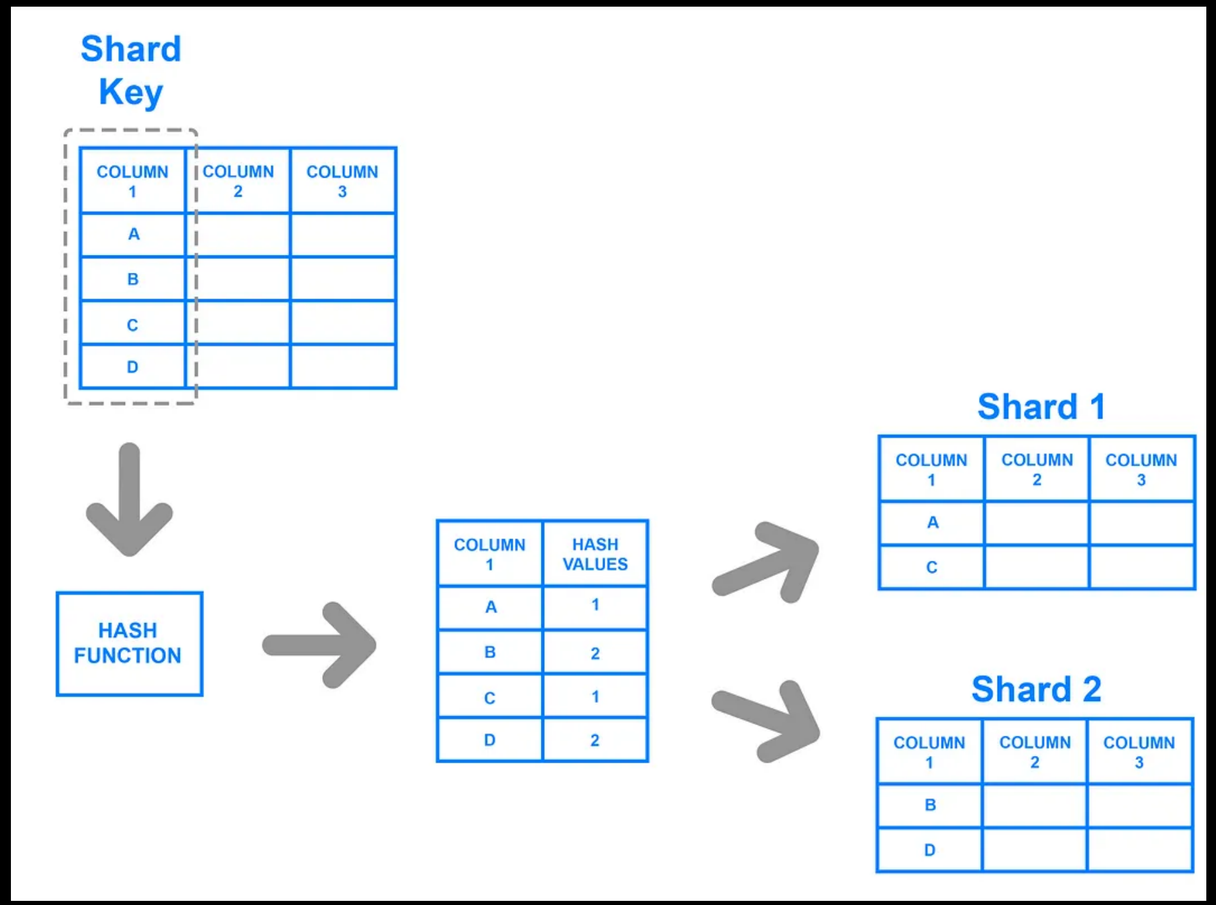
Node	Description
clickhouse-01	Data
clickhouse-02	Data
clickhouse-keeper-01	Distributed coordination
clickhouse-keeper-02	Distributed coordination
clickhouse-keeper-03	Distributed coordination

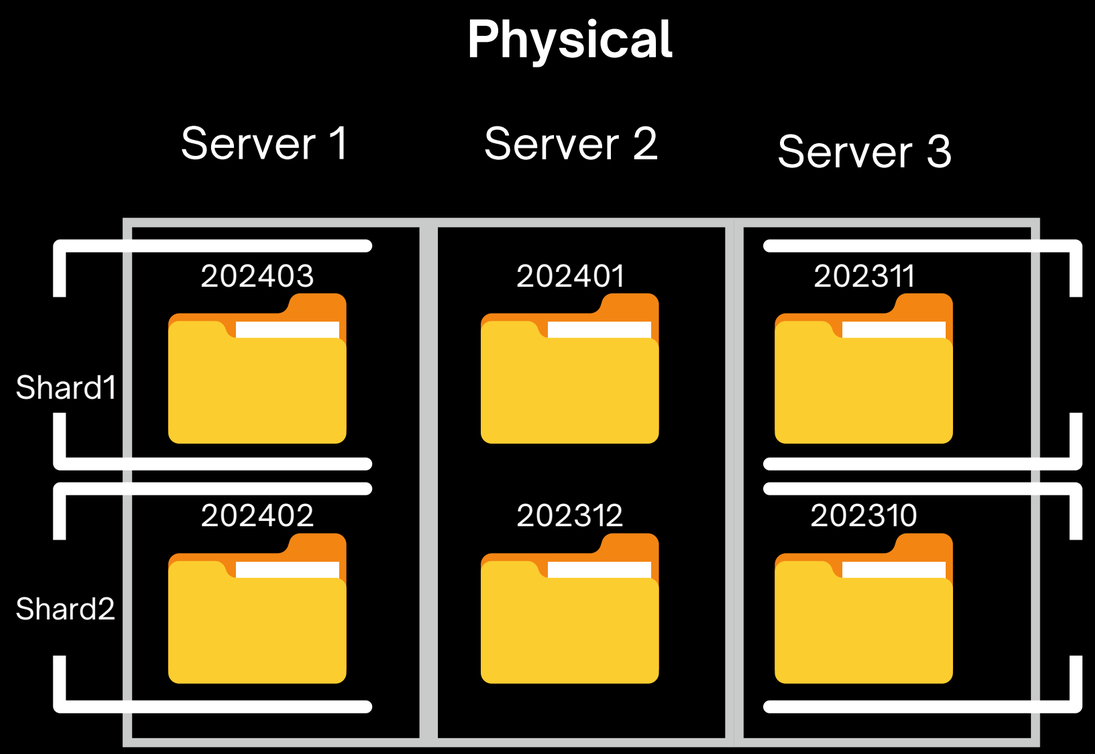


Sharding

Arise by INFINITAS

Sharding is a technique used in distributed database systems, including ClickHouse, to horizontally partition data across multiple servers or nodes. Each shard contains a subset of the data, and the distribution of data is designed to improve performance, scalability, and parallel processing. In ClickHouse, sharding is often used in conjunction with replication for fault tolerance and high availability.





Arise by INFINITAS

Tips - Why favorite use snake case naming convention ?

Because Snake case with lower case support for almost database providers.

Naming	ORACLE	PostgreSQL	MySQL	Microsoft SQL Server	MongoDB
	CASE SENSITIVE	CASE SENSITIVE ?	CASE SENSITIVE ?	CASE Insensitive	CASE Sensitive
CUSTOMERS	CUSTOMERS	“CUSTOMERS”	`CUSTOMERS`	CUSTOMERS	CUSTOMERS
ORDER_ITEMS	ORDER_ITEMS	“ORDER_ITEMS”	`ORDER_ITEMS`	ORDER_ITEMS	ORDER_ITEMS

```
select *
from
Customers;
```

Summary

OVERVIEW

- ClickHouse is an open-source columnar database management system (DBMS)
- Designed for high-performance analytics processing.
- Used in scenarios such as data warehousing, business intelligence, log analytics, and real-time analytics.

INSTALLATION AND SETUP

- Package installation
- Docker
- Cloud
- Linux Production Systemctl

BASIC CONCEPTS

- Tables and databases are fundamental
- Columnar storage, which is efficient for analytical queries involving large datasets.
- Partitions and sorting, data types, compression, replication, and sharding.

KEY TAKEAWAYS

- Columnar database
- System requirement
- Installation
- Basic concepts like tables, databases, partitions, and sorting.

ClickHouse

Arise by INFINITAS

Module: 2

ClickHouse Query Language (CHQL)

MODULE 2 : CLICKHOUSE QUERY LANGUAGE (CHQL)

2.1 Basic Queries

- Insert statement
- SELECT statement
- Updating and Deleting ClickHouse Data
- WHERE clause
- ORDER BY and LIMIT

2.2 Advanced Queries

- JOIN operations
- Subqueries
- Aggregation functions

2.3 Performance Optimization

- Indexing
- Query profiling
- Using the EXPLAIN statement

Insert statement

Inserts data into a table.

```
INSERT INTO [TABLE] [db.]table
[(c1, c2, c3)]
[SETTINGS ...]
VALUES (v11, v12, v13), (v21, v22, v23), ...
```

- **INSERT INTO** - reserved word to insert.
- **TABLE -** table name to insert. If not specify prefix will refer to using database.
- **[db.]table -** prefix of database name with table name.
- **VALUES -** reserved word to separate values.
- **(v1, v2, v3), (vn, vn, vn)** ... values in bracket to insert

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Kaushik', 23, 'Kota', 2000.00 );
```

2.1 Basic Queries

Try It!

- create table click_events

```
CREATE TABLE click_events
(
    `dt` DateTime,
    `event` String,
    `status` Enum8('success' = 1, 'fail' = 2)
)
ENGINE = MergeTree
ORDER BY dt
```

- insert data to click_events table

```
INSERT INTO click_events
SELECT (parseDateTimeBestEffortOrNull('12:00') -
toIntervalHour(randNormal(0, 3))) -
toIntervalDay(number % 30),
'Click',
['fail', 'success'][randBernoulli(0.9) + 1]
FROM numbers(10000000)
```

- select data of click_events

```
SELECT
    dt,
    count(*) AS c,
    bar(c, 0, 100000)
FROM click_events
GROUP BY dt
ORDER BY dt ASC
```

dt	c	bar(count(), 0, ...
Filter...	Filter...	Filter...
2023-12-02 23:00:00	2	
2023-12-03 00:00:00	10	
2023-12-03 01:00:00	33	
2023-12-03 02:00:00	108	
2023-12-03 03:00:00	308	▎
2023-12-03 04:00:00	767	▊
2023-12-03 05:00:00	2027	▋
2023-12-03 06:00:00	4393	▍
2023-12-03 07:00:00	8305	▉
2023-12-03 08:00:00	14514	▌...
2023-12-03 09:00:00	22670	▋...
2023-12-03 10:00:00	31175	▏...
2023-12-03 11:00:00	38961	▉...
2023-12-03 12:00:00	86982	▋...
2023-12-03 13:00:00	39336	▎...
2023-12-03 14:00:00	31160	▏...
2023-12-03 15:00:00	22298	▋...
2023-12-03 16:00:00	14582	▌...
2023-12-03 17:00:00	8190	▊
2023-12-03 18:00:00	4190	▍

2.1 Basic Queries

SELECT statement

The basic structure and usage of the SELECT statement in ClickHouse.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
ORDER BY column1, ...
LIMIT n
OFFSET n
or ROW FETCH FIRST ROW n ROWS ONLY
```

- The SELECT statement is fundamental for querying data in ClickHouse.
- WHERE clause is used for filtering rows based on specified conditions.
- ORDER BY clause helps in sorting query results.
- LIMIT clause is useful for restricting the number of rows returned.
- OFFSET and FETCH allow you to retrieve data by portions. They specify a row block which you want to get by a single query.

Try it !

Arise by INFINITAS

Prepare table on Clickhouse

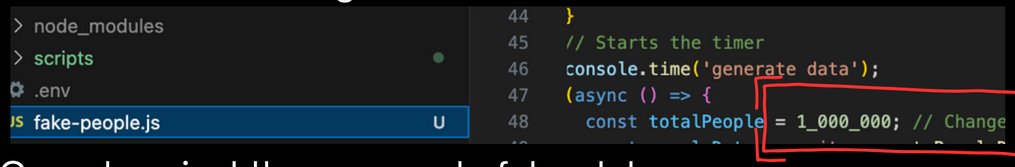
```
CREATE TABLE IF NOT EXISTS people
(
    id LowCardinality(FixedString(13)),
    first_name String,
    last_name String,
    dateOfBirth Date,
    laser_id String
)
ENGINE = MergeTree
Primary Key (id, laser_id)
ORDER BY (id,laser_id, dateOfBirth);
```

Go to fake-data folder

- set .env
- set number data to generate

clickhouse-tutorial > fake-data > .env

```
DB_HOST=localhost
DB_PORT=8123
DB_USER=default
DB_PASSWORD=
DB_NAME=default
```



- Open terminal then run node fake-data

- run node fake-data

```
→  fake-data git:(main) ✗ node fake-people
generate data: 3.604s
insert data: 1.014s
Successfully inserted 1000000 people data into ClickHouse table.
```

Basic SELECT:

- Retrieve all columns for all records in the `**people**` table with limit.

```
SELECT *
FROM people limit 10;
```

Filtering with WHERE Clause:

- Retrieve records for people born after a certain date.

```
SELECT id, first_name, last_name, dateOfBirth
FROM people
WHERE dateOfBirth > '2000-01-01' limit 10;
```

Sorting with ORDER BY:

- Retrieve records sorted by last_name in descending order.

```
SELECT id, first_name, last_name, dateOfBirth
FROM people
ORDER BY last_name DESC limit 10;
```

Updating and Deleting ClickHouse Data

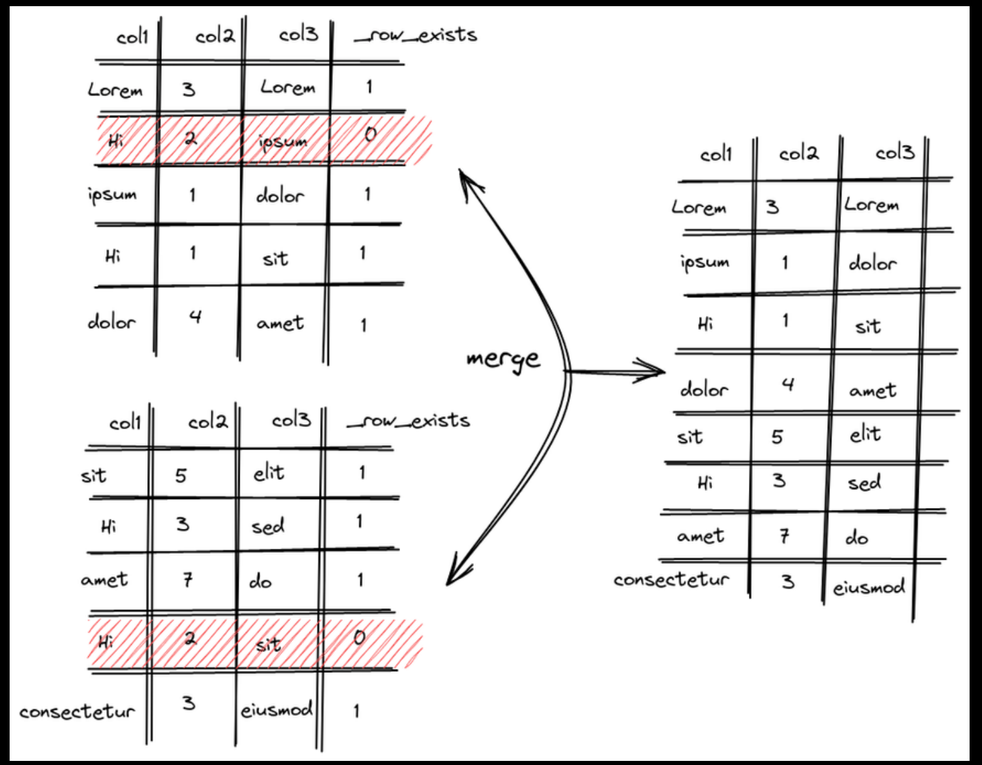
ClickHouse inserts and queries data quickly, but updates and deletes become slow when dealing with large datasets because it needs to restructure data blocks.

```
Alter table people
Update last_name = 'Zulauf-Wiza'
Where id ='3158190887792'
```

```
Delete from people Where id ='3158190887792'
```

```
DELETE FROM [db.]table [ON CLUSTER cluster] [WHERE expr]
```

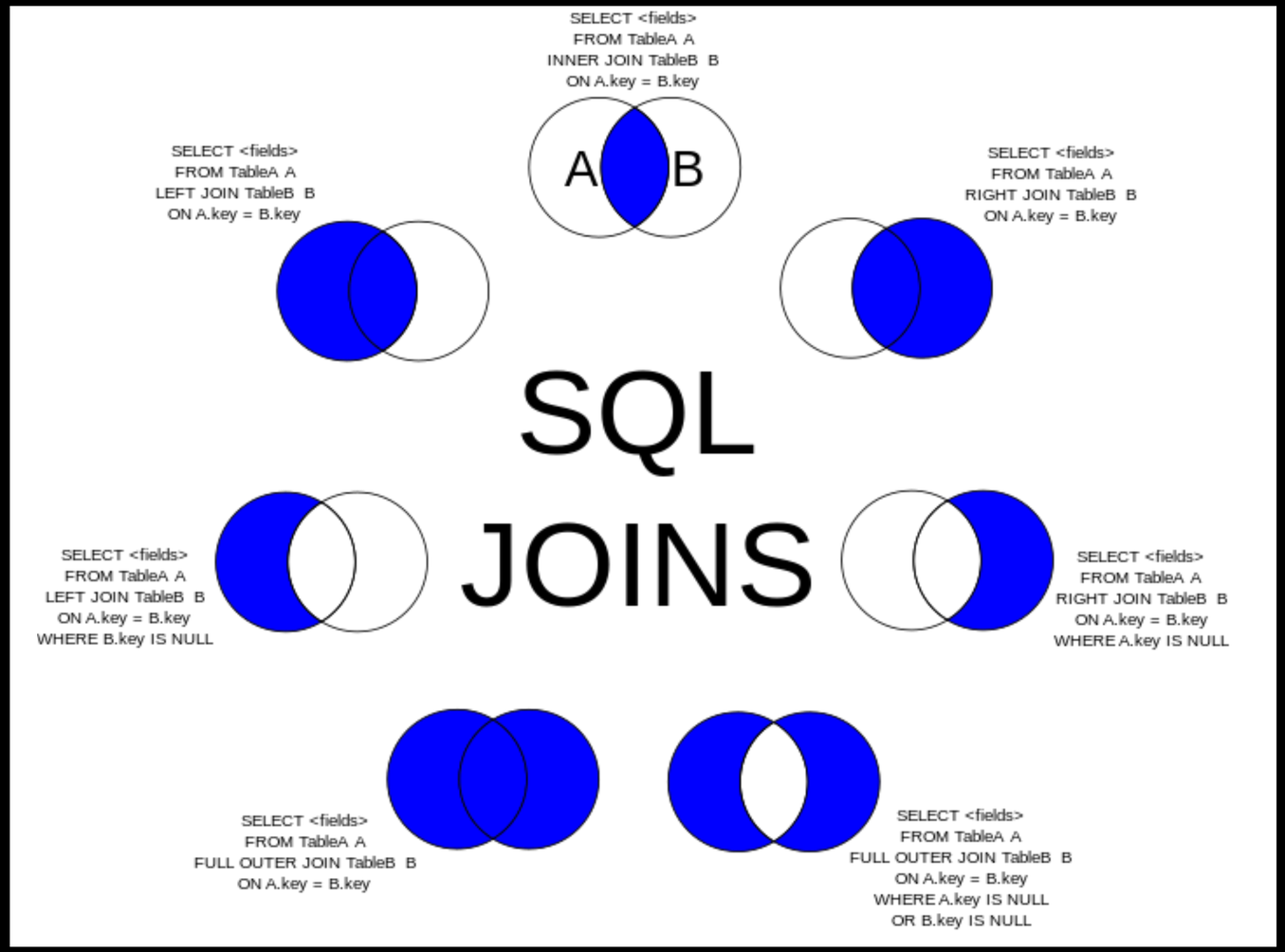
1	en	3452180
~~2~~	~~es~~	~~4491590~~
~~3~~	~~de~~	~~4490097~~
4	fr	3390573
5	it	2015989
6	ja	1379148
7	pt	1259443
~~8~~	~~tr~~	~~1254182~~
9	zh	988780



Clickhouse come with Immutable Concept.

JOIN statement

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.



2.2 Advanced Queries

Try It! - JOIN statement

Prepare table and data.

```
CREATE TABLE IF NOT EXISTS customers (
    customer_id UInt32,
    customer_name String,
    email String,
    date_of_birth Date
) ENGINE = MergeTree() PRIMARY KEY (customer_id);

INSERT INTO customers (customer_id, customer_name, email,date_of_birth)
VALUES (1, 'John Doe', 'john@example.com', '1990-01-01'),
    (2, 'Jane Smith', 'jane@example.com', '2001-05-09'),
    (3, 'Bob Johnson', 'bob@example.com', '1979-11-13');
```

```
CREATE TABLE IF NOT EXISTS products
(
    product_id UInt32,
    product_name String,
    price Float64

)
ENGINE = MergeTree()
PRIMARY KEY (product_id, product_name);

INSERT INTO products (product_id, product_name, price)
VALUES
    (501, 'Laptop', 800.00),
    (502, 'Smartphone', 500.00),
    (503, 'Headphones', 80.00);
```

```
CREATE TABLE IF NOT EXISTS orders
(
    order_id UInt32,
    customer_id UInt32,
    order_date DateTime,
    total_amount Float64
)
ENGINE = MergeTree()
PRIMARY KEY (order_id, customer_id)
Order by (order_id, customer_id, order_date);

INSERT INTO orders (order_id, customer_id, order_date, total_amount)
VALUES
    (101, 1, '2023-02-15', 150.50),
    (102, 2, '2023-02-16', 220.75),
    (103, 3, '2023-02-17', 120.00);
```

```
CREATE TABLE IF NOT EXISTS order_items
(
    order_item_id UInt32,
    order_id UInt32,
    product_id UInt32,
    quantity UInt32,
    subtotal Float64

)
ENGINE = MergeTree()
PRIMARY KEY (order_item_id, order_id)
Order by (order_item_id, order_id, product_id);

INSERT INTO order_items (order_item_id, order_id, product_id, quantity, subtotal)
VALUES
    (1001, 101, 501, 2, 1600.00),
    (1002, 101, 502, 1, 500.00),
    (1003, 102, 503, 3, 240.00);
```

```
SELECT
    c.customer_id,
    c.customer_name,
    o.order_id,
    o.order_date,
    p.product_name,
    oi.quantity,
    oi.subtotal
FROM
    customers AS c
INNER JOIN
    orders AS o ON c.customer_id = o.customer_id
INNER JOIN
    order_items AS oi ON o.order_id = oi.order_id
INNER JOIN
    products AS p ON oi.product_id = p.product_id;
```

c.customer_id	c.customer_na...	o.order_id	o.order_date	p.product_name	oi.quantity	oi.subtotal
abc Filter...	abc Filter...	abc Filter...	abc Filter...	abc Filter...	abc Filter...	abc Filter...
1	John Doe	101	2023-02-15 00:00:00	Laptop	2	1600
1	John Doe	101	2023-02-15 00:00:00	Smartphone	1	500
2	Jane Smith	102	2023-02-16 00:00:00	Headphones	3	240

Subqueries

Sub Queries is writing Sql nested Sql, in which each part of sql will have separate results. The data storage structure may not be able to produce the desired results in every case, so a complex sequence must be written to get the desired results. which will be related to the main query.

```sql
SELECT
    customer_id,
    customer_name,
    (
      SELECT count()
      FROM orders
      WHERE orders.customer_id = customer_id
    ) AS total_orders
FROM customers
```

```sql
SELECT
    c.customer_name,
    o.order_date,
    o.total_amount
FROM
    orders o
    left join customers c
    on o.customer_id = c.customer_id
WHERE
    total_amount > (SELECT AVG(total_amount) FROM orders);
```

Caution: Becareful to use subqueries about performance aviod to used it by joining.

Aggregation functions

Aggregate is the combination of multiple pieces of data to be grouped into the same category to calculate to find a new result, such as Count, Sum, Min, Max, Medium, Avg. Therefore, the Aggregate functions command is often **used together with Group by.**

```
select
    count(*) as totalItems,
    min(total_amount) as minAmount,
    avg(total_amount) as avgAmount,
    max(total_amount) as maxAmount,
    sum(total_amount) as  totalAmount
    from orders
```

```
SELECT
    p.product_id,
    p.product_name,
    sum(oi.quantity) AS total_quantity_sold
FROM products AS p
LEFT JOIN order_items AS oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.product_name
ORDER BY total_quantity_sold DESC
LIMIT 10;
```

```
SELECT
    p.product_id,
    p.product_name,
    sum(oi.quantity) AS total_quantity_sold
FROM products AS p
LEFT JOIN order_items AS oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.product_name
ORDER BY total_quantity_sold ASC
LIMIT 10;
```

- count
- min
- max
- sum
- avg
- any
- stddevPop
- stddevSamp
- varPop
- varSamp
- corr
- covarPop
- covarSamp
- entropy
- exponentialMovingAverage
- intervalLengthSum
- kolmogorovSmirnovTest
- mannwhitneyutest
- median
- rankCorr
- sumKahan
- studentTTest
- welchTTest

ref : https://youtu.be/CAS2otEoerM , https://chistadata.com/why-clickhouse-is-so-fast/

2.2 Advanced Queries

WITH Clause

Arise by INFINITAS

ClickHouse supports Common Table Expressions (CTE) and substitutes the code defined in the WITH clause in all places of use for the rest of SELECT query. Named subqueries can be included to the current and child query context in places where table objects are allowed. Recursion is prevented by hiding the current level CTEs from the WITH expression.

```
WITH
    -- Best Selling Products
    best_selling_products AS (
        SELECT
            p.product_id,
            p.product_name,
            sum(oi.quantity) AS total_quantity_sold
        FROM products AS p
        LEFT JOIN order_items AS oi ON p.product_id = oi.product_id
        GROUP BY p.product_id, p.product_name
        ORDER BY total_quantity_sold DESC
        LIMIT 1
    ),
    -- Worst Performing Products (Least Sold)
    worst_performing_products AS (
        SELECT
            p.product_id,
            p.product_name,
            sum(oi.quantity) AS total_quantity_sold
        FROM products AS p
        LEFT JOIN order_items AS oi ON p.product_id = oi.product_id
        GROUP BY p.product_id, p.product_name
        ORDER BY total_quantity_sold ASC
        LIMIT 1
    ),
```

```
    avg_age_best_customer AS (
        select avg(age) as avg_age from (
        select  c.customer_id,  oi.subtotal , toYear(now()) - toYear(c.date_of_birth) as age from
customers c
        inner join orders o on c.customer_id = o.customer_id
        inner join order_items oi on o.order_id = oi.order_id
        order by subtotal desc
        limit 10)a
    ),
    -- Best Hour Time to Sales
    best_hour_sales AS (
        SELECT
            toHour(o.order_date) AS hour,
            count() AS total_orders
        FROM orders AS o
        GROUP BY hour
        ORDER BY total_orders DESC
        LIMIT 1
    )
-- Combine the results

    select
        b.product_name as best_seller_product,
        w.product_name as worst_performing_product ,
        age_c.avg_age as best_customer_avg_age,
        bhs.hour as best_hour_sales
    from best_selling_products b,
        worst_performing_products w,
        avg_age_best_customer age_c,
        best_hour_sales bhs
```

Indexing

Indexing is a database optimization technique used to **improve the speed and efficiency of queries** by creating a data structure that allows for faster retrieval of records.

Sparse Index

- The default index type in ClickHouse, suitable for time-series data.
- Organizes data in parts and merges them periodically.
- Efficient for range-based queries.

Data Skipping Indexes

- Useful for filtering data based on a condition.
- Stores information about minimum and maximum values for each mark.
- Improves performance for queries with conditions.

Approximate Nearest Neighbor Search Indexes [experimental]

- For Geolocations
- Finding the M closest points for a given point in an N-dimensional vector space.
- The distance between all points in the vector space and the reference point.

Inverted Index [experimental]

- Fast text search capabilities for String or FixedString columns.
- Tokenized cells of the string column.

B-Tree Index (not in Clickhouse)

In traditional relational database management systems, the primary index would contain one entry per table row.
Searching an entry in a **B(+)-Tree** data structure has an average time **complexity of O(log n).**
If data table row is 8.2 million rows.

The formula for k is $k = \log_2(n)$, where $\log_2$ denotes the logarithm base 2.

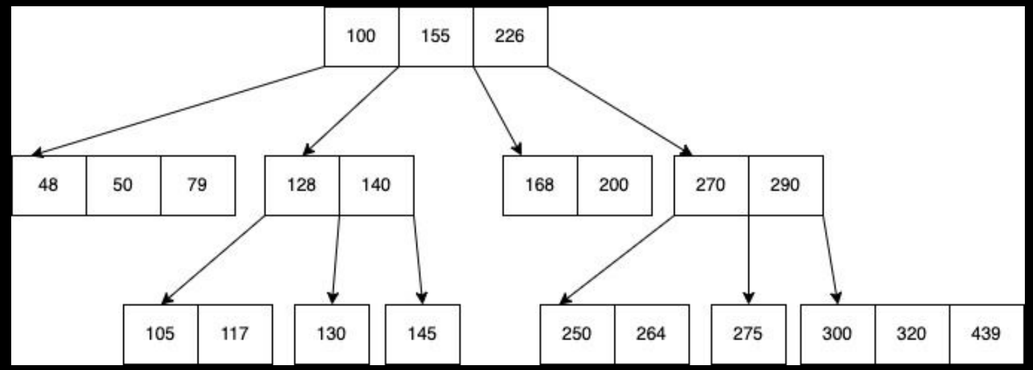
$$k = \log_2(8.2 \times 10^6)$$

Let's calculate this:

$$k \approx \log_2(8.2) + \log_2(10^6)$$

$$k \approx 3.00739 + 20$$

$$k \approx 23.00739$$

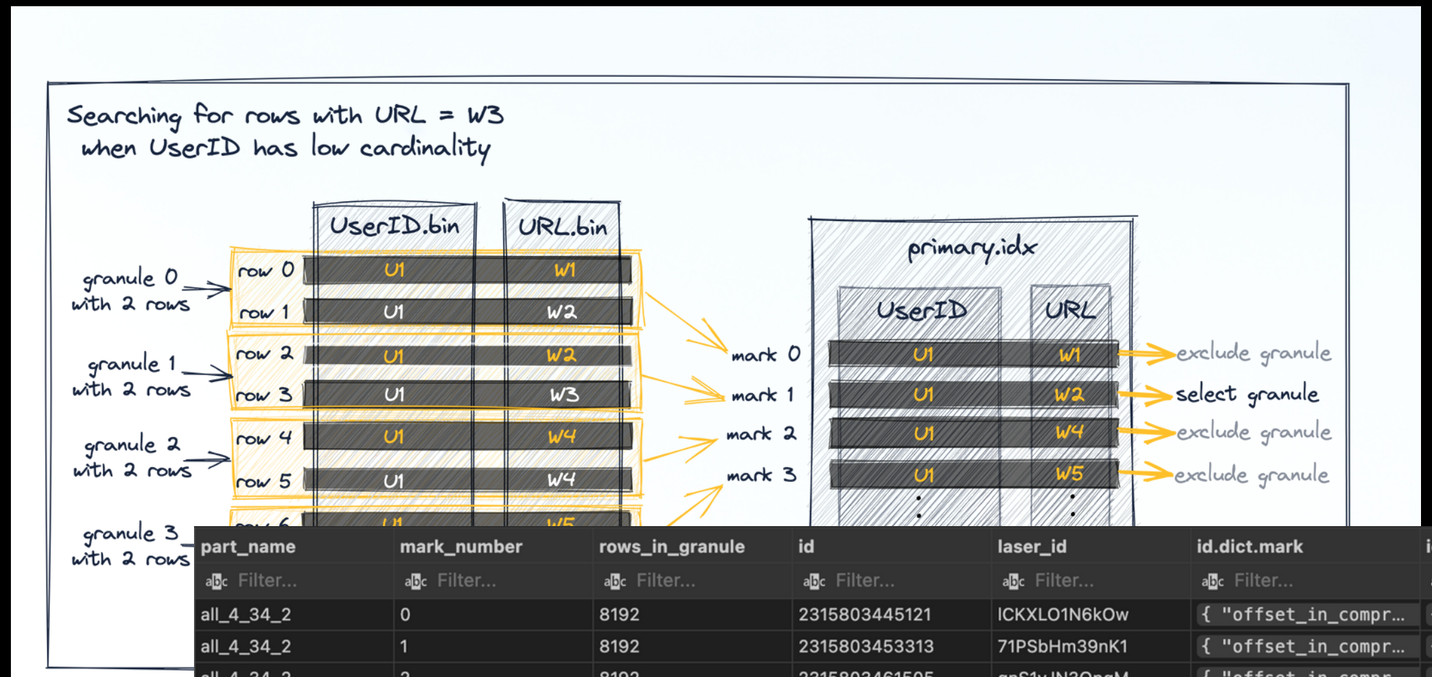


Approximately 23 steps.!

2.3 Performance Optimization

Sparse Index

Sparse indexing is storing the rows for a part on disk ordered by the primary key column(s).



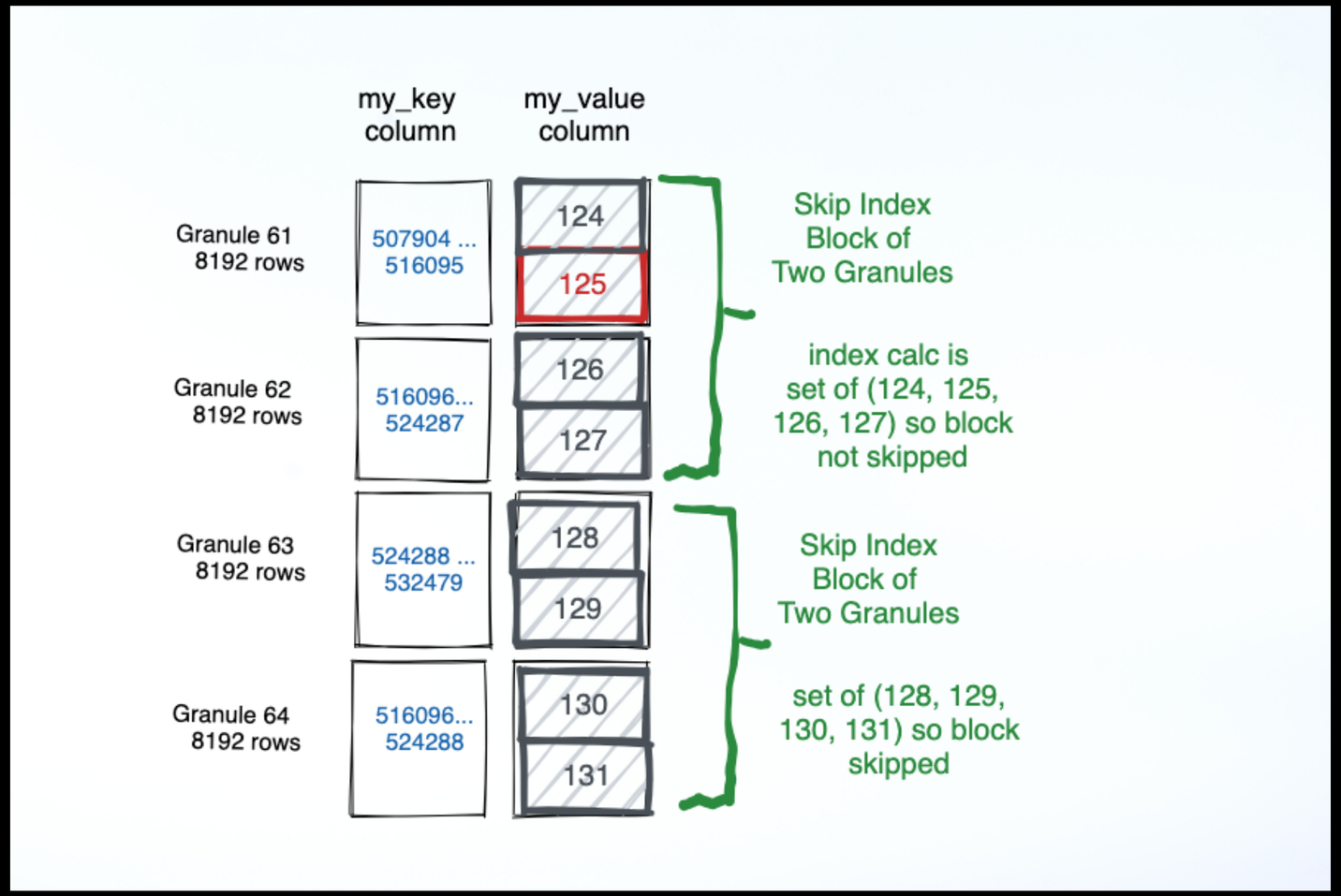
- Primary Keys allow 2 columns
- Chunk *n* rows to granule
- Keep Min Value in granule
- Append value to granule in parallel process

```
SELECT * FROM mergeTreeIndex(currentDatabase(), people, with_marks = true);
```

part_name	mark_number	rows_in_granule	id	laser_id	id.dict.mark	id.mark	first_name.mark
all_4_34_2	0	8192	2315803445121	lCKXLO1N6kOw	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "0", "offset_in_d
all_4_34_2	1	8192	2315803453313	71PSbHm39nK1	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "0", "offset_in_d
all_4_34_2	2	8192	2315803461505	gnS1vJN3QpqM	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "85016", "offset_
all_4_34_2	3	8192	2315803469697	iKmbiD8517vL	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "85016", "offset_
all_4_34_2	4	8192	2315803477889	J7TCaqBrAPUw	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "169773", "offset
all_4_34_2	5	8192	2315803486081	R1qae4X1dOyS	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "169773", "offset
all_4_34_2	6	8192	2315803494273	hYmA2lsSOQf3	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "254802", "offset
all_4_34_2	7	8192	2315803502465	oCFGflxfChZy	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "254802", "offset
all_4_34_2	8	8192	2315803510657	roctBryUxh16	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "339767", "offset
all_4_34_2	9	8192	2315803518849	9Z87fZ3Y7dke	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "339767", "offset
all_4_34_2	10	8192	2315803527041	KDQZV5v8TlBJ	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "424413", "offset
all_4_34_2	11	8192	2315803535233	bF8AGOevupm0	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "424413", "offset
all_4_34_2	12	8192	2315803543425	rl9uDyPvwZAz	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "509197", "offset

Data Skipping Indexes

Read data only in granules that store values in the searched range.



- Type
  - minmax
  - set
  - Bloom Filter

Suitable

- Range values such as DateTime, id
- Avoid use in more same value across multiple granules.

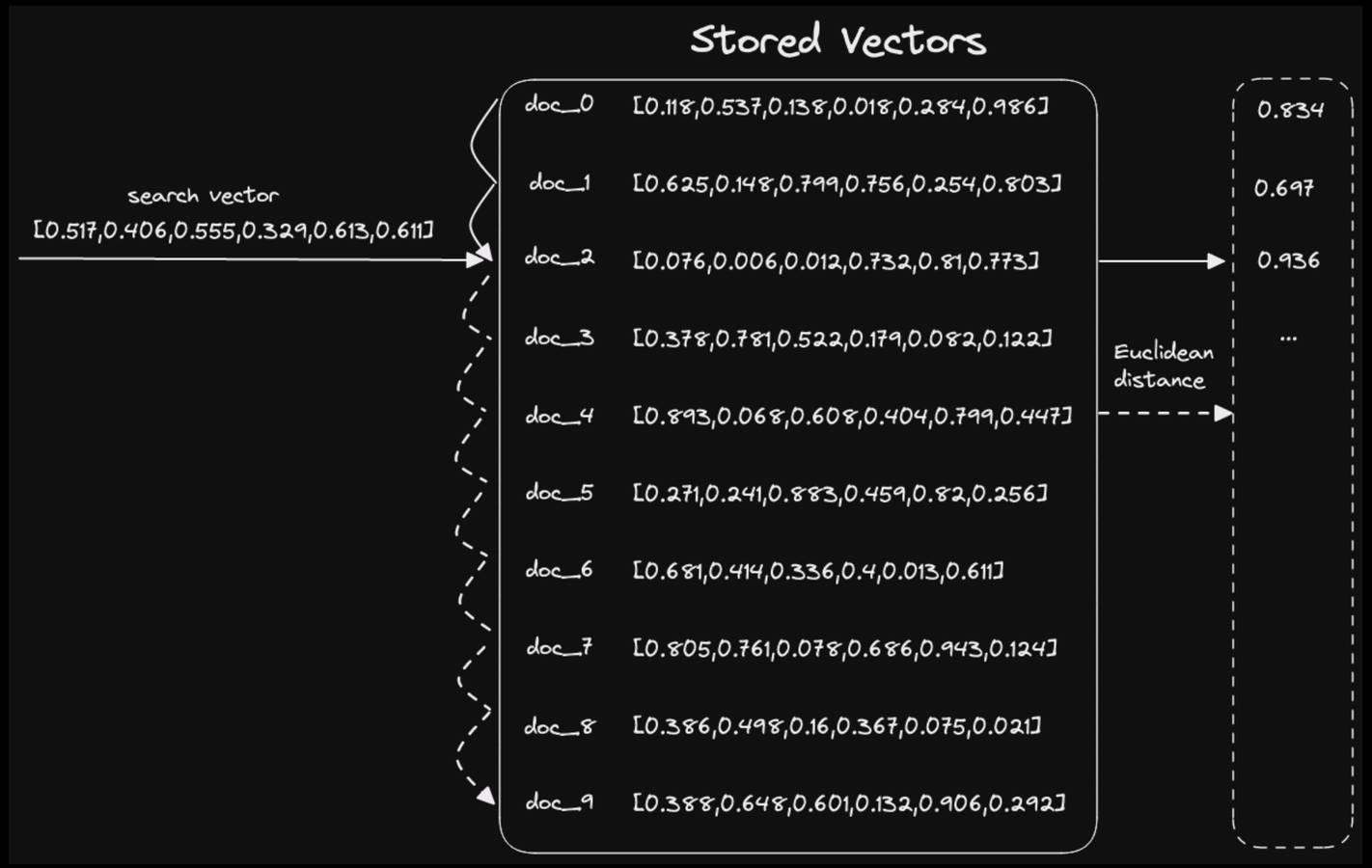
```
CREATE TABLE IF NOT EXISTS sample_table
(
    column1 Int32,
    column2 String,
    column3 Float64
) ENGINE = MergeTree
ORDER BY column1;

-- Create a Data Skipping Index on column1
CREATE INDEX idx_column1_skip
    ON sample_table(column1)
    TYPE minmax
    GRANULARITY 1;

insert into sample_table
values
(1, 'Hello 1', 99), (2, 'Hello 2',  5000),(3, 'Hello 3', 100), (4, 'Hello 4',  5009)
```

part_name	mark_number	rows_in_granule	column1
Filter...	Filter...	Filter...	Filter...
all_1_1_0	0	2	1
all_1_1_0	1	0	2
all_2_2_0	0	2	3
all_2_2_0	1	0	4

Approximate Nearest Neighbor Search Indexes

Arise by INFINITAS

Nearest neighborhood search is the problem of finding the M closest points for a given point in an N-dimensional vector space. The most straightforward approach to solve this problem is a brute force search where the distance between all points in the vector space and the reference point is computed.




```sql
CREATE TABLE IF NOT EXISTS my_table (
    id UInt32,
    vector Array(Float32)
) ENGINE = MergeTree
Primary Key (id)

-- Insert sample data
INSERT INTO my_table (id, vector)
VALUES
    (1, [1.0, 2.0, 3.0]),
    (2, [2.0, 3.0, 4.0]),
    (3, [3.0, 4.0, 5.0]),
    (4, [4.0, 5.0, 6.0]);

-- Create the Usearch index
ALTER TABLE my_table ADD INDEX vector_ann_index vector type usearch('cosineDistance', 'f32')
```

Full-text Search using Inverted Indexes

Arise by INFINITAS

Inverted indexes are an experimental type of secondary indexes which provide fast text search capabilities for String or FixedString columns. The main idea of an inverted index is to store a mapping from "terms" to the rows which contain these terms. "Terms" are tokenized cells of the string column. For example, the string cell "I will be a little late" is by default tokenized into six terms "I", "will", "be", "a", "little" and "late"

```
ALTER TABLE tab ADD INDEX inv_idx(s) TYPE inverted(2);
```

```
INSERT INTO tab(key, str) values (1, 'Hello World');
SELECT * from tab WHERE str == 'Hello World';
SELECT * from tab WHERE str IN ('Hello', 'World');
SELECT * from tab WHERE str LIKE '%Hello%';
SELECT * from tab WHERE multiSearchAny(str, ['Hello', 'World']);
SELECT * from tab WHERE hasToken(str, 'Hello');
```

- inverted(0) (or shorter: inverted()) set the tokenizer to "tokens", i.e. split strings along spaces,
- inverted(N) with N between 2 and 8 sets the tokenizer to "ngrams(N)"

The maximum rows per postings list can be specified as the second parameter. This parameter can be used to control postings list sizes to avoid generating huge postings list files. The following variants exist:

- inverted(ngrams, max_rows_per_postings_list): Use given max_rows_per_postings_list (assuming it is not 0)
- inverted(ngrams, 0): No limitation of maximum rows per postings list
- inverted(ngrams): Use a default maximum rows which is 64K.

Query Profiling

ClickHouse runs sampling profiler that allows analyzing query execution. Using profiler you can find source code routines that used the most frequently during query execution. You can trace CPU time and wall-clock time spent including idle time.

```
select *
FROM system.trace_log limit 10
```

hostname	event_date	event_time	event_time_microsec...	timestamp_ns	revisio
Filter...	Filter...	Filter...	Filter...	Filter...	Filt
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.611203	17075505236112035...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.611204	17075505236112045...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.6544...	1707550523654443...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.6544...	1707550523654444...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.6544...	1707550523654453...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.6544...	1707550523654453...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:30	2024-02-10 07:35:30.627719	1707550530627719...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:30	2024-02-10 07:35:30.627726	1707550530627726...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:30	2024-02-10 07:35:30.636901	1707550530636901...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:30	2024-02-10 07:35:30.636970	1707550530636970...	54482

Trace Type

- Real
- CPU
- Memory
- MemorySample
- MemoryPeak
- ProfileEvent

https://clickhouse.com/docs/en/operations/system-tables/trace_log

* Need to set up at config.xml.

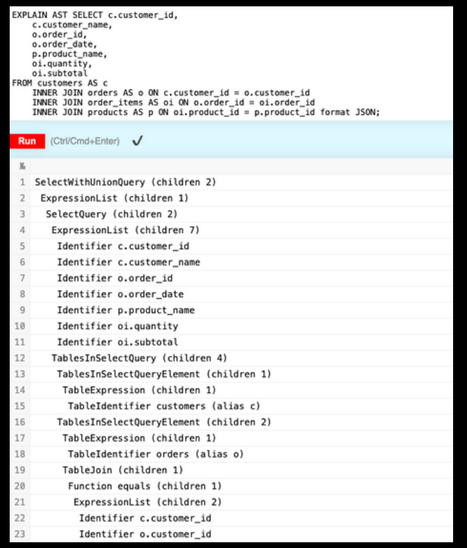
EXPLAIN

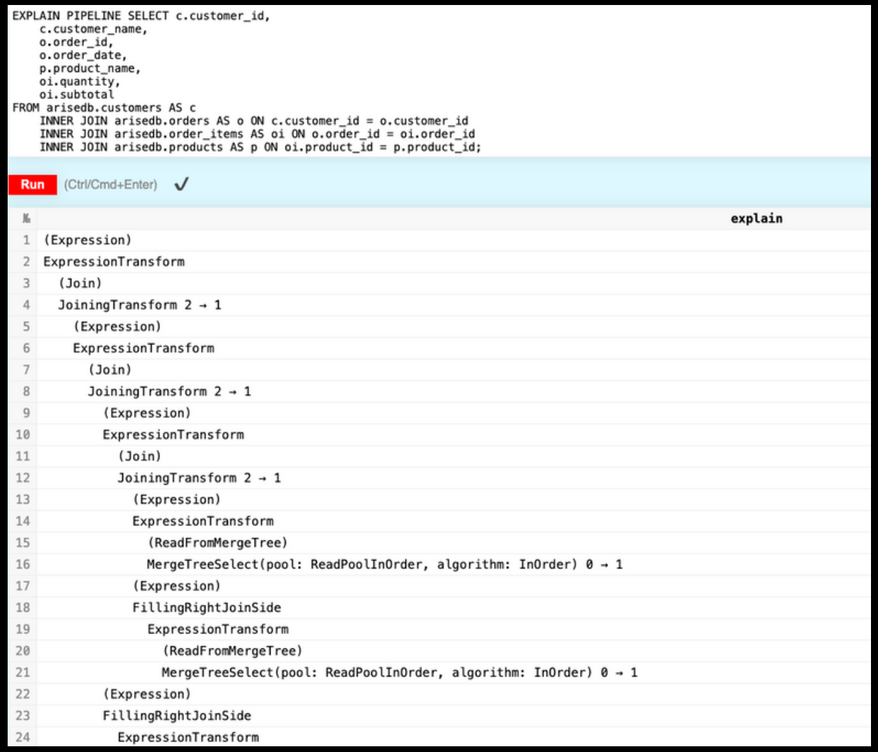
EXPLAIN: Use the EXPLAIN keyword before your query to get an execution plan without actually running the query. This can be useful for understanding how ClickHouse processes your query.

EXPLAIN Types

- AST — Abstract syntax tree.
- SYNTAX — Query text after AST-level optimizations.
- QUERY TREE — Query tree after Query Tree level optimizations.
- PLAN — Query execution plan.
- PIPELINE — Query execution pipeline.

```
EXPLAIN AST  SELECT c.customer_id,
    c.customer_name,
    o.order_id,
    o.order_date,
    p.product_name,
    oi.quantity,
    oi.subtotal
FROM customers AS c
    INNER JOIN orders AS o ON c.customer_id = o.customer_id
    INNER JOIN order_items AS oi ON o.order_id = oi.order_id
    INNER JOIN products AS p ON oi.product_id = p.product_id
    FORMAT JSON;
```





https://clickhouse.com/docs/en/sql-reference/statements/explain

Summary

OVERVIEW

- Support standard SQL
- More built-in aggregate functions
- Understand Sparse and Skipping Index
- Include Performance monitoring and analyze with trace_log and explain.

QUERIES

- Insert
- Select
- Avoid Update and Delete
- Join
- Sub Queries
- Aggregation

PERFORMANE OPTIMIZATION

- Sparse Index
- Skipping Index
- Approximate Nearest Neighbor Search Indexes [Experimental]
- Inverted Index [Experimental]
- Query Profiler
- Explain

KEY TAKEAWAYS

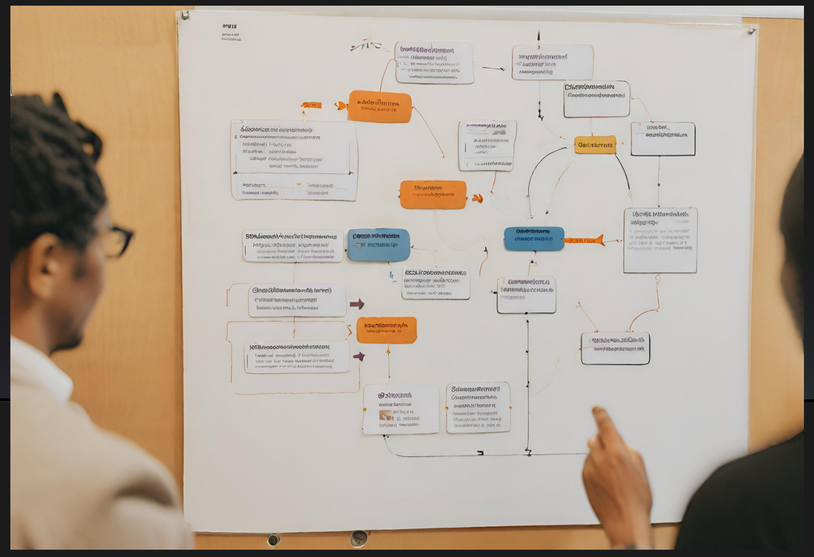
- Basic Queries
- Advanced Queries
- Index for fast Query
- Query Profiler for Monitoring
- Explain for Query Performance Analyzer.

ClickHouse

Arise by INFINITAS

Module 3

ClickHouse Data Modeling



MODULE 3 : CLICKHOUSE DATA MODELING

3.1 Schema Design

- Choosing the right data types
- Designing efficient tables
- Normalization and denormalization

3.2 Integration Methods

- Using the ClickHouse client libraries
- Integrating with admin tools

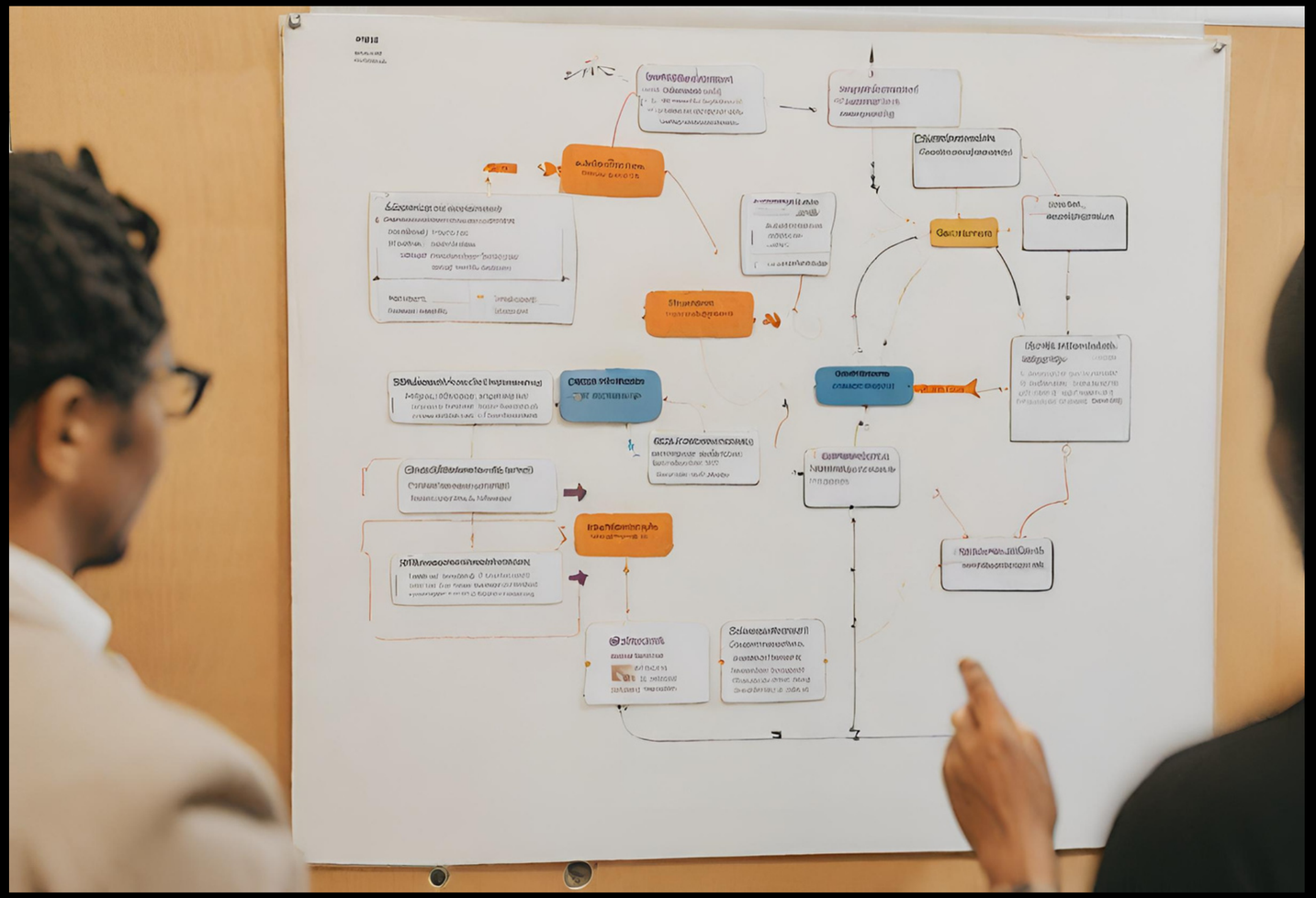
3.3 Data Transformation

- View
- Materialized views
- Aggregating tables
- Custom transformations using lamda expression
- Custom transformations using Other language

Choosing the right data types

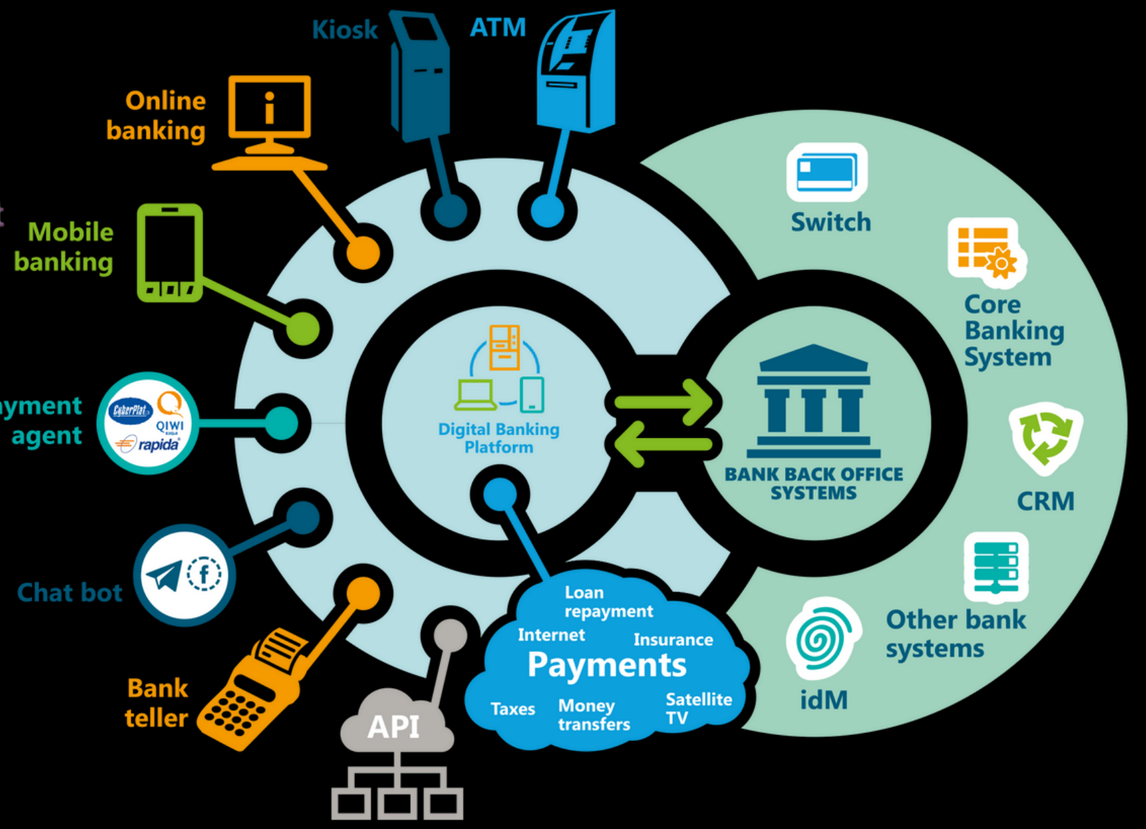
9 Steps

- Understand Your Data
- Use Fixed-size Data Types When Possible
- Leverage Low Cardinality Types
- Consider Enumerations for Categorical Data
- Use Appropriate Decimal Precision
- Choose Date and DateTime Types
- Explore String Types
- Optimize for Compression
- Regularly Review and Adjust



1. Understand Your Data

- Analyze the nature of your data, including its range, precision, and characteristics.
- Identify common patterns and distributions within your dataset.



2. Use Fixed-size Data Types When Possible

Arise by INFINITAS

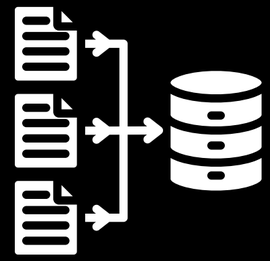
Storage Efficiency

- Fixed-size data types don't require additional space
- Consistent amount of space
- Predictable Storage Space
- Reduced Storage Overhead

Query Performance

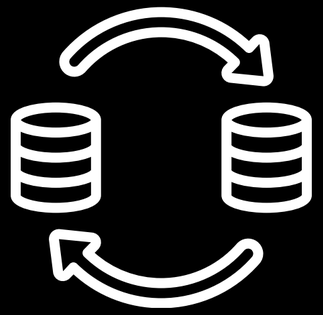
- Sequential Access
  - Size of each record is constant
  - Quicker sequential reads and aggregations
- Indexing Efficiency

Data Integrity



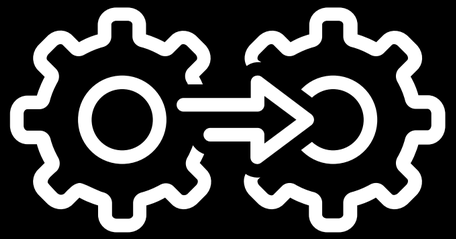
- Avoids Variable-Length Issues
- Prevents Buffer Overflows

Simplicity in Schema Design



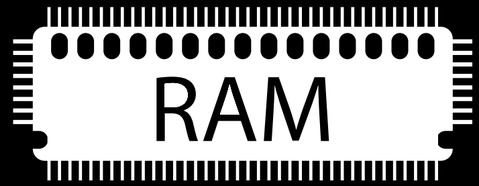
- In Migration work or software development, Data Types with Fixed Size such as Int, Float will have the opportunity to change the size infrequently. If calculated and selected appropriately from the beginning.

Compatibility and Interoperability



- Consistent representation of data across different systems and platforms

Improved Memory Usage



- Predictable Memory Consumption

3. Leverage Low Cardinality Types

LowCardinality is a superstructure that changes a data storage method and rules of data processing. ClickHouse applies dictionary coding to LowCardinality-columns. Operating with dictionary encoded data significantly increases performance of SELECT queries for many applications. Data type support String, FixedString, Date, DateTime, and numbers.

```
LowCardinality(data_type)
```

- Reduced Storage
- Improved Query Performance
- Memory Efficiency
- Enhanced Indexing
- Optimized for Aggregations
- Reduced I/O Operations

```
CREATE TABLE lc_t
(
    `id` UInt16,
    `strings` LowCardinality(String)
)
ENGINE = MergeTree()
ORDER BY id
```

4. Consider Enumerations for Categorical Data

In the context of databases and programming, an enumeration (often referred to as an enum) is a user-defined data type consisting of a set of named values, which represent distinct elements of a finite set. example:

- 1 = Deposit
- 2 = Withdraw
- 3 = Transfer
- 4 = Payment

```
CREATE TABLE t_transaction_history
(
    id UUID,
    account_id FixedString(10),
    transaction_type Enum('Deposit' =1, 'Withdraw' = 2, 'Transfer' = 3, 'Payment' =4),
    amount Decimal(18,6)
)
ENGINE = MergeTree()
Primary Key (id, account_id)
Order by (id, account_id, transaction_type)

insert into
t_transaction_history
(id, account_id, transaction_type , amount)

select generateUUIDv4() as id   ,
 toFixedString(toString(1000000000 + CAST(1000000000  * rand() AS UInt32)),10)  as account_id,
1 + CAST(3 * rand() AS UInt8) % 4 as transaction_type,
100 + CAST(4900 * rand() AS Int32) % 5000  as amount
from numbers(100)


 select * from t_transaction_history
```

5. Use Appropriate Decimal Precision

Recommendation for Financial, Insurance and Blockchain

- Decimal (x,6)
- Decimal(x,8)
- Decimal(x,15) Some system need to advice to Mathematic

```
select 1000 /  0.787564 , 1000 * 0.787564, 1000/0.78 , 1000 * 0.78
```

divide(1000, 0....	multiply(1000, ...	divide(1000, 0....	multiply(1000, ...
abc Filter...	abc Filter...	abc Filter...	abc Filter...
1269.7380784291815	787.5640000000001	1282.051282051282	780

http://www.sysmod.com/eurofaq.htm#ROUNDING

6. Choose Date and DateTime Types

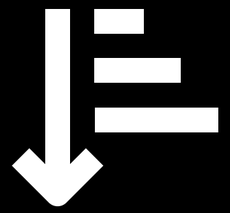
Arise by INFINITAS

Accurate Timestamps

- Incorrect Calcuate Leap Years
- Convert from A.D. to B.E

2024 -> 2567

Chronological Sorting



"DD/MM/YYYY"
Task A: Due 01/03/2024
Task B: Due 12/12/2023

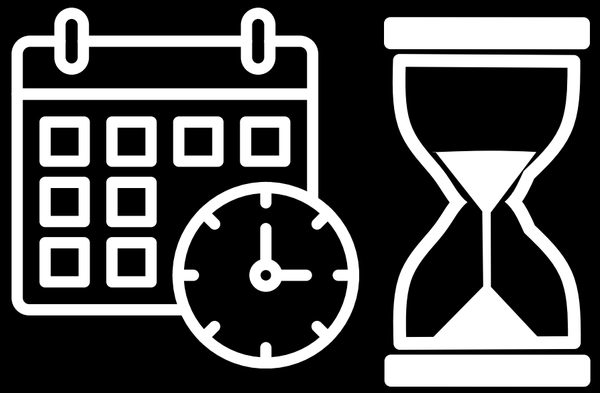
ds
Filter...
01/03/2024
12/12/2023

Event Logging and Audit Trails

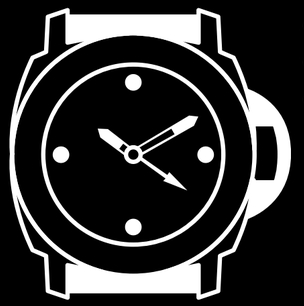
hostname	event_date	event_time	event_time_microsec...	timestamp_ns	revisio
Filter...	Filter...	Filter...	Filter...	Filter...	Fil
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.611203	1707550523611203...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.611204	1707550523611204...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.6544...	1707550523654443...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.6544...	1707550523654444...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.6544...	1707550523654453...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:23	2024-02-10 07:35:23.6544...	1707550523654453...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:30	2024-02-10 07:35:30.627719	1707550530627719...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:30	2024-02-10 07:35:30.627726	1707550530627726...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:30	2024-02-10 07:35:30.636901	1707550530636901...	54482
196cd9dc3323	2024-02-10	2024-02-10 07:35:30	2024-02-10 07:35:30.636970	1707550530636970...	54482

Compatibility with Time Zones

Precision in Duration Calculations



Standardize Timezone



- UTC
- UTC + n
- Must use only one timezone in database

7. Explore String Types

Arise by INFINITAS

Storage Efficiency

- Using FixedString for fields with a constant length can reduce storage overhead.

FixedString(100)

Variable-Length Strings

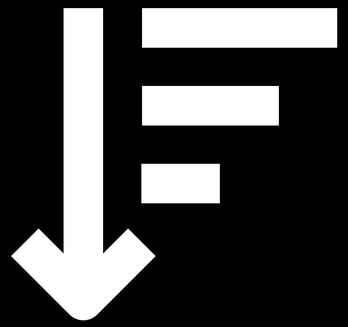
- **String** types, which are variable-length, provide flexibility in storing strings of varying lengths

Character Set Support

- Character Encoding : UTF-8

Collation and Sorting

- UTF-8



Data Validation and Constraints:

- **Phone**
- **Zipcode**

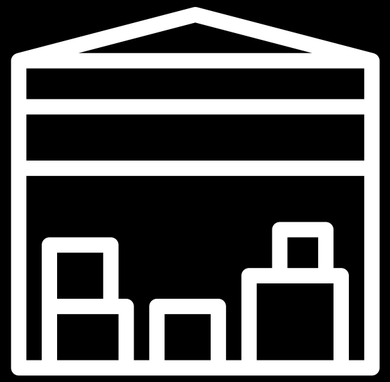
```
CREATE TABLE my_table (
    id UInt32,
    phone_number FixedString(12) CODEC(ZSTD),
    CONSTRAINT check_phone_number CHECK phone_number REGEXP '^\\d{3}-\\d{3}-\\d{4}$'
) ENGINE = MergeTree
PRIMARY KEY (id);
```

Pattern Matching and Indexing

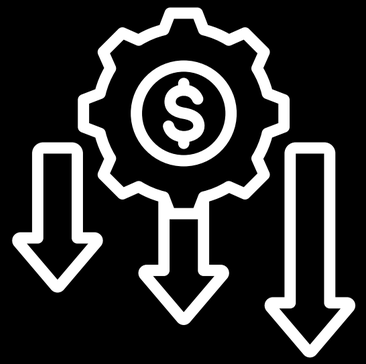
- Regular Expression
- Text Search

8. Optimize for Compression

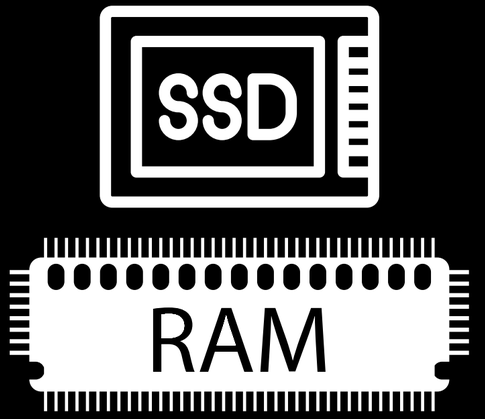
Storage Space Savings



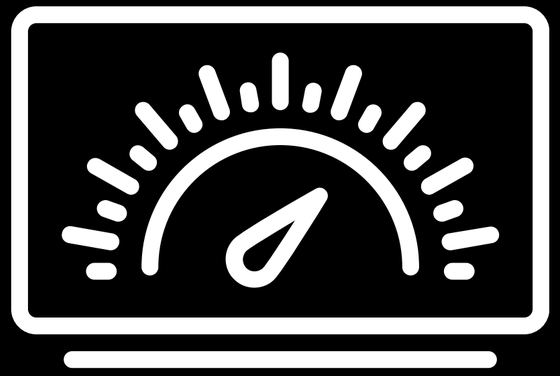
Lower Storage Costs



Faster I/O Operations

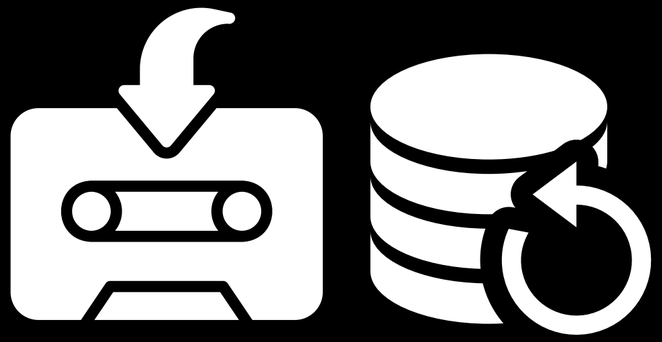


Bandwidth Efficiency:



Improved Query Performance

Backup and Restore Efficiency



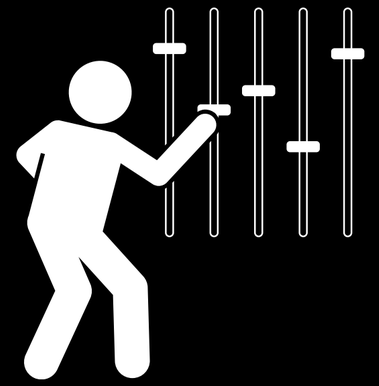
9. Regularly Review and Adjust

Arise by INFINITAS

Adaptation to Business Changes

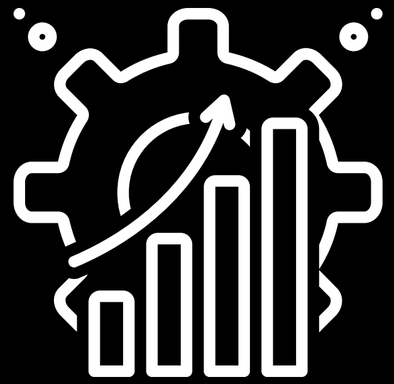
Index Maintenance

Query Tuning



Compliance Requirements

Capacity Planning



User Access and Permissions

Special Type by Use cases

Arise by INFINITAS

UUID

The UUID type represents a universally unique identifier. It is a 128-bit value typically used to uniquely identify entities in a distributed system or database.
61f0c404-5cb3-11e7-907b-a6006ad3dba0

IPv4, v6

IPv4 addresses. Stored in 4 bytes as UInt32. ex. 116.106.34.242

Tuple

A tuple of elements, each having an individual type.

```
`a` Tuple(s String, i Int64)
```

Array

An array of T-type items, with the starting array index as 1. T can be any data type, including an array.

```
Array(Nullable(UInt8))
```

Variant

This type represents a union of other data types. Type Variant(T1, T2, ..., TN)

```
v Variant(UInt64, String, Array(UInt64))
```

Map

Map(key, value) data type stores key:value pairs.

```
a Map(String, UInt64)
```

JSON

Stores JavaScript Object Notation (JSON) documents in a single column.

```
o JSON
'{"a": 1, "b": { "c": 2, "d": [1, 2, 3] }}'
```

Nested

A nested data structure is like a table inside a cell.

```
...
    Goals Nested
    (
        ID UInt32,
        Serial UInt32,
        EventTime DateTime,
        Price Int64,
        OrderID String,
        CurrencyID UInt32
    ),
```

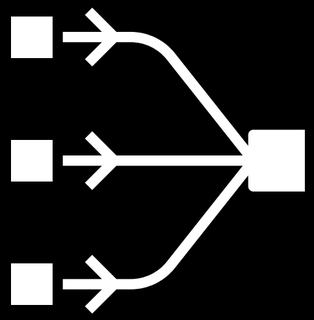
Geo

ClickHouse supports data types for representing geographical objects — locations, lands, etc.

```
CREATE TABLE geo_point (p Point) ENGINE = Memory();
INSERT INTO geo_point VALUES((10, 10));
```

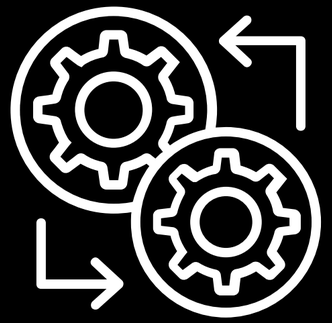
Table Engine

There are 4 Engine Families:



MergeTree

- MergeTree
- ReplacingMergeTree
- SummingMergeTree
- AggregatingMergeTree
- CollapsingMergeTree
- VersionedCollapsingMergeTree
- GraphiteMergeTree

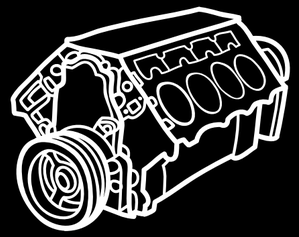


Integration Engines

- ODBC
- JDBC
- MySQL
- MongoDB
- Redis
- HDFS
- S3
- Kafka
- EmbeddedRocksDB
- RabbitMQ
- PostgreSQL
- S3Queue
- SQLite
- etc

Log

- TinyLog
- StripeLog
- Log



Special Engines

- Distributed
- Dictionary
- Merge
- File
- Null
- Set
- Join
- URL
- View
- Memory
- Buffer
- KeeperMap

3.1.2.1 Table Engines

MergeTree

Arise by INFINITAS

The most universal and functional table engines for **high-load tasks.** The property shared by these engines is **quick data insertion** with subsequent background data processing. MergeTree family engines **support data replication** (with Replicated* versions of engines), **partitioning, secondary data-skipping indexes**, and other features not supported in other engines.

ReplacingMergeTree

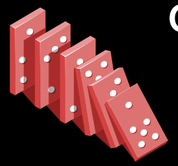
- The engine differs from MergeTree in that it removes duplicate entries with the same sorting key value (ORDER BY table section, not PRIMARY KEY).
- Use for Master Table.

SummingMergeTree

- Replaces all the rows with the same primary key.
- One row which contains summarized values for the columns with the numeric data type.
- Use for Summary Report.

AggregatingMergeTree

- Replaces all rows with the same primary key
- With a single row (within a one data part) that stores a combination of states of aggregate functions.
- Use for Aggregate Report.

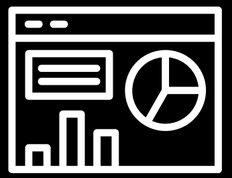


CollapsingMergeTree

- Asynchronously deletes (collapses) pairs of rows if all of the fields in a **sorting key**
- Use for Accumulate last stage data such as CurrentBalance.

VersionedCollapsingMergeTree

- Allows quick writing of object states that are continually changing.
- Deletes old object states in the background. This significantly reduces the volume of storage.
- Use for Historical Data, Configuration, Templates.



GraphiteMergeTree

- Use for Time Series, Real-time Monitoring Data.

ReplacingMergeTree

When using the ReplacingMergeTree engine in ClickHouse for a BankAccount table, the data structure typically includes a primary key that uniquely identifies each bank account. The primary key is used by the ReplacingMergeTree engine to replace old data with new data for the same primary key value.

```
CREATE TABLE BankAccount
(
    account_id String,
    account_holder String,
    balance Decimal(18, 2),
    last_updated DateTime,

    PRIMARY KEY account_id
)
ENGINE = ReplacingMergeTree(last_updated)
ORDER BY (account_id);
```

```
INSERT INTO BankAccount (account_id, account_holder, balance, last_updated
VALUES
    ('A001', 'John Doe', 5000.00, '2023-05-01 10:00:00'),
    ('B002', 'Jane Smith'  2500.75, '2023-05-01 11:30:00'),
    ('C003', 'Michael Johnson', 7800.25, '2023-05-01 09:15:00');
```

```
-- Updating the balance for account 'A001'
INSERT INTO BankAccount (account_id, account_holder, balance, last_updated)
VALUES
    ('A001', 'John Doe', 6000.00, '2023-05-02 14:45:00');
```

```
┌─account_id─┬─account_holder─┬─balance─┬─last_updated────────┐
│ A001       │ John Doe       │  5000.00│ 2023-05-01 10:00:00│
│ B002       │ Jane Smith     │  2500.75│ 2023-05-01 11:30:00│
│ C003       │ Michael Johnson│  7800.25│ 2023-05-01 09:15:00│
│ A001       │ John Doe       │  6000.00│ 2023-05-02 14:45:00│
└────────────┴────────────────┴─────────┴─────────────────────┘
```

```
select * from BankAccount final
```

```
INSERT INTO BankAccount (account_id, account_holder, balance, last_updated)
VALUES
    ('D004', 'Emily Davis', 3200.50, '2023-05-02 16:20:00');
```

```
OPTIMIZE TABLE BankAccount FINAL DEDUPLICATE;
```

```
┌─account_id─┬─account_holder─┬─balance─┬─last_updated────────┐
│ A001       │ John Doe       │  6000.00│ 2023-05-02 14:45:00│
│ B002       │ Jane Smith     │  2500.75│ 2023-05-01 11:30:00│
│ C003       │ Michael Johnson│  7800.25│ 2023-05-01 09:15:00│
│ D004       │ Emily Davis    │  3200.50│ 2023-05-02 16:20:00│
└────────────┴────────────────┴─────────┴─────────────────────┘
```

CollapsingMergeTree

When using the CollapsingMergeTree engine in ClickHouse for an AccountBalance table, the data structure typically includes a primary key that uniquely identifies the account, along with columns for the balance and other relevant information.

```
CREATE TABLE AccountBalance
(
    account_id String,
    account_holder String,
    last_transaction_time DateTime,
    amount Decimal(18, 2),
    sign Int8 default 1,
    PRIMARY KEY (account_id, last_transaction_time)
)
ENGINE = CollapsingMergeTree(sign)
PARTITION BY toYYYYMM(last_transaction_time)
ORDER BY (account_id, last_transaction_time)
```

```
INSERT INTO AccountBalance (account_id, account_holder, last_transaction_time, amount)
VALUES
    ('A001', 'John Doe', '2023-05-01 10:00:00', 500.00),
    ('B002', 'Jane Smith', '2023-05-01 09:15:00', 1000.00),
    ('C003', 'Michael Johnson', '2023-05-02 11:10:00', 2500.00);
```

```
INSERT INTO AccountBalance (account_id, account_holder, last_transaction_time, amount, sign)
VALUES
    ('A001', 'John Doe', '2023-05-01 15:45:00', 150.00,1),
    ('B002', 'Jane Smith', '2023-05-01 18:20:00', 300.00,1),
    ('C003', 'Michael Johnson', '2023-05-02 14:30:00', 1200.00,1),
    ('A001', 'John Doe', '2023-05-01 10:00:00', 500.00, -1),
    ('B002', 'Jane Smith', '2023-05-01 09:15:00', 1000.00, -1),
    ('C003'  'Michael Johnson', '2023-05-02 11:10:00', 2500.00, -1);
```

account_id	account_holder	last_transactio...	amount	sign
Filter...	Filter...	Filter...	Filter...	Fi
A001	John Doe	2023-05-01 15:45:00	150	1
B002	Jane Smith	2023-05-01 18:20:00	300	1
C003	Michael Johnson	2023-05-02 14:30:00	1200	1


```
select * from AccountBalance final
```

SummingMergeTree

- Replaces all the rows with the same primary key.
- One row which contains summarized values for the columns with the numeric data type.
- Use for Summary Report.

```
CREATE TABLE IF NOT EXISTS transaction_history (
    transaction_id UUID,
    transaction_type Enum('deposit' =1, 'withdrawal'=2, 'transfer' =3  'payment' =4),
    account_id UInt64,
    ref_num UUID,
    amount Float64,
    transaction_date DateTime DEFAULT now(),
    created_at DateTime DEFAULT now(),
    created_by UInt64,
    remarks Nullable(String),
    status_id Nullable(UInt8)
) ENGINE = SummingMergeTree(amount)
Primary Key (account_id,ref_num)
PARTITION BY toYYYYMM(transaction_date)
ORDER BY (account_id, ref_num, transaction_date, transaction_type);
```

```
INSERT INTO transaction_history (transaction_id, transaction_type, account_id, ref_num, amount,
transaction_date, created_at, created_by, remarks, status_id)
VALUES
    ('00000000-0000-0000-0000-000000000001', 'deposit', 1001, '11111111-1111-1111-1111-111111111101',
1000.00, '2024-02-01 12:00:00', '2024-02-01 12:00:00', 1, 'Initial deposit', 1),
    ('00000000-0000-0000-0000-000000000002', 'withdrawal', 1001,
'11111111-1111-1111-1111-111111111102', -500.00, '2024-02-05 15:30:00', '2024-02-05 15:30:00', 2,
'ATM withdrawal', 1),
    ('00000000-0000-0000-0000-000000000003', 'transfer', 1001,
'11111111-1111-1111-1111-111111111103', -200.00, '2024-02-10 09:45:00', '2024-02-10 09:45:00', 1,
'Transfer to savings account', 1),
    ('00000000-0000-0000-0000-000000000004', 'payment', 1001, '11111111-1111-1111-1111-111111111104',
-300.00, '2024-02-15 14:20:00', '2024-02-15 14:20:00', 3, 'Credit card payment', 1),
    ('00000000-0000-0000-0000-000000000005', 'deposit', 1002, '11111111-1111-1111-1111-111111111105',
1500.00, '2024-02-02 10:00:00', '2024-02-02 10:00:00', 2, 'Salary credit', 1),
    ('00000000-0000-0000-0000-000000000006', 'withdrawal', 1002,
'11111111-1111-1111-1111-111111111106', -700.00, '2024-02-06 11:45:00', '2024-02-06 11:45:00', 1,
'Grocery shopping', 1),
    ('00000000-0000-0000-0000-000000000007', 'transfer', 1002,
'11111111-1111-1111-1111-111111111107', -300.00, '2024-02-11 16:00:00', '2024-02-11 16:00:00', 3,
'Transfer to investment account', 1),
    ('00000000-0000-0000-0000-000000000008', 'payment', 1002, '11111111-1111-1111-1111-111111111108',
-500.00, '2024-02-20 08:30:00', '2024-02-20 08:30:00', 2, 'Utility bill payment', 1),
    ('00000000-0000-0000-0000-000000000009', 'deposit', 1001, '11111111-1111-1111-1111-111111111109',
900.00, '2024-02-21 12:00:00', '2024-02-21 12:00:00', 1, 'Deposit', 1),
    ('00000000-0000-0000-0000-000000000010', 'deposit', 1002, '11111111-1111-1111-1111-111111111110',
500.00, '2024-02-21 12:00:00', '2024-02-21 12:00:00', 2, 'Deposit', 1);
```

```
select account_id , sum(amount) as totalAccount
from transaction_history
group by account_id
```

account_id	totalAccount
abc Filter...	abc Filter...
1001	900
1002	500

VersionedCollapsingMergeTree

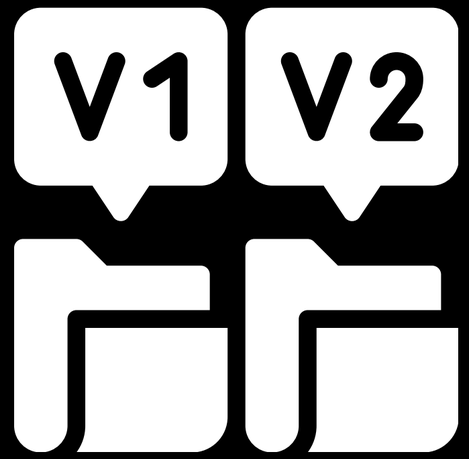
- Allows quick writing of object states that are continually changing.
- Deletes old object states in the background. This significantly reduces the volume of storage.
- Use for Historical Data, Configuration, Templates.

```
CREATE TABLE IF NOT EXISTS user_account_states (
    user_id UInt64,
    created_at DateTime default now(),
    created_by UInt64,
    remarks Nullable(FixedString(255)),
    status_id Enum('Inactive'  0, 'Active' = 1, 'Banned' = 2, 'Locked' =3, 'Unscription' =4),
    sign Int8,
    version UInt32
) ENGINE = VersionedCollapsingMergeTree(sign, version)
Primary Key (user_id)
PARTITION BY toYYYYMM(created_at)
ORDER BY (user_id, created_at);
```

```
INSERT INTO user_account_states ( user_id,created_by, remarks,status_id, sign, version)
VALUES
    (1001,  1001,'Register', 0,  1, 1),
    (1002, 1002, 'Register', 0, 1, 1);
```

```
INSERT INTO user_account_states ( user_id,created_by, remarks,status_id, sign, version)
VALUES
    (1001, 1001, 'Active',1 ,1, 2);
```

user_id	created_at	created_by	remarks	status_id	sign	version
Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...
1001	2024-03-14 13:59:08	1001	Active	Active	1	2
1001	2024-03-14 13:59:05	1001	Register	Inactive	1	1
1002	2024-03-14 13:59:05	1002	Register	Inactive	1	1



AggregatingMergeTree

- Replaces all rows with the same primary key
- With a single row (within a one data part) that stores a combination of states of aggregate functions.
- Use for Aggregate Report.

transaction_date	transaction_type	account_id	_count	_sum	_avg	_min	_max	_peak_time
Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...
2023-02-19	Deposit	5079460352	1	1652	1652	1652	1652	2023-02-19 13:33:44
2023-02-19	Withdraw	5079460352	2	-8384	-4192	-4656	-3728	2023-02-19 13:33:44
2023-02-19	Transfer	5079460352	4	-5768	-1442	-2412	-396	2023-02-19 13:33:44
2023-02-19	Payment	5079460352	3	-8992	-2997.333333333333	-3928	-1864	2023-02-19 13:33:44
2016-01-19	Deposit	5079460352	1	1452	1452	1452	1452	2016-01-19 16:36:08
2014-01-17	Deposit	5079460352	1	2692	2692	2692	2692	2014-01-17 11:22:22

```sql
CREATE TABLE IF NOT EXISTS transaction_analytics (
    transaction_date Date,
    transaction_type Enum('Deposit' =1, 'Withdraw' = 2, 'Transfer' = 3, 'Payment' =4),
    account_id UInt64,
    _count UInt64,
    _sum Float64,
    _avg Float64,
    _min Float64,
    _max Float64,
    _peak_time DateTime
) ENGINE = AggregatingMergeTree()
PARTITION BY toYYYYMM(transaction_date)
ORDER BY (transaction_date, transaction_type, account_id);
```

```sql
INSERT INTO transaction_analytics

SELECT
    transaction_date,
    transaction_type,
    account_id,
    count() AS Count,
    sum(amount) AS Sum,
    avg(amount) AS Avg,
    min(amount) AS Min,
    max(amount) AS Max,
    argMax(transaction_date, amount) AS PeakTime
FROM t_transaction_history
GROUP BY
    transaction_date,
    transaction_type,
    account_id;
```

GraphiteMergeTree

This engine is designed for thinning and aggregating/averaging (rollup) Graphite data. It may be helpful to developers who want to use ClickHouse as a data store for Graphite. Use for Time Series, Real-time Monitoring Data.

```
CREATE TABLE IF NOT EXISTS transaction_status_monitoring (
    event_date Date,
    transaction_type String,
    account_id UInt64,
    success Float64,
    event_time DateTime,
    version UInt16
) ENGINE = GraphiteMergeTree('graphite_rollup')
PARTITION BY toYYYYMM(event_date)
ORDER BY (event_time, transaction_type );
```

```
INSERT INTO  transaction_status_monitoring (event_date, transaction_type, account_id, success,
event_time,version )
VALUES
    ('2023-03-13', 'Deposit', 123456, 1, '2023-03-13 10:00:00',1),
    ('2023-03-13', 'Withdrawal', 123456, 1, '2023-03-13 11:00:00',1),
    ('2023-03-13', 'Transfer', 789012, 1, '2023-03-13 12:50:00',1),
    ('2023-03-13', 'Payment', 123456, 1, '2023-03-13 13:00:00',1);
```

config.xml

```
<graphite_rollup>
    <path_column_name>transaction_type</path_column_name>
    <time_column_name>event_time</time_column_name>
    <value_column_name>success</value_column_name>
    <version_column_name>version</version_column_name>
    <pattern>
        <regexp>transaction_status\..*</regexp>
        <function>sum</function>
        <retention>
            <age>0</age>
            <precision>60</precision>
        </retention>
        <retention>
            <age>7</age>
            <precision>3600</precision>
        </retention>
        <retention>
            <age>30</age>
            <precision>86400</precision>
        </retention>
    </pattern>
```

Arise by INFINITAS

Integrate to other engine to get advanced

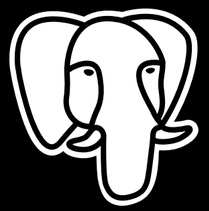
The highlight of Clickhouse is besides inserting data and querying quickly. It can also connect to Datasource Engine from many famous brands. To makes it possible to pull together the outstanding features of that database.

S3

- Able to write SQL, read and write data in Object Storage such as S3 and Minio.

Sqlite

- Borrow ACID Feature

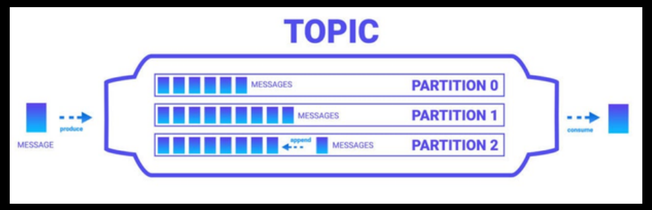


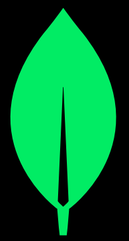
Postgres Network / Mac Address Type

- Connect to Postgres to use the Network Address Type feature to check the range IP address or the number of IPs supported in that range.

Kafka

- Batch Consume for Batch Processing





MongoDB

- Use Mongo as Operation DB integrate to Clickhouse to process analytics report.

Integrate to S3

Arise by INFINITAS

Able to write SQL, read and write data in Object Storage such as S3 and Minio.

```
SELECT *
FROM s3(
   'https://datasets-documentation.s3.eu-west-3.amazonaws.com/aapl_stock.csv',
   'CSVWithNames'
)
LIMIT 5;
```

Date	Open	High	Low	Close	Volume	OpenInt
abc Filter...	abc Filter...	abc Filter...	abc Filter...	abc Filter...	abc Filter...	abc Filter...
1984-09-07	0.4238800000000000	0.42902	0.4187400000000000	0.4238800000000000	23220030	0
1984-09-10	0.4238800000000000	0.4251600000000000	0.41366	0.4213400000000000	18022532	0
1984-09-11	0.4251600000000000	0.43668	0.4251600000000000	0.42902	42498199	0
1984-09-12	0.42902	0.43157	0.4161800000000000	0.4161800000000000	37125801	0
1984-09-13	0.4392700000000000	0.44052	0.4392700000000000	0.4392700000000000	57822062	0

3.1.2.3 Log

Log Engine Family

These engines were developed for scenarios when you need to quickly write many small tables (up to about 1 million rows) and read them later as a whole.
Engines of the family:

Common

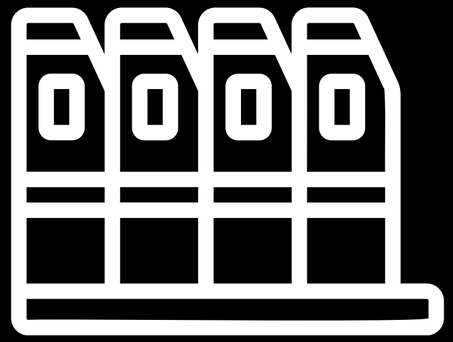
- Store data on a disk.
- Append data to the end of file when writing.
- Support locks for concurrent data access.
- Do not support mutations.
- Do not support indexes.
- Do not write data atomically.

StripeLog

- Use this engine in scenarios when you need to write many tables with a small amount of data (less than 1 million rows).
- **Stores all the columns in one file**

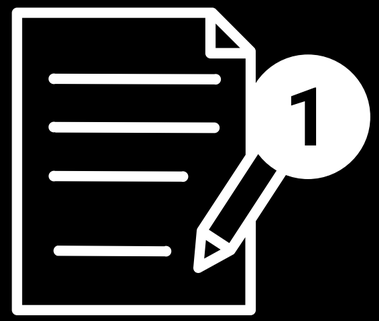
Log

- These marks are written on every data block and contain offsets that indicate where to start reading the file in order to skip the specified number of rows. This makes it possible to **read table data in multiple threads.**



TinyLog

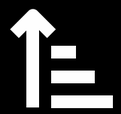
- This table engine is typically used with the **write-once method**: write data one time, then read it as many times as necessary.



3.1.2.3 Log

Log Engine Family


```
CREATE TABLE stripe_log_table
(
    timestamp DateTime,
    message_type String,
    message String
)
ENGINE = StripeLog

INSERT INTO stripe_log_table VALUES (now(),'REGULAR','The first regular message');
INSERT INTO stripe_log_table VALUES (now(),'REGULAR','The second regular
message'),(now(),'WARNING','The first warning message');

SELECT * FROM stripe_log_table
```



timestamp	message_type	message
Filter...	Filter...	Filter...
2024-03-18 16:43:51	REGULAR	The second regular message
2024-03-18 16:43:51	WARNING	The first warning message
2024-03-18 16:43:39	REGULAR	The first regular message

```
CREATE TABLE logs_example (
    timestamp DateTime,
    message String
) ENGINE = Log;

INSERT INTO logs_example (timestamp, message)
VALUES ('2024-03-15 12:00:00', 'Log entry 1'),
       ('2024-03-15 12:05:00', 'Log entry 2'),
       ('2024-03-15 12:10:00', 'Log entry 3');

SELECT *
FROM logs_example
```

timestamp	message
Filter...	Filter...
2024-03-15 12:00:00	Log entry 1
2024-03-15 12:05:00	Log entry 2
2024-03-15 12:10:00	Log entry 3

```
CREATE TABLE example_table (
    timestamp DateTime,
    value Float64
) ENGINE = TinyLog

INSERT INTO example_table (timestamp, value)
VALUES
    ('2024-03-01 12:00:00', 10.5),
    ('2024-03-02 09:30:00', 15.2),
    ('2024-03-03 14:45:00', 20.0),
    ('2024-03-04 08:00:00', 18.7),
    ('2024-03-05 16:20:00', 25.3),
    ('2024-03-06 11:10:00', 30.1),
    ('2024-03-07 13:00:00', 22.6),
    ('2024-03-08 10:45:00', 28.9),
    ('2024-03-09 09:15:00', 35.7),
    ('2024-03-10 15:30:00', 40.2);

    select * from example_table
```

timestamp	value
Filter...	Filter...
2024-03-01 12:00:00	10.5
2024-03-02 09:30:00	15.2
2024-03-03 14:45:00	20
2024-03-04 08:00:00	18.7
2024-03-05 16:20:00	25.3
2024-03-06 11:10:00	30.1
2024-03-07 13:00:00	22.6
2024-03-08 10:45:00	28.9
2024-03-09 09:15:00	35.7
2024-03-10 15:30:00	40.2

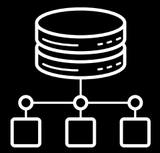
3.1.2.4 Special Engines

Arise by INFINITAS

ClickHouse offers various special-purpose storage engines tailored to specific use cases. These engines provide specialized functionalities and optimizations to meet specific requirements. Some of the special engines available in ClickHouse include:

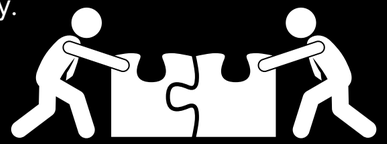
Distributed Table Engine

Tables with Distributed engine do not store any data of their own, but allow distributed query processing on multiple servers.



Merge Table Engine

The Merge engine (not to be confused with MergeTree) does not store data itself, but allows reading from any number of other tables simultaneously.



Executable and ExecutablePool Table Engines

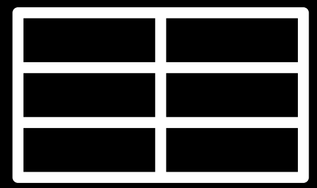
The Executable and ExecutablePool table engines allow you to define a table whose rows are generated from a script that you define (by writing rows to stdout). The executable script is stored in the users_scripts directory and can read data from any source

File Table Engine

The File table engine keeps the data in a file in one of the supported file formats (TabSeparated, Native, etc.).

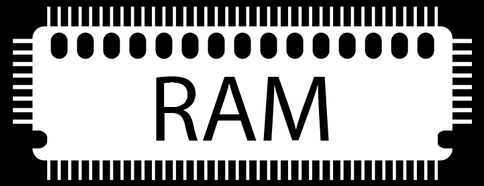
View Table Engine

Used for implementing views (for more information, see the CREATE VIEW query). It does not store data, but only stores the specified SELECT query. When reading from a table, it runs this query (and deletes all unnecessary columns from the query).



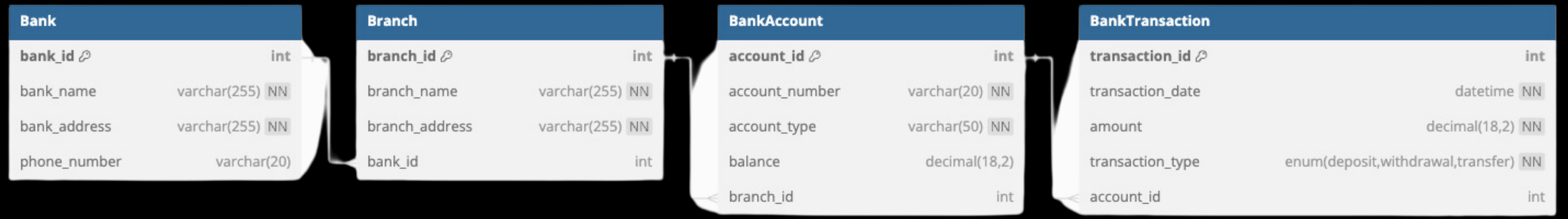
Memory Table Engine

The Memory engine stores data in RAM, in uncompressed form. Data is stored in exactly the same form as it is received when read.



Normalization

- Organizing data into multiple related tables instead of having one large table.
- Eliminating redundant data and ensuring only related data is stored in each table.
- Removes data anomalies and improves data integrity.
- Adhering to normalization forms like 1NF, 2NF, 3NF, BCNF.
- Leads to efficient storage through reduced data redundancy.




```sql
SELECT
    bt.transaction_id,
    bt.transaction_date,
    bt.amount,
    bt.transaction_type,
    b.bank_name,
    br.branch_name,
    ba.account_number
FROM
    bank_transactions bt
INNER JOIN
    bank_account ba ON bt.account_id = ba.account_id
INNER JOIN
    branch br ON ba.branch_id = br.branch_id
INNER JOIN
    bank b ON br.bank_id = b.bank_id;
```

bt.transaction_id	bt.transaction_...	bt.amount	bt.transaction_...	b.bank_name	br.branch_name	ba.account_number
Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...
1	2024-03-01 12:00:00	1000	deposit	Bank A	Branch X	10001
2	2024-03-02 09:30:00	200	withdrawal	Bank A	Branch X	10002
3	2024-03-03 14:45:00	1500	deposit	Bank A	Branch Y	20001
4	2024-03-04 08:00:00	500	withdrawal	Bank B	Branch Z	20002

De-normalization

- Intentionally adding redundant data to tables.
- Joins some previously normalized tables so related data can be queried faster from one table.
- Adding frequently accessed columns into main table to improve read performance.
- Useful when heavy read activity outweighs data updates.
- Prone to data anomalies if not handled properly.

Order Date	Customer Name	Email	Product Name	Quantity	Unit Price	Total Price
2022-01-01	Customer A	customer_a@email.com	Product X	2	50.00	100.00
2022-01-02	Customer B	customer_b@email.com	Product Y	3	30.00	90.00

Using the ClickHouse client libraries

JavaScript

Java

Go

Python

Third-party Developers

https://clickhouse.com/docs/en/interfaces/third-party/client-libraries

Using the admin tools

Arise by INFINITAS

SQLTools
Database management for VS Code

DBeaver

DbVisualizer

jupyter

qStudio

Using the BI tools

Arise by INFINITAS

Metabase

Using the ClickHouse client libraries

Go github.com/ClickHouse/clickhouse-go/v2

Connect / Open (*ConnectionString*)

Success / Error

// Execute SELECT query

Exec / Query

Results

Disconnect / Close

Success / Error

```
db, err := sql.Open("clickhouse", connectionString)
```

```
_, err := db.Exec("INSERT INTO people (id, first_name, last_name, dateOfBirth, laser_id) VALUES (?,
?, ?, ?, ?)", person.ID, person.FirstName, person.LastName, person.DateOfBirth, person.LaserID)
if err != nil { c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
  return
}
```

```
rows, err := db.Query("SELECT * FROM testdb.people WHERE id = ?", id)
```

View

Turns a subquery into a table. The function implements views (see CREATE VIEW). The resulting table **does not store data, but only stores the specified SELECT query.** When reading from the table, ClickHouse executes the query and deletes all unnecessary columns from the result.

```
Create View vw_bank_transaction
as
SELECT
    bt.transaction_id as transaction_id,
    bt.transaction_date as transaction_date,
    bt.amount as amount,
    bt.transaction_type as transaction_type,
    b.bank_name as bank_name,
    br.branch_name as branch_name,
    ba.account_number as account_number
FROM
    bank_transactions bt
INNER JOIN
    bank_account ba ON bt.account_id = ba.account_id
INNER JOIN
    branch br ON ba.branch_id = br.branch_id
INNER JOIN
    bank b ON br.bank_id = b.bank_id;

select * from vw_bank_transaction

select * from system.tables where engine = 'View'
```

transaction_id	transaction_date	amount	transaction_type	bank_name	branch_name	account_number
1	2024-03-01 12:00:00	1000	deposit	Bank A	Branch X	10001
2	2024-03-02 09:30:00	200	withdrawal	Bank A	Branch X	10002
3	2024-03-03 14:45:00	1500	deposit	Bank A	Branch Y	20001
4	2024-03-04 08:00:00	500	withdrawal	Bank B	Branch Z	20002

database	name	uuid	engine
INFORMATION_SCH...	COLUMNS	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	KEY_COLUMN_USAGE	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	REFERENTIAL_CONSTR...	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	SCHEMATA	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	STATISTICS	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	TABLES	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	VIEWS	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	columns	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	key_column_usage	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	referential_constraints	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	schemata	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	statistics	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	tables	00000000-0000-0000-0000-0000000...	View
INFORMATION_SCH...	views	00000000-0000-0000-0000-0000000...	View
default	vw_bank_transaction	654a509a-9ca8-4c97-9f00-44ad5038...	View
information_schema	COLUMNS	00000000-0000-0000-0000-0000000...	View
information_schema	KEY_COLUMN_USAGE	00000000-0000-0000-0000-0000000...	View

as_select
SELECT database AS table_catalog, database AS table_schema, `table` AS table_name, nam...
SELECT 'def' AS constraint_catalog, database AS constraint_schema, 'PRIMARY' AS constrai...
SELECT '' AS constraint_catalog, NULL AS constraint_name, '' AS constraint_schema, '' AS u...
SELECT name AS catalog_name, name AS schema_name, 'default' AS schema_owner, NULL ...
SELECT '' AS table_catalog, '' AS table_schema, '' AS table_name, 0 AS non_unique, '' AS ind...
SELECT database AS table_catalog, database AS table_schema, name AS table_name, multilf...
SELECT database AS table_catalog, database AS table_schema, name AS table_name, as_sel...
SELECT database AS table_catalog, database AS table_schema, `table` AS table_name, nam...
SELECT 'def' AS constraint_catalog, database AS constraint_schema, 'PRIMARY' AS constrai...
SELECT '' AS constraint_catalog, NULL AS constraint_name, '' AS constraint_schema, '' AS u...
SELECT name AS catalog_name, name AS schema_name, 'default' AS schema_owner, NULL ...
SELECT '' AS table_catalog, '' AS table_schema, '' AS table_name, 0 AS non_unique, '' AS ind...
SELECT database AS table_catalog, database AS table_schema, name AS table_name, multilf...
SELECT database AS table_catalog, database AS table_schema, name AS table_name, as_sel...
SELECT bt.transaction_id AS transaction_id, bt.transaction_date AS transaction_date, bt.amo...
SELECT database AS table_catalog, database AS table_schema, `table` AS table_name, nam...
SELECT 'def' AS constraint_catalog, database AS constraint_schema, 'PRIMARY' AS constrai...
SELECT '' AS constraint_catalog, NULL AS constraint_name, '' AS constraint_schema, '' AS u...
SELECT name AS catalog_name, name AS schema_name, 'default' AS schema_owner, NULL ...

Materialized views

Arise by INFINITAS

Materialized views are database objects that **contain the results of a pre-computed query**. They are similar to regular views in that they represent a virtual table based on an underlying query. However, materialized **views differ in that they physically store the computed data**, making them suitable for **improving query performance by reducing the need for repetitive calculations**.

1. Prepare Aggreate Table

```
CREATE Table transaction_summary_by_type_day
(
    transaction_day Date,
    transaction_type Enum('Deposit' =1, 'Withdraw' = 2, 'Transfer' = 3, 'Payment' =4),
    min_amount Decimal(18,6),
    max_amount Decimal(18,6),
    avg_amount Decimal(18,6),
    std_dev_amount Decimal(18,6)
)
ENGINE = AggregatingMergeTree
ORDER BY (transaction_type, transaction_day)
```

2. Create Materialized View

```
CREATE MATERIALIZED VIEW mv_transaction_summary_by_type_day
to transaction_summary_by_type_day

AS
SELECT
    toDate(transaction_date) AS transaction_day,
    transaction_type,
    min(amount) AS min_amount,
    max(amount) AS max_amount,
    avg(amount) AS avg_amount,
    stddevPop(amount) AS std_dev_amount
FROM t_transaction_history
GROUP BY transaction_day, transaction_type;
```

3. Dump First Snapshot Data

```
insert into transaction_summary_by_type_day
( transaction_day,
    transaction_type ,
    min_amount ,
    max_amount ,
    avg_amount ,
    std_dev_amount)
SELECT
    toDate(transaction_date) AS transaction_day,
    transaction_type,
    min(amount) AS min_amount,
    max(amount) AS max_amount,
    avg(amount) AS avg_amount,
    stddevPop(amount) AS std_dev_amount
FROM t_transaction_history
GROUP BY transaction_day, transaction_type;
```

4. After Insert data to t_transadtion_history then the materialized view will be refresh.

Aggregating tables

An AggregatingMergeTree is a type of table engine in ClickHouse designed for efficiently storing and querying pre-aggregated data. It is optimized for use cases where you need to perform aggregations on large volumes of data frequently, such as in analytics and reporting scenarios.

1. Prepare Aggreate Table

```
CREATE Table transaction_summary_by_type_month
(
    transaction_month Date,
    transaction_type Enum('Deposit' =1, 'Withdraw' = 2, 'Transfer' = 3, 'Payment' =4),
    min_amount Decimal(18,6),
    max_amount Decimal(18,6),
    avg_amount Decimal(18,6),
    std_dev_amount Decimal(18,6)
)
ENGINE = AggregatingMergeTree
ORDER BY (transaction_type, transaction_month)
```

2. Create Materialized View

```
CREATE MATERIALIZED VIEW mv_transaction_summary_by_type_month
to transaction_summary_by_type_month

AS
SELECT
    toStartOfMonth(transaction_date) AS transaction_month,
    transaction_type,
    min(amount) AS min_amount,
    max(amount) AS max_amount,
    avg(amount) AS avg_amount,
    stddevPop(amount) AS std_dev_amount
FROM t_transaction_history
GROUP BY transaction_month, transaction_type;
```

3. Dump First Snapshot Data

```
insert into transaction_summary_by_type_month
( transaction_month,
    transaction_type ,
    min_amount ,
    max_amount ,
    avg_amount ,
    std_dev_amount
)
select * from (
SELECT
    toStartOfMonth(transaction_date) AS transaction_month,
    transaction_type,
    min(amount) AS min_amount,
    max(amount) AS max_amount,
    avg(amount) AS avg_amount,
    stddevPop(amount) AS std_dev_amount
FROM t_transaction_history
GROUP BY
    transaction_month,
    transaction_type
)a
order by transaction_month
```

4. After Insert data to t_transadtion_history then the materialized view will be refresh.

Custom transformations using functions

User-defined functions (UDF) allow users to extend the behavior of ClickHouse, by creating lambda expressions that can utilize SQL constructs and functions. These functions can then be used like any in-built function in a query.

```
CREATE FUNCTION date_to_month_name AS (input) ->
if(input= 1, 'January',
if(input=2, 'Febuary',
if(input=3, 'March',
if(input=4, 'April',
if(input=5, 'May',
if(input=6, 'June',
if(input=7, 'July',
if(input=8, 'August',
if(input=9, 'September',
if(input=10, 'October',
if(input=11, 'November',
if(input=12, 'December', 'Invalid'
))))))))))));
```

```
select  date_to_month_name(month(transaction_month))
as month_name, *
from mv_transaction_summary_by_type_month
```

month_name	transaction_m...	transaction_type	min_amount	max_amount	avg_amount	std_dev_amount
Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...
August	2013-08-01	Deposit	220	4852	2617.333333	1511.243616
September	2013-09-01	Deposit	132	5044	2763.47826	1594.178454
October	2013-10-01	Deposit	156	5092	2630	1352.439277
November	2013-11-01	Deposit	108	4908	2355.826086	1525.083936
December	2013-12-01	Deposit	100	5052	2459.854545	1545.128549
January	2014-01-01	Deposit	300	4996	2692.711111	1402.165509
Febuary	2014-02-01	Deposit	300	4924	2697.217391	1420.590169
March	2014-03-01	Deposit	228	4796	2261.565217	1184.726709
April	2014-04-01	Deposit	116	5012	2947.489361	1556.724027
May	2014-05-01	Deposit	388	5092	2897.647058	1331.760133
June	2014-06-01	Deposit	132	5028	2711.5	1453.871068
July	2014-07-01	Deposit	276	4908	2601.061224	1395.562594
August	2014-08-01	Deposit	124	4940	2277.488372	1233.347804
September	2014-09-01	Deposit	212	5092	2713.209302	1534.777501
October	2014-10-01	Deposit	148	4964	2170.947368	1403.791762
November	2014-11-01	Deposit	140	5036	2654.448979	1530.457008
December	2014-12-01	Deposit	132	5060	2452.813559	1286.045633
January	2015-01-01	Deposit	204	4852	1997.333333	1373.921926

Custom transformations using Other language

ClickHouse can call any external executable program or script to process data.
The configuration of executable user defined functions can be located in one or more xml-files. The path to the configuration is specified in the user_defined_executable_functions_config parameter.

1. **Prepare test_function.xml in root path of clickhouse.**

```xml
<functions>
    <function>
        <type>executable</type>
        <name>test_function_python</name>
        <return_type>String</return_type>
        <argument>
            <type>UInt64</type>
            <name>value</name>
        </argument>
        <format>TabSeparated</format>
        <command>test_function.py</command>
    </function>
</functions>
```

2. Prepare test_function.py Script in user_scripts path.

```python
#!/opt/homebrew/bin/python3

import sys

if __name__ == '__main__':
    for line in sys.stdin:
        print("Value " + line, end='')
        sys.stdout.flush()
```

3. Set file permission

```
chmod +x test_function.py
```

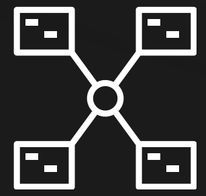
4. Test in sql

```
SELECT test_function_python(toUInt64(2));
```

Summary

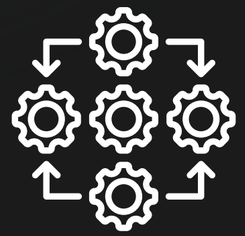
OVERVIEW

- Schema Design
- Integration Methods
- Data Transformation



SCHEMA DESIGN

- Choosing the right data types wtih String, Boolean, Numerics, DateTime and Special Types.
- Designing efficient tables with MergeTree Engine
- Normalization and denormalization for suitable use cases.



INTEGRATION METHODS

- Using the ClickHouse client libraries with Golang Example
- Integrating with admin tools VSCode

DATA TRANSFORMATION

- View
- Materialized views
- Aggregating tables
- Custom transformations using lamda expression
- Custom transformations using Python language

ClickHouse

Arise by INFINITAS

Module 4

ClickHouse
Performance Tuning

MODULE 4 : CLICKHOUSE PERFORMANCE TUNING

4.1 Indexing Strategies
- Key indexing strategies
- Indexing best practices

4.2 Configuration Optimization
- Memory settings
- Disk settings
- Query parallelism

4.3 Monitoring and Diagnostics
- System tables
- Logs
- Profiling
- Using external monitoring tools

Key indexing strategies

Clickhouse employs several indexing strategies to optimize query performance. Here are some of the key indexing strategies used in Clickhouse:

Primary and Sparse Key Index

Sparse indexing is storing the rows for a part on disk ordered by the primary key column(s).

Data Skipping Indexes

Read data only in granules that store values in the searched range.

MinMax Indexes

Type of data skipping index that stores the minimum and maximum values for each column in a data block

Set Indexes

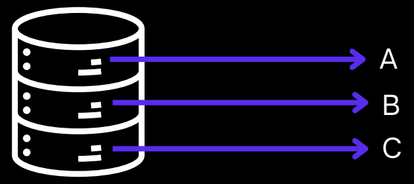
Used to optimize queries involving IN operators or constant value lookups.

Bloom Fliter Indexes

To optimize queries that involve checking for the existence of values in **a set or column**. It is particularly useful for optimizing queries with **IN or NOT IN** operators, as well as for enhancing the performance of semi-join operations.

Partition Key

Data is physically stored and organized based on the partition key values, which can significantly improve query performance for queries



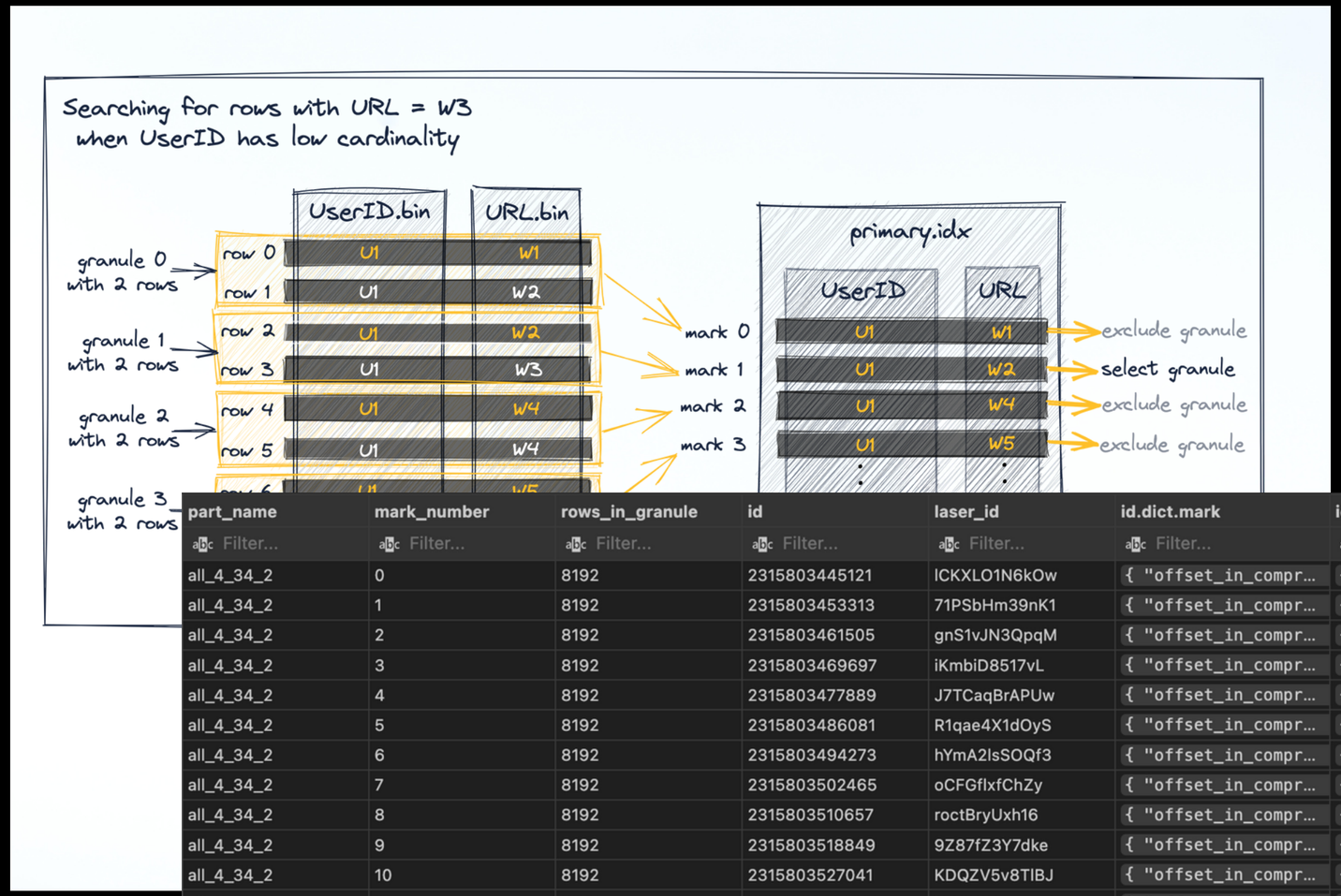
Sorting Key

determines the order in which data is physically stored on disk.

Indexing Strategies

Primary Keys are Sparse Index

Sparse indexing is storing the rows for a part on disk ordered by the primary key column(s).



- Primary Keys allow 2 columns
- Chunk *n* rows to granule
- Keep Min Value in granule
- Append value to granule in parallel process

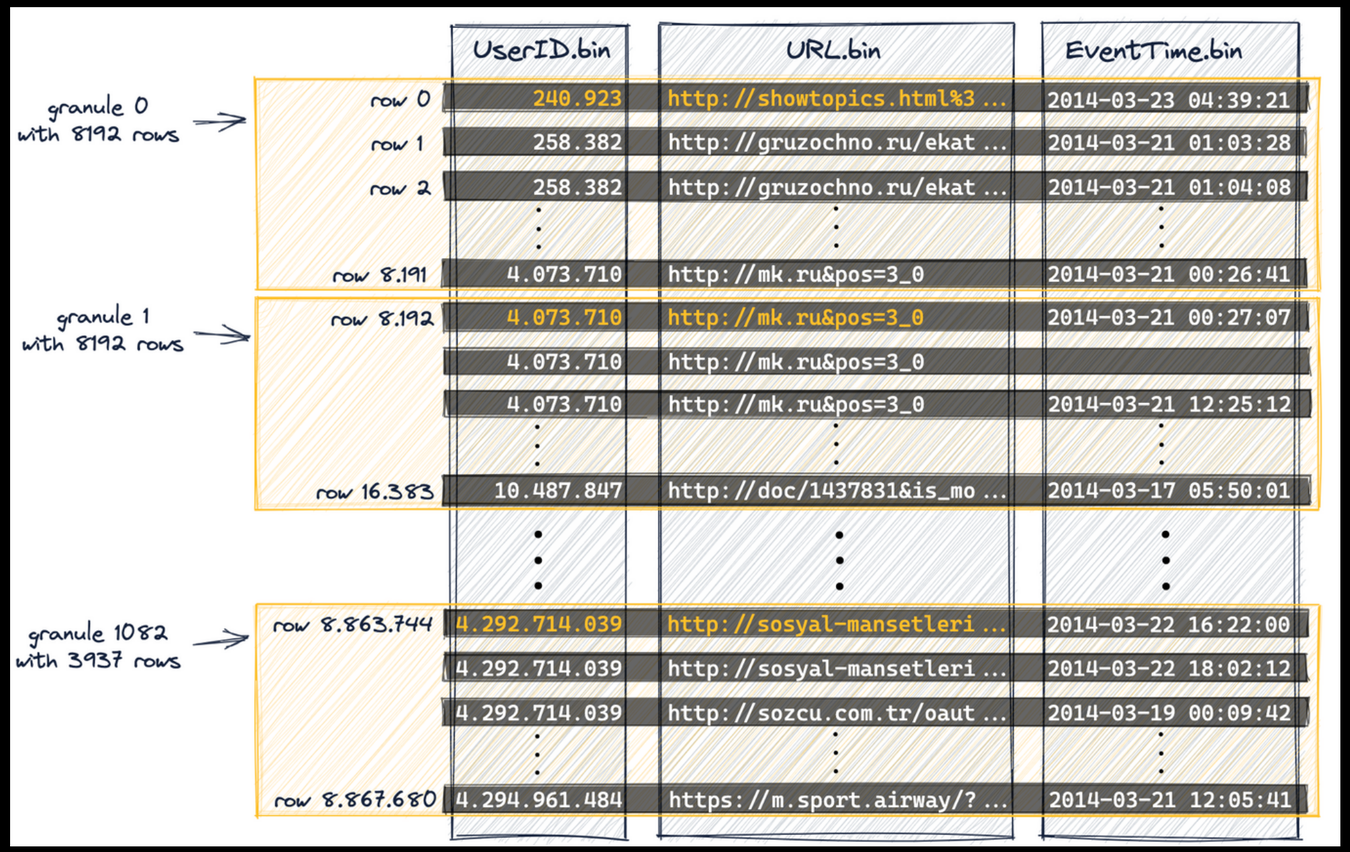
```
SELECT * FROM mergeTreeIndex(currentDatabase(), people, with_marks = true);
```

part_name	mark_number	rows_in_granule	id	laser_id	id.dict.mark	id.mark	first_name.mark
all_4_34_2	0	8192	2315803445121	lCKXLO1N6kOw	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "0", "offset_in_d
all_4_34_2	1	8192	2315803453313	71PSbHm39nK1	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "0", "offset_in_d
all_4_34_2	2	8192	2315803461505	gnS1vJN3QpqM	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "85016", "offset_
all_4_34_2	3	8192	2315803469697	iKmbiD8517vL	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "85016", "offset_
all_4_34_2	4	8192	2315803477889	J7TCaqBrAPUw	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "169773", "offset
all_4_34_2	5	8192	2315803486081	R1qae4X1dOyS	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "169773", "offset
all_4_34_2	6	8192	2315803494273	hYmA2lsSOQf3	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "254802", "offset
all_4_34_2	7	8192	2315803502465	oCFGflxfChZy	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "254802", "offset
all_4_34_2	8	8192	2315803510657	roctBryUxh16	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "339767", "offset
all_4_34_2	9	8192	2315803518849	9Z87fZ3Y7dke	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "339767", "offset
all_4_34_2	10	8192	2315803527041	KDQZV5v8TlBJ	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "424413", "offset
all_4_34_2	11	8192	2315803535233	bF8AGOevupm0	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "424413", "offset
all_4_34_2	12	8192	2315803543425	rl9uDyPvwZAz	{ "offset_in_compr…	{ "offset_in_c…	{ "offset_in_compressed_file": "509197", "offset

Primary Keys are Sparse Index

Sparse indexing is storing the rows for a part on disk ordered by the primary key column(s).

```
CREATE TABLE hits_UserID_URL
(
    `UserID` UInt32,
    `URL` String,
    `EventTime` DateTime
)
ENGINE = MergeTree
PRIMARY KEY (UserID, URL)
ORDER BY (UserID, URL, EventTime)
SETTINGS index_granularity = 8192, index_granularity_bytes = 0;
```



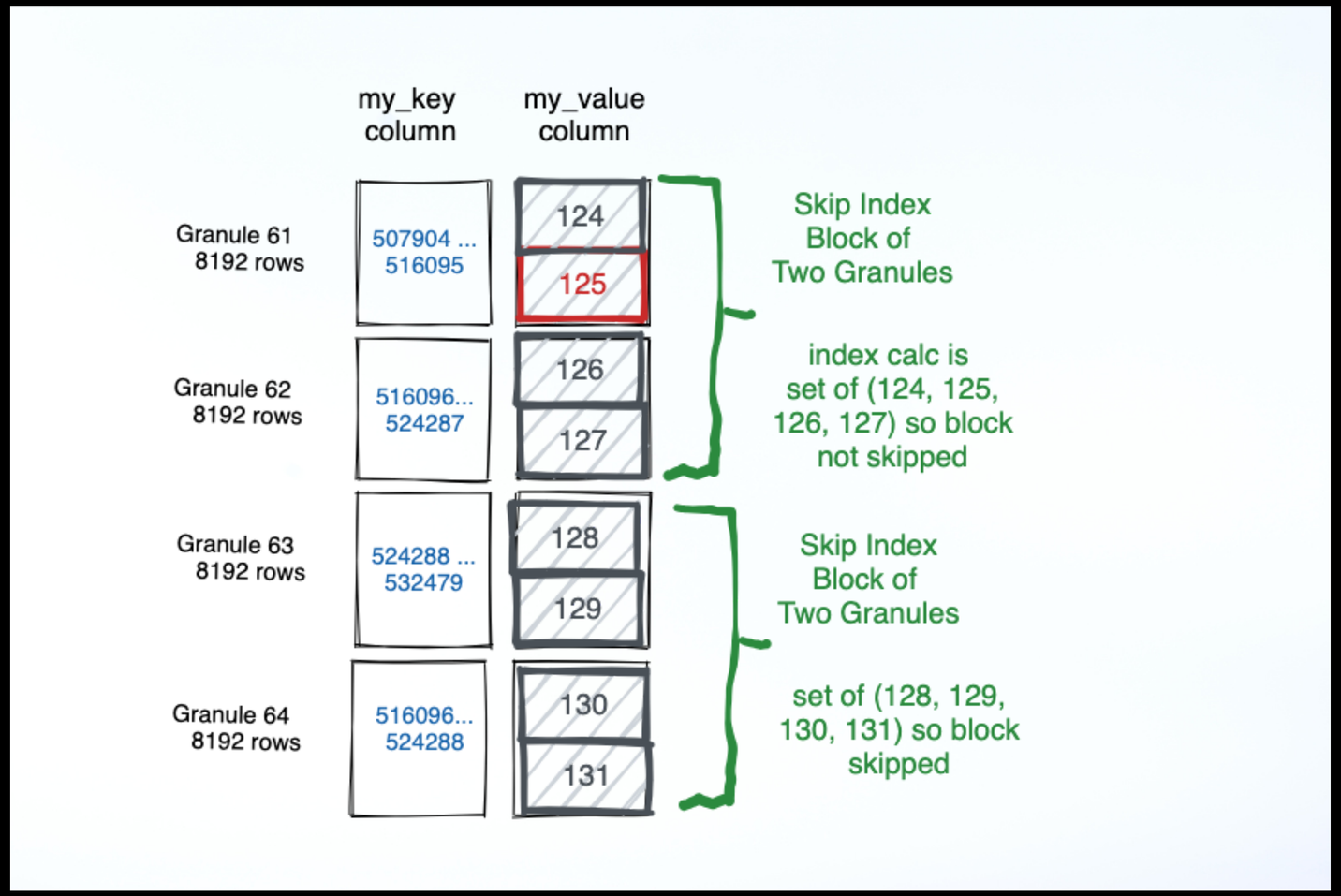
Primary and Sparse Key Index

- ClickHouse is designed to handle massive data volumes and high ingestion rates, making traditional **B-Tree indexing inefficient** due to overhead and rebalancing costs.
- **Instead of indexing every row**, ClickHouse uses a sparse primary index that has one index entry **(mark) per group of rows (granule)**.
- **Data is stored on disk ordered by the primary key columns**, allowing the **sparse index to quickly identify groups** of potentially matching rows via binary search.
- The sparse index **can fit entirely in memory** while still providing significant query performance benefits, especially for analytical range queries.
- **Instead of locating single rows like a B-Tree,** ClickHouse **streams the identified granules** of potentially **matching rows in parallel** for further processing.
- This approach trades off point lookup performance for better ingestion rates, **lower overhead, and improved performance for typical analytical range queries over large datasets.**

Indexing Strategies

Data Skipping Indexes

Read data only in granules that store values in the searched range.



- Type
  - minmax
  - set
  - Bloom Filter

Suitable

- Range values such as DateTime, id
- Avoid use in more same value across multiple granules.

```
CREATE TABLE IF NOT EXISTS sample_table
(
    column1 Int32,
    column2 String,
    column3 Float64
) ENGINE = MergeTree
ORDER BY column1;

-- Create a Data Skipping Index on column1
CREATE INDEX idx_column1_skip
    ON sample_table(column1)
    TYPE minmax
    GRANULARITY 1;

insert into sample_table
values
(1, 'Hello 1', 99), (2, 'Hello 2',  5000),(3, 'Hello 3', 100), (4, 'Hello 4',  5009)
```

part_name	mark_number	rows_in_granule	column1
Filter...	Filter...	Filter...	Filter...
all_1_1_0	0	2	1
all_1_1_0	1	0	2
all_2_2_0	0	2	3
all_2_2_0	1	0	4

MinMax Indexes

This lightweight index type requires no parameters. It stores the minimum and maximum values of the index expression for each block (if the expression is a tuple, it separately stores the values for each member of the element of the tuple). This type is ideal for columns that tend to be loosely sorted by value. This index type is usually the least expensive to apply during query processing.

```
CREATE TABLE orders (
    order_id UInt64,
    customer_id UInt32,
    order_date Date,
    total_amount Decimal(10, 2)
) ENGINE = MergeTree()
ORDER BY (order_date, order_id);
```

```
INSERT INTO orders (order_id, customer_id, order_date, total_amount)
SELECT
    number + 1 AS order_id,
    rand64() % 1000 AS customer_id,
    today() - rand() % 365 AS order_date,
    rand64() % 100000 / 100.0 AS total_amount
FROM numbers(10000000);
```

```
SELECT count(*)
FROM orders
WHERE total_amount BETWEEN 100  AND 500;
```

```
ALTER TABLE orders ADD INDEX idx_total_amount total_amount type MINMAX GRANULARITY 8192;
```

Set Indexes

In ClickHouse, a Set index is a type of sparse index that is particularly useful for optimizing queries involving the **IN or NOT IN** operators, as well as for optimizing certain types of semi-join operations.

```
CREATE TABLE products (
    product_id UInt32,
    category_id UInt16,
    name String,
    price Decimal(10, 2)
) ENGINE = MergeTree()
ORDER BY (product_id, category_id);
```

```
INSERT INTO products (product_id, category_id, name, price)
SELECT
    number + 1 AS product_id,
    rand64() % 20 + 1 AS category_id,
    concat('Product ', toString(number + 1)) AS name,
    rand64() % 10000 / 100.0 AS price
FROM numbers(1000000);
```

```
select count(*) from products
where category_id in (5,6)
```

```
ALTER TABLE products ADD INDEX idx_category_id category_id type SET(20) GRANULARITY 8192;
```

Bloom Fliter Indexes

In ClickHouse, a set index is a special type of index used to efficiently filter data based on a set of discrete values. Here's an example of how to create a set index in ClickHouse:

```
CREATE TABLE users (
    user_id UInt64,
    email String,
    city String
) ENGINE = MergeTree()
ORDER BY user_id;
```

```
INSERT INTO users (user_id, email, city)
SELECT
    number + 1 AS user_id,
    concat('user', toString(number + 1), '@example.com') AS email,
    arrayElement(
        ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix', 'Philadelphia', 'San Antonio',
'San Diego', 'Dallas', 'San Jose'],
        rand64() % 10
    ) AS city
FROM numbers(1000000);
```

```
SELECT count(*)
FROM users
WHERE city IN ('New York', 'Los Angeles', 'Chicago');
```

```
ALTER TABLE users ADD INDEX idx_city_bloom_filter_index city TYPE Bloom_Filter GRANULARITY 8192;
```

```
ALTER TABLE users
MATERIALIZE INDEX idx_city_bloom_filter_index
```

Indexing Strategies

Partition Key

In ClickHouse, partition keys are used to **physically partition data** within a table based on one or more columns. Partitioning is a powerful technique that can significantly **improve query performance**, especially for queries that involve **filtering, aggregating, or joining data based on the partition key columns**.

```
CREATE TABLE events (
    event_id UInt64,
    event_type String,
    event_time DateTime,
    user_id UInt32
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(event_time)
ORDER BY (event_time, event_id);
```

```
INSERT INTO events (event_id, event_type, event_time, user_id)
SELECT
    number + 1 AS event_id,
    arrayElement(['purchase', 'view', 'click', 'search'],
        rand64() % 4) AS event_type,
    today() - rand() % 365 AS event_time,
    rand64() % 10000 AS user_id
FROM numbers(1000000);
```

```
SELECT
    partition,
    name,
    rows,
    active
FROM system.parts
WHERE table = 'events'
```

partition	name	rows	active
Filter...	Filter...	Filter...	Filter...
202303	202303_6_6_0	21840	1
202304	202304_7_7_0	82186	1
202305	202305_5_5_0	84849	1
202306	202306_13_13_0	82257	1
202307	202307_4_4_0	85000	1
202308	202308_9_9_0	85203	1
202309	202309_3_3_0	82528	1
202310	202310_11_11_0	85106	1
202311	202311_2_2_0	82156	1
202312	202312_1_1_0	85280	1
202401	202401_10_10_0	84337	1
202402	202402_8_8_0	79231	1
202403	202403_12_12_0	60027	1

Indexing Strategies

Sorting Key

In ClickHouse, a sorting key is used to specify the order in which **data is physically sorted** and stored on disk within a table. The sorting key is defined in addition to the **primary key and can significantly improve query performance** for specific types of queries that involve sorting or range scans on the sorting key columns.

```
CREATE TABLE orders (
    order_id UInt64,
    customer_id UInt32,
    order_date Date,
    total_amount Decimal(10, 2)
) ENGINE = MergeTree()
ORDER BY (order_date, order_id);
```

```
INSERT INTO orders (order_id, customer_id, order_date, total_amount)
SELECT
    number + 1 AS order_id,
    rand64() % 1000 AS customer_id,
    today() - rand() % 365 AS order_date,
    rand64() % 100000 / 100.0 AS total_amount
FROM numbers(10000000);
```

```
SELECT
    partition,
    partition_id,
    name,
    rows,
    path,
    active
FROM system.parts
WHERE table = 'orders'
```

partition	partition_id	name	rows
abc Filter...	abc Filter...	abc Filter...	abc Filter...
tuple()	all	all_1_6_1	6290694
tuple()	all	all_7_7_0	1048449
tuple()	all	all_8_8_0	1048449
tuple()	all	all_9_9_0	1048449
tuple()	all	all_10_10_0	563959

Indexing Best Practices

Index Selection

Arise by INFINITAS

Primary Key

Drop Unused Indexes

Sparse Index

Denormalization

Partition Key

Test and Benchmark

Sorting Key

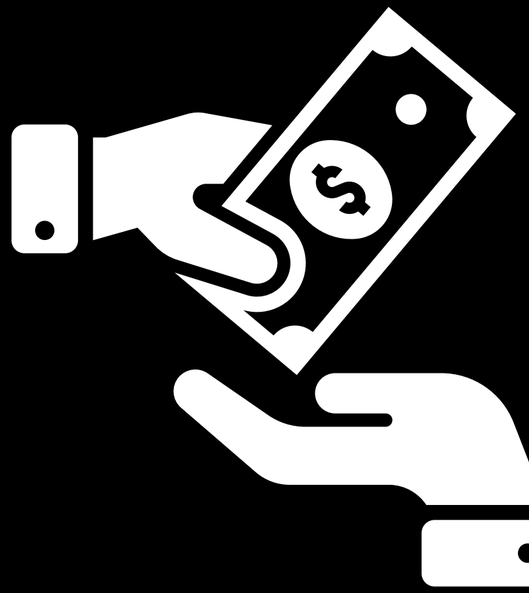
Avoid Indexing Low-Cardinality Columns

Primary Key

Choose an appropriate primary key for your table. The primary key should be a column or a combination of columns **that uniquely identifies each row and is frequently used for filtering or sorting**. A well-chosen primary key can significantly improve query performance.

Master Data : ID, No, Code, Name
Time Series : Date => OrderDate, EventDate

Student
- **ID**
- **No**



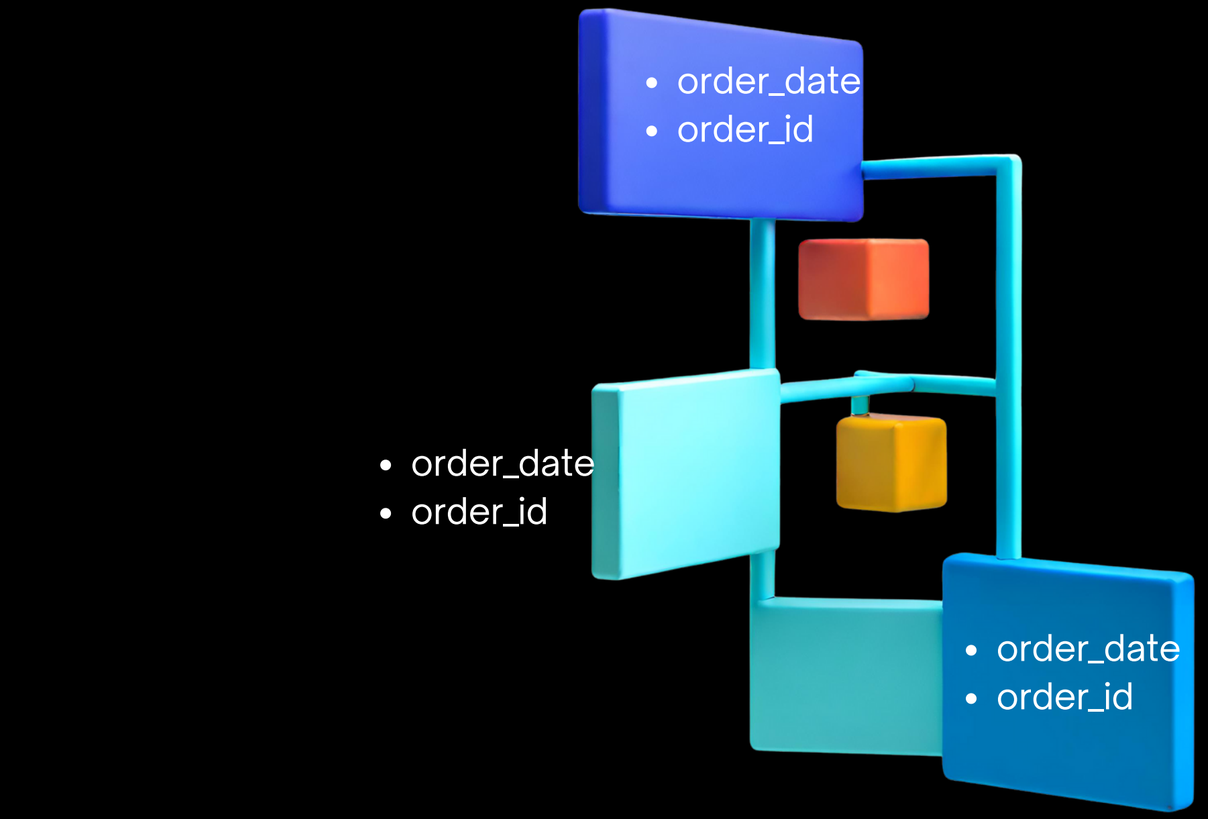
Payment
- **PaymentDate**
- **PayerID**

* ClickHouse does not inherently prevent duplicate data. Instead, it relies on the design of the primary key, which should be suitable for the table engine being used. By employing the 'FINAL' keyword, ClickHouse selects the last state of the primary key, effectively avoiding duplicates.

Sparse Indexes

Create sparse indexes on columns that are frequently used in WHERE, **ORDER BY, or JOIN clauses.** Sparse indexes can greatly improve the performance of queries involving those columns.

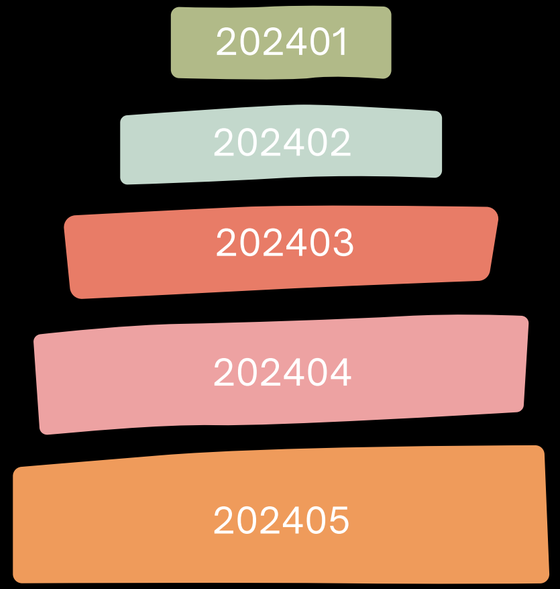
```
CREATE TABLE orders (
    order_id UInt64,
    customer_id UInt32,
    order_date Date,
    total_amount Decimal(10, 2)
) ENGINE = MergeTree()
ORDER BY (order_date, order_id);
```



Partition Key Selection

The natural time-based or range-based partitioning, consider using the appropriate column(s) as the partition key. This can significantly improve query performance for queries that filter or aggregate data based on the partition key.

```
CREATE TABLE events (
    event_id UInt64,
    event_type String,
    event_time DateTime,
    user_id UInt32
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(event_time)
ORDER BY (event_time, event_id);
```

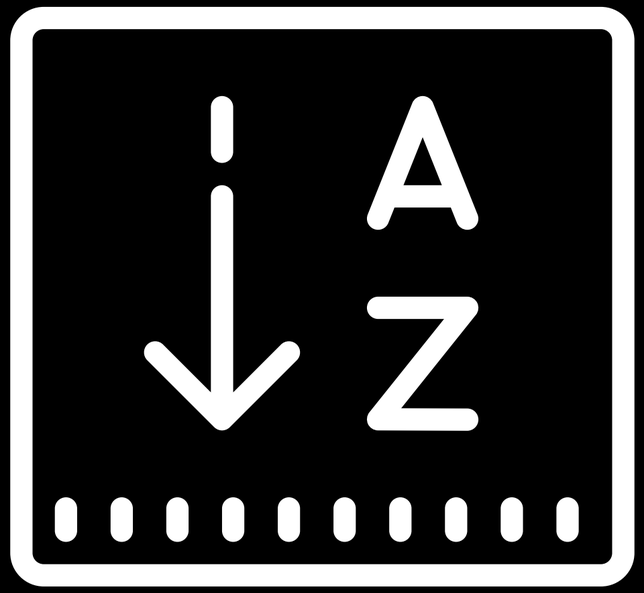


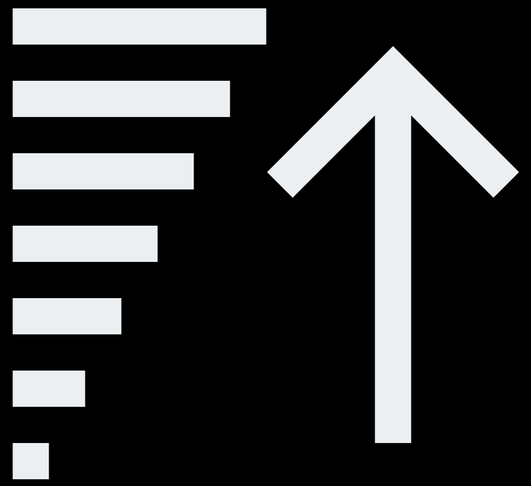
Sorting Key Selection

Queries frequently sort or scan data based on specific columns, consider using those columns as the sorting key. This can improve performance for those queries by taking advantage of the physical data ordering on disk.

Master Data : ID, No, Code, Name
Time Series : Date => OrderDate, EventDate
Value => Nummeric

- For clickhouse, the Sort Key must be created in the same order as the Primary Key, but can have more columns than the Primary Key.





Avoid Indexing Low-Cardinality Columns

void creating indexes on columns with low cardinality (few distinct values), as these indexes may not provide significant performance benefits and can consume additional storage space.

Monitor Index Usage

Use the system.query_log table or other monitoring tools to identify queries that are not using indexes effectively. This can help you identify opportunities for creating new indexes or modifying existing ones.

```
select
    initial_query_id,type,
    event_date,event_time,query_kind,
    query_duration_ms,result_rows,
    result_bytes, memory_usage, query
    from system.query_log
```

initial_query_id	type	event_date	event_time	query_kind	query_duration...	result_rows	result_bytes	memory_usage	query
Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...	Filter...
4d47f703-5de9-4f7e-...	QueryStart	2024-03-19	2024-03-19 14:31:37	Select	0	0	0	0	SELECT d.na
4d47f703-5de9-4f7e-...	QueryFinish	2024-03-19	2024-03-19 14:31:37	Select	1	4	9394	4189514	SELECT d.na
43d10dac-ac55-4a89...	QueryStart	2024-03-19	2024-03-19 14:31:38	Select	0	0	0	0	SELECT d.na
43d10dac-ac55-4a89...	QueryFinish	2024-03-19	2024-03-19 14:31:38	Select	0	4	9394	4189514	SELECT d.na
f5fe8811-5c66-4830-...	QueryStart	2024-03-19	2024-03-19 14:31:39	Select	0	0	0	0	SELECT d.na
f5fe8811-5c66-4830-...	QueryFinish	2024-03-19	2024-03-19 14:31:39	Select	0	4	9394	4189514	SELECT d.na
16f76b11-7d44-4ce8-...	QueryStart	2024-03-19	2024-03-19 14:31:41	Select	0	0	0	0	SELECT d.na
16f76b11-7d44-4ce8-...	QueryFinish	2024-03-19	2024-03-19 14:31:41	Select	0	4	9394	4189514	SELECT d.na
d585106f-94cd-4044...	QueryStart	2024-03-19	2024-03-19 14:31:42	Select	0	0	0	0	SELECT d.na
d585106f-94cd-4044...	QueryFinish	2024-03-19	2024-03-19 14:31:42	Select	0	4	9394	4189514	SELECT d.na
10d3271d-e742-4c01-...	QueryStart	2024-03-19	2024-03-19 14:32:57	Show	0	0	0	0	show databa
10d3271d-e742-4c01-...	QueryFinish	2024-03-19	2024-03-19 14:32:57	Show	1	4	4352	4190204	show databa
f4e3ca2b-ef95-40dc-...	QueryStart	2024-03-19	2024-03-19 14:33:11	Create	0	0	0	0	CREATE TAB
f4e3ca2b-ef95-40dc-...	QueryFinish	2024-03-19	2024-03-19 14:33:11	Create	4	0	0	0	CREATE TAB
7721d701-a1ad-438e-...	QueryStart	2024-03-19	2024-03-19 14:34:16	Create	0	0	0	0	CREATE DAT
7721d701-a1ad-438e-...	QueryFinish	2024-03-19	2024-03-19 14:34:16	Create	2	0	0	0	CREATE DAT
3813324b-f85c-4090...	QueryStart	2024-03-19	2024-03-19 14:34:19	Create	0	0	0	0	CREATE TAB
3813324b-f85c-4090...	QueryFinish	2024-03-19	2024-03-19 14:34:19	Create	5	0	0	0	CREATE TAB
dd7bc8dd-9bc0-4589...	QueryStart	2024-03-19	2024-03-19 14:37:45	Insert	0	0	0	0	INSERT INTC
dd7bc8dd-9bc0-4589...	QueryFinish	2024-03-19	2024-03-19 14:37:45	Insert	358	1000000	57821893	393101972	INSERT INTC
077e1446-cb4d-4c1b...	QueryStart	2024-03-19	2024-03-19 14:38:51	Select	0	0	0	0	SELECT d.na

Indexing Best Practices

Drop Unused Indexes

Periodically review and drop indexes that are not being used by your queries. Unused indexes can consume storage space and potentially slow down data ingestion and updates.

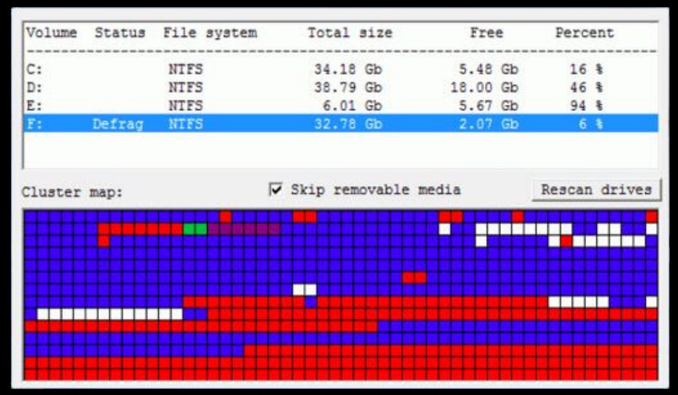
```
SHOW INDEXES from users
```

```
drop index idx_city_bloom_filter_index  on users
```

Index Maintenance:

Frequently update or modify data in your tables, consider the potential impact on index maintenance. Clickhouse **automatically maintains indexes**, but frequent updates can lead to index fragmentation and degraded performance over time.

```
OPTIMIZE TABLE events;
```



Denormalization

In some cases, denormalizing data and duplicating columns can be more efficient than relying on joins and secondary indexes, especially for analytical workloads.

Join

```
select * from (
SELECT
    toYear(s.sale_date) as by_year,
    p.product_name as product_name,
    c.category_name as category_name,
    sum(s.quantity) as quantity,
    sum(s.price) as total_price,
    sum(s.quantity) * sum(s.price) AS revenue
FROM sales s
JOIN products p ON s.product_id = p.product_id
JOIN categories c ON p.category_id = c.category_id
group by by_year, product_name, category_name
)a
order by by_year desc, revenue desc
```

Not Join

```
SELECT
    toYear(sale_date) as by_year,
    product_name ,
    category_name,
sum(quantity) as total_quantity,
    sum(price) as total_price,
    sum(quantity) * sum(price) AS revenue
FROM sales_analytics
group by by_year, product_name, category_name
```

Test and Benchmark

Before implementing indexing strategies, test and benchmark your queries with and without indexes to understand the performance impact. Indexing strategies can improve query performance, but they also have overhead and storage costs.

No Index

```
SELECT * FROM skip_table WHERE my_value IN (125, 700)

┌─my_key─┬─my_value─┐
│ 512000 │      125 │
│ 512001 │      125 │
│    ... │      ... │
└────────┴──────────┘

8192 rows in set. Elapsed: 0.079 sec. Processed 100.00 million rows, 800.10 MB (1.26 billion rows/s., 10.10 GB/s.
```

With Index

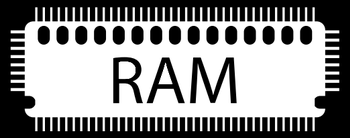
```
SELECT * FROM skip_table WHERE my_value IN (125, 700)

┌─my_key─┬─my_value─┐
│ 512000 │      125 │
│ 512001 │      125 │
│    ... │      ... │
└────────┴──────────┘

8192 rows in set. Elapsed: 0.051 sec. Processed 32.77 thousand rows, 360.45 KB (643.75 thousand rows/s., 7.08 MB/s.)
```

Opinion from proof: From testing with hundreds of millions of rows of data, having an index or not has not seen much difference in query results because most of the main keys searched from the Primary Key, Sorting Key, and Partition Key have already been designed for good performance. Reading from an Index uses similar principles. It can be said that all three preparations are for Tuning Performance.

Configuration Optimization

Arise by INFINITAS

Memory settings



In ClickHouse, memory settings play a crucial role in optimizing query performance and resource utilization. ClickHouse is designed to efficiently utilize available memory to achieve high-performance data processing.

Disk settings

In ClickHouse, you can configure various aspects related to disk storage and management through the config.xml file and other configuration options.

Query parallelism

In ClickHouse, you can configure various settings related to query parallelism to control how queries are executed in parallel and manage the utilization of system resources.

Configuration Optimization

Memory settings

max_memory_usage

- Memory for Query
- Default is 0 “Unlimited”
- When the memory usage exceeds this limit, ClickHouse will start using temporary files on disk

max_bytes_before_external_group_by

- Memory for Group by
- After size of the data exceeds this limit will use temporary files on disk instead

max_bytes_before_external_sort

- Memory for Sorting
- After size of the data exceeds this limit will use temporary files on disk instead

join_algorithm

- Specify Join Algorithm
- default algorithm is hash for small to medium-sized
- partial_merge or prefer_partial_merge for larger datasets

max_memory_usage_for_all_queries

- Maximum memory usage for all queries running concurrently

max_threads

- Maximum number of threads that ClickHouse can use for query execution
- Default is 12

Configuration Optimization

Disk settings

In ClickHouse, you can configure various aspects related to disk storage and management through the config.xml file and other configuration options. Here are some important disk configuration settings and options:

Disk Definitions

- <path>
- <keep_free_space_bytes>
- <max_data_part_size_bytes>

Storage Policies

- <volumes>
- <move_factor>

Disk Caching

- uncompressed_cache_size
- custom_cached_disks_base_directory>
- mark_cache_size
- index_mark_cache_size
- mmap_cache_size
- role_cache_expiration_time_seconds

Disk Layout

- <thread_pool>

Merge Tree Settings

- <merge_tree>
- <max_bytes_to_merge_at_min_space_in_pool>

Compression Settings

- <compression>
- <method>

Monitoring and Logging

- <logger>
- <level>
- <log>
- <errorlog>
- <size>
- <count>

Configuration Optimization

Query parallelism

max_parallel_replicas

- Maximum number of replicas that ClickHouse can read data from in parallel when executing distributed queries.

group_by_overflow_mode

- Controls the behavior of the GROUP BY operation when the temporary data exceeds the available memory. It can be set to throw, file, or external_memory to manage the trade-off between memory usage and parallelism.

parallel_view_processing

- Process views in parallel, potentially improving query performance for workloads involving views.

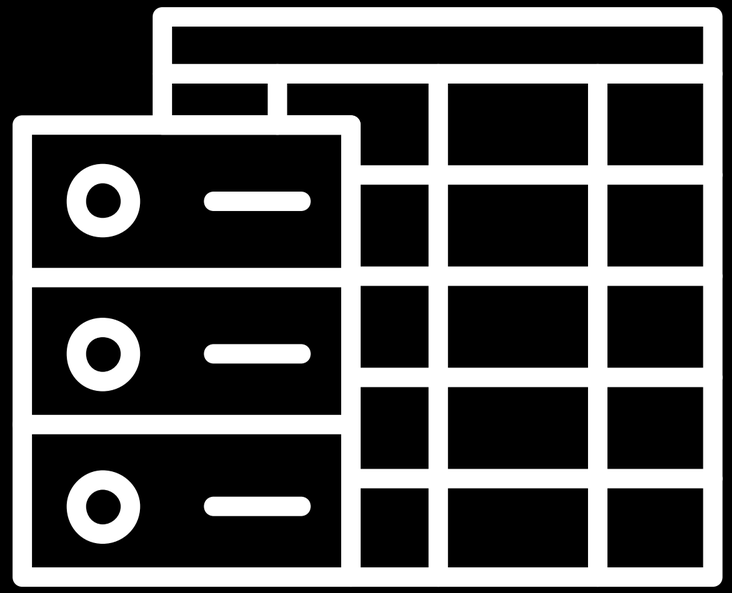
distributed_aggregation_memory_efficient

- Option enables a more memory-efficient distributed aggregation algorithm, which may improve query performance for certain workloads

Monitoring and Diagnostics

ClickHouse provides various monitoring and diagnostic tools to help you understand the performance, resource utilization, and potential issues within your ClickHouse cluster. Effective monitoring and diagnostics are crucial for maintaining optimal performance, identifying bottlenecks, and troubleshooting problems. Here are some key monitoring and diagnostic features in ClickHouse:

- System Tables
- Logs
- Profiling
- Monitoring Tools Integration



System tables

ClickHouse provides a rich set of system tables that expose various metrics and information about the system's internal state. Provide valuable insights into CPU, memory, disk, network, and query execution metrics.

- system.metrics
- system.events
- system.asynchronous_metrics
- system.processes

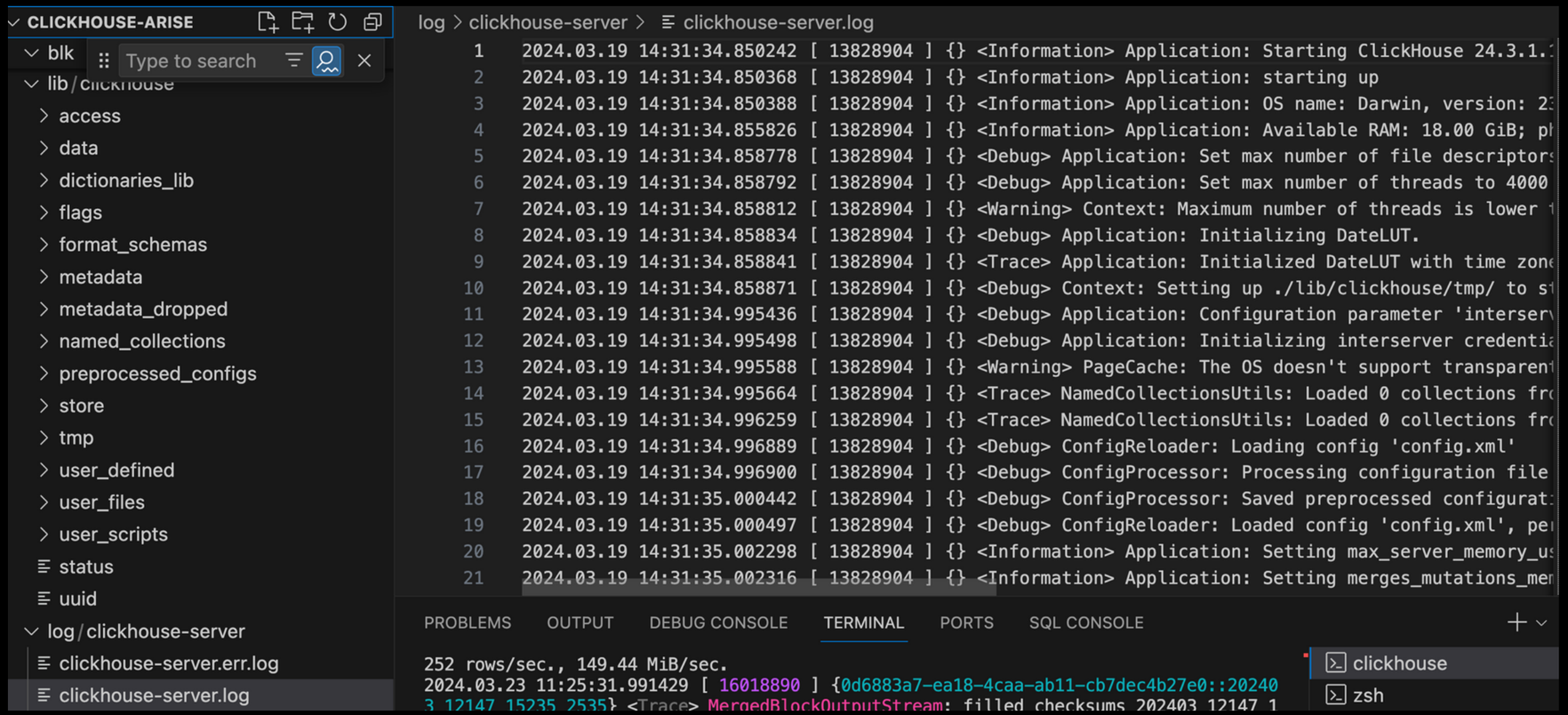
metric	value	description
Filter...	Filter...	Filter...
Query	1	Number of executing queries
Merge	0	Number of executing background merges
Move	0	Number of currently executing moves
PartMutation	0	Number of mutations (ALTER DELETE/UPDATE)
ReplicatedFetch	0	Number of data parts being fetched from replica
ReplicatedSend	0	Number of data parts being sent to replicas
ReplicatedChecks	0	Number of data parts checking for consistency
BackgroundMergesA...	0	Number of active merges and mutations in an associa...
BackgroundMergesA...	64	Limit on number of active merges and mutations in an...
BackgroundFetches...	0	Number of active fetches in an associated backgroun...
BackgroundFetches...	32	Limit on number of simultaneous fetches in an associ...
BackgroundCommon...	0	Number of active tasks in an associated background ...
BackgroundCommon...	16	Limit on number of tasks in an associated backgroun...
BackgroundMovePo...	0	Number of active tasks in BackgroundProcessingPool...
BackgroundMovePo...	16	Limit on number of tasks in BackgroundProcessingPo...

event	value	description
Filter...	Filter...	Filter...
Query	15	Number of queries to be interpreted and potentially executed. Does not include queries that f...
SelectQuery	15	Same as Query, but only for SELECT queries.
InitialQuery	15	Same as Query, but only counts initial queries (see is_initial_query).
QueriesWithSubqueries	15	Count queries with all subqueries
SelectQueriesWithSubqueries	15	Count SELECT queries with all subqueries
FailedQuery	1	Number of failed queries.
QueryTimeMicroseconds	55179	Total time of all queries.
SelectQueryTimeMicroseconds	55179	Total time of SELECT queries.
FileOpen	55876	Number of files opened.
ReadBufferFromFileDescripto...	308235	Number of reads (read/pread) from a file descriptor. Does not include sockets.
ReadBufferFromFileDescripto...	646722172	Number of bytes read from file descriptors. If the file is compressed, this will show the compr...
WriteBufferFromFileDescript...	61970	Number of writes (write/pwrite) to a file descriptor. Does not include sockets.
WriteBufferFromFileDescript...	744997110	Number of bytes written to file descriptors. If the file is compressed, this will show compresse...
ReadCompressedBytes	635517764	Number of bytes (the number of bytes before decompression) read from compressed sources...
CompressedReadBufferBlocks	1203355	Number of compressed blocks (the blocks of data that are compressed independent of each ...
CompressedReadBufferBytes	18370687907	Number of uncompressed bytes (the number of bytes after decompression) read from compr...
OpenedFileCacheMisses	18499	Number of times a file has been found in the opened file cache, so we had to open it again.
OpenedFileCacheMicrosecon...	34474	Amount of time spent executing OpenedFileCache methods.
IOBufferAllocs	145690	Number of allocations of IO buffers (for ReadBuffer/WriteBuffer).
IOBufferAllocBytes	28677482163	Number of bytes allocated for IO buffers (for ReadBuffer/WriteBuffer).
ArenaAllocChunks	5	Number of chunks allocated for memory Arena (used for GROUP BY and similar operations)

Logs

ClickHouse generates various log files that can be used for diagnostics and troubleshooting.

- clickhouse-server.log
- system.query_log
- system.query_views_log
- system.trace_log



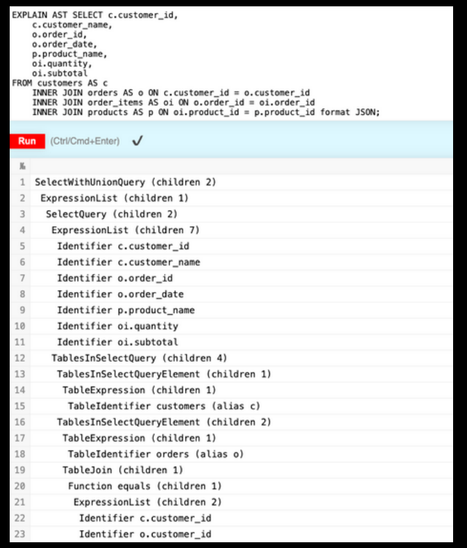
Profiling

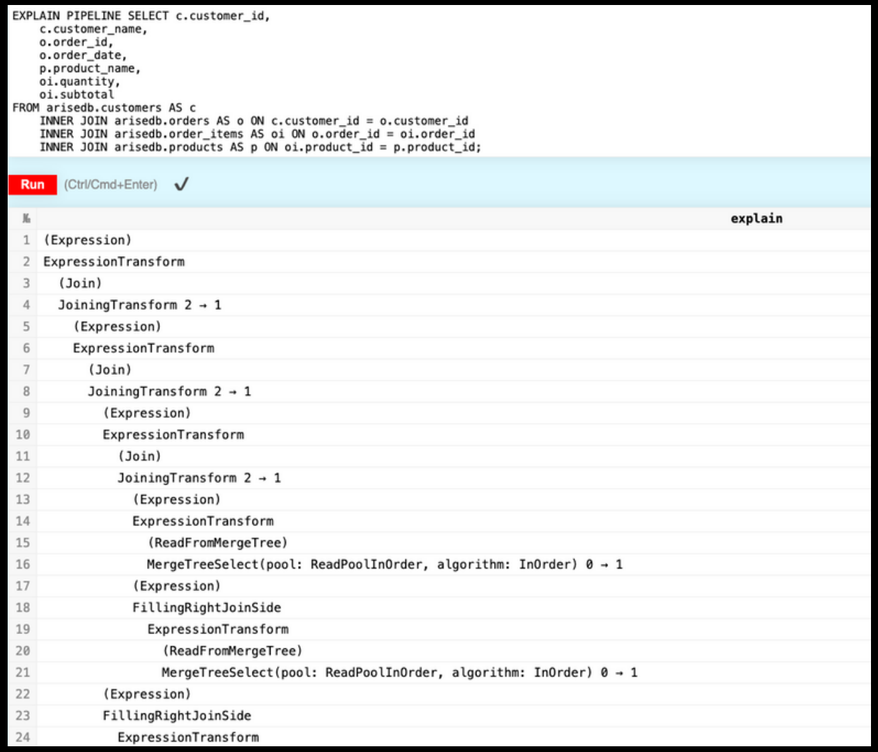
EXPLAIN: Use the EXPLAIN keyword before your query to get an execution plan without actually running the query. This can be useful for understanding how ClickHouse processes your query.

EXPLAIN Types

- AST — Abstract syntax tree.
- SYNTAX — Query text after AST-level optimizations.
- QUERY TREE — Query tree after Query Tree level optimizations.
- PLAN — Query execution plan.
- PIPELINE — Query execution pipeline.

```
EXPLAIN AST  SELECT c.customer_id,
    c.customer_name,
    o.order_id,
    o.order_date,
    p.product_name,
    oi.quantity,
    oi.subtotal
FROM customers AS c
    INNER JOIN orders AS o ON c.customer_id = o.customer_id
    INNER JOIN order_items AS oi ON o.order_id = oi.order_id
    INNER JOIN products AS p ON oi.product_id = p.product_id
    FORMAT JSON;
```





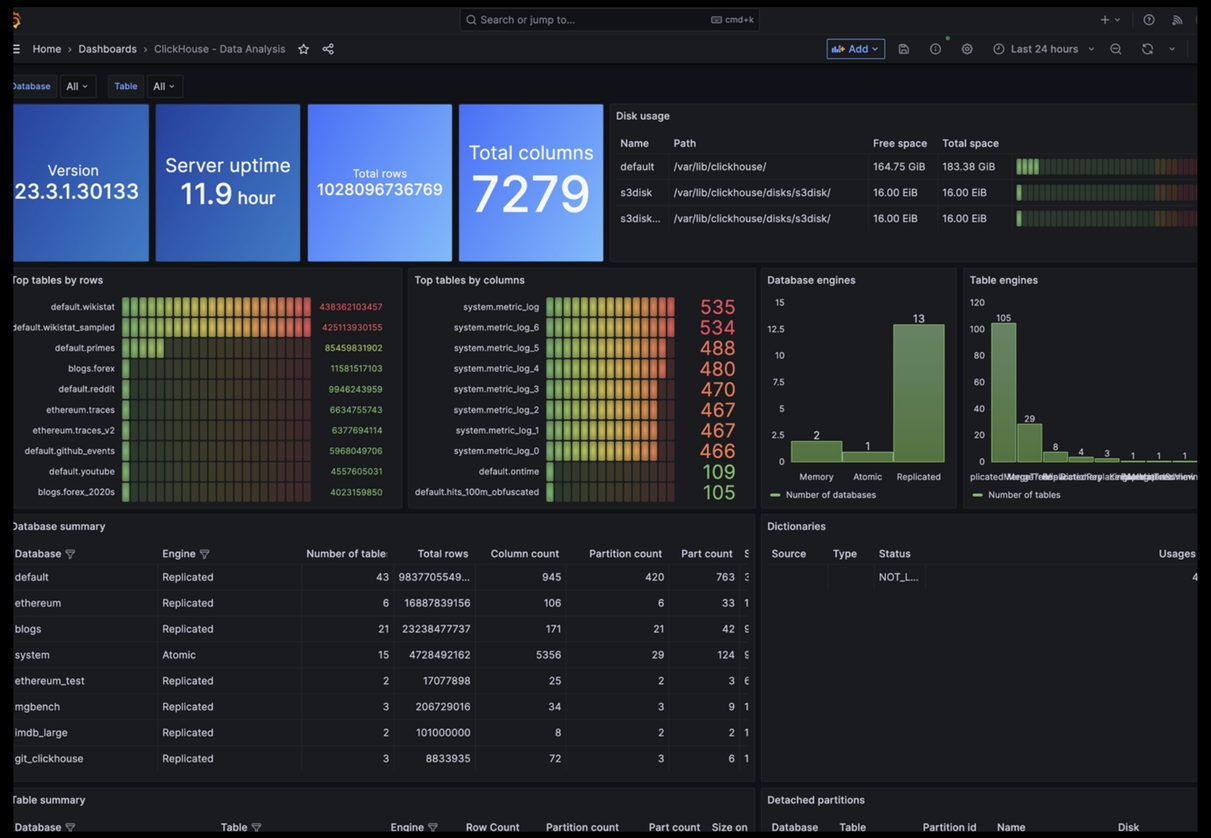
https://clickhouse.com/docs/en/sql-reference/statements/explain

Monitoring and Diagnostics

Using external monitoring tools

Grafana

Arise by INFINITAS



https://clickhouse.com/docs/en/sql-reference/statements/explain

Summary

OVERVIEW

- Indexing Strategies
- Configuration Optimization
- Monitoring and Diagnostics

INDEXING STRATEGIES

- Key indexing strategies such as Primary Key, Data Skipping , MinMax, Set, BloomFilter, Partition Key and Sorting Key.
- Best practices for maintain index.

CONFIGURATION OPTIMIZATION

- Memory settings
- Disk settings
- Query parallelism
- Fine tune of configuration of any products need to test and monitor to benchmark because we don’t understand behaviour.

MONITORING AND DIAGNOSTICS

- System Tables
- Logs
- Profiling
- Monitoring Tools Integration with Grafana