Up and Running with ClickHouse

Learn and Explore ClickHouse, It's Robust Table Engines for Analytical Tasks, ClickHouse SQL, Integration with External Applications, and Managing the ClickHouse Server

VIJAY ANAND R

bpb

Up and Running with ClickHouse

Learn and Explore ClickHouse, It's Robust Table Engines for Analytical Tasks, ClickHouse SQL, Integration with External Applications, and Managing the ClickHouse Server

Vijay Anand R

www.bpbonline.com

FIRST EDITION 2022

ISBN: 978-93-91392-246

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

To View Complete
BPB Publications Catalogue
Scan the QR Code:

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

My beloved family and friends

About the Author

Vijay Anand is a technology professional based out of Chennai who works on solving data engineering and data science problems. He has worked across multiple domains, including sports analytics and telecom. He is currently working as a technical lead for the world's leading manufacturer of construction and mining equipment. He has done his Masters from Vrije Universiteit, Brussel. His areas of interest include database design, building software solutions, machine learning and data science.

About the Reviewer

With over 17 years of software development experience in Microsoft and various Open source Big Data technologies, **Mallanagouda Patil** is a Senior Staff Data Engineer at Netskope. He has in-depth working experience on Apache Hadoop, Apache Kafka, Apache Spark, Apache Pulsar, Clickhouse and Kubernetes technologies. He is also a technical reviewer for BPB Publications. He is passionate about teaching and conducts seminars for students in his free time.

Acknowledgement

There are quite a few people I want to thank for the continued and ongoing support they have given me while writing this book. First and foremost, I would like to thank my managers who gave me an opportunity to evaluate and adopt ClickHouse at work, which in turn provided me the possibility to learn this amazing database management system. Heartfelt thanks to the innumerous developers and contributors of this open source project - ClickHouse.

I would like to thank my family and friends, who seldom complained when I had to sacrifice my time with them, while working on this book. I couldn't have completed this endeavor without their generous support.

My gratitude also goes to Nrip Jain (BPB Publications), who gave me an opportunity to write this book. Special thanks to the team at BPB Publications for being patient and supportive enough right from day one and guiding me through the publishing process. The whole team, including the reviewers were instrumental in getting the book in much better state than I ever could have managed. Above all, I thank the almighty for all the good things and the opportunities in my life.

Preface

This book is about ClickHouse, a relatively new database management system that was open-sourced in 2016 (Apache License 2.0) . Despite being a new entrant, it is gaining lot of popularity and large scale adoptions have become common in the industry. ClickHouse was initially developed by Yandex and now being actively developed by ClickHouse, Inc and it is primarily focused on speed, reliability and ease of use. The readers of this book are required to have basic programming, Linux and database knowledge but learning ClickHouse with this book can be from the scratch with little extra effort.

This book takes learn-by-doing approach. This book is aimed at being a practical guide for ClickHouse. The best way to get the maximum benefit out of this book is by working out all the examples by yourself right from setting up the environment, installing ClickHouse, setting up sample databases, querying the database with SQL examples, creating and working with different table engines and knowing the difference, installing and integrating with third party tools and frameworks and performing the administrative tasks in the sandbox. Special care is taken to ensure simple and easy-to-understand examples are provided for each and every individual functionality / topics.

The book is divided in to three major parts. The first part will cover the introduction, basics of relational databases and setting up the environment and tools required to get started with ClickHouse.

The second part will cover the basics ClickHouse SQL, querying the ClickHouse tables via SQL and SQL functions used to perform analytical tasks. It is followed by the core table engine of ClickHouse – MergeTree family and light weight Log family of engines. We will also cover on integrating ClickHouse with external data sources like HDFS, MySQL, Postgres, etc. and special table engines that are shipped with ClickHouse.

The final part will cover the administrative aspects of ClickHouse. Various server configurations, role based access control, user roles, quotas are covered in this section. This will be quite helpful for database administrators and developers.

Key topics are summarized at the end of every chapter and most of the chapters have a Q&A section to test your understanding. All the very best.

Downloading the code bundle and coloured images:

Please follow the link to download the
Code Bundle and the *Coloured Images* of the book:

https://rebrand.ly/3154de

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit **www.bpbonline.com** and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at **https://GitHub-bpbpublications/Up-and-Running-with-ClickHouse**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at **https://github.com/bpbpublications**. Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**.

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

CHAPTER 1
Introduction

In this fast-paced age of digital economy, the data has gained more importance ever since the dawn of this century. Data-driven organizations are growing at a rapid pace; and the importance of data cannot be stressed enough. Data helps the organizations to understand and solve problems, make informed decisions, and improvise their process. With the ever growing demand for various types of data, there has been multiple efforts to store the data in an efficient and an optimal way, which in turn has led to the development of different database technologies. Currently, there are more than 300 database management systems that are actively developed and maintained (source: db-engines.com).

In this book, we will focus on a relatively new database management system called ClickHouse, which is a column-oriented database management system used for the online analytical processing systems.

Structure

In this chapter, we will discuss the following topics:

- What is a database?
- Different types of database management systems
- Online transaction processing versus online analytical processing systems

- Row versus columnar database
- Introduction to ClickHouse

Objectives

After reading this chapter, you will be able to:

- Know what is a database
- Understand the commonly used database types
- OLAP versus OLTP and when to use the row and columnar databases
- Brief history of ClickHouse and its success stories

Data and databases

Data - "Information, especially facts or numbers, collected to be examined and considered and used to help decision-making, or information in an electronic form that can be stored and used by a computer"

- Cambridge Dictionary

Simply put, the data is a collection of numbers (measurements or observations), words, or just description of things. There is a small difference between the data and the information. The information is derived from the smaller chunks of data, which has to be analyzed, put into a context in order to retrieve the meaningful information. Data is collected, organized, and stored electronically in the computer database, which is also used to manage the stored collections.

In the last decade, the ever growing demand for data has caused a rapid increase in the volume of data, which has to be stored. This has left the traditional data storage/processing applications behind and a new subfield called the big data has taken the center stage. With an exponential increase in the amount of data that is stored, the speed of processing has remained as a challenge, especially for online analytical applications. This, in turn, has led to a rapid development of a special category of systems called the **Online Analytical Processing Systems** (**OLAP Systems**). Before getting into them, we shall have an overview on different types of database systems.

Different types of database management systems

Although it is not a formal classification, the following are the different types of database management systems classified based on how the data is stored and

retrieved. In spite of this being classified into different groups, these systems may also exhibit some commonalities.

Relational database

In relational databases, the data is organized into tables of rows and columns and the information in multiple tables can be connected together by a logical connection called relationships. The rows are also referred as records or tuples and the columns as attributes.

Each row in the table will have a unique key called the primary key, which is used to define the relationship among the tables. When a new row is added to the table, a new and unique primary key is added. The primary key in one table will become a foreign key in the other table, as shown in the following figure:

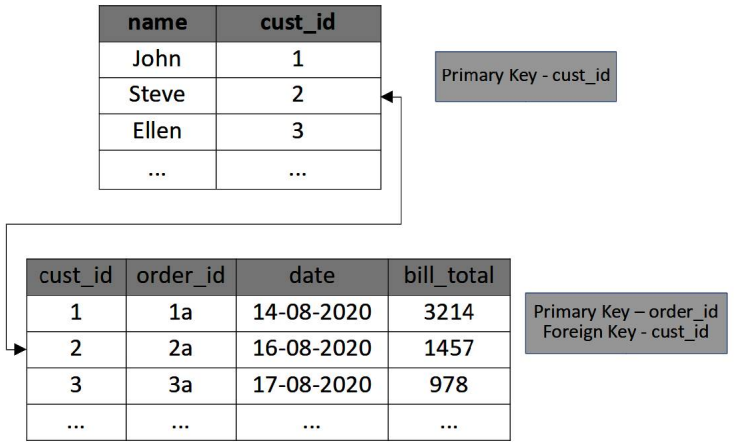


***Figure 1.1**: Tables in relational databases*

In the preceding example, we have two tables. The first table has the customer ID (**`cust_id`**) as a primary key and the second one has the order ID (**`order_id`**) as a primary key. The customer ID field in the second table is an example of a foreign key. In order to find out the orders made by the customer named Steve (with customer ID 2), we can use the customer ID to extract the relevant records from both the tables.

Most of the relational DBMS uses the **Structured Query Language (SQL)** for maintaining and querying the database. Examples of RDBMS include Oracle, MySQL, PostgreSQL, MariaDB, Microsoft Access, Microsoft SQL Server, IBM DB2, and SQLite.

Advantages:

- Simple
- Easy to query using SQL

- Accuracy – primary keys prevent data duplication
- Reduces data redundancy and improves data integrity via normalization
- Supports transactions
- Supports ACID properties in transactions to ensure data validity

No-SQL database

No-SQL (sometimes called **Not Only SQL**) databases provide an alternate way of storing and retrieving a large amount of the unstructured data. More recently, some of the No-SQL databases added support for SQL-like query languages. The two major ways of storing data are:

Key–value stores

The data is stored as key–value pairs where the keys are usually unique (like the primary key) and values are blobs (can be of any data type). The responsibility of decoding the values correctly lies with the client accessing the database. The client can read, write, update, and delete the values. As the values are read based on keys, this method is fast and scalable for larger datasets.

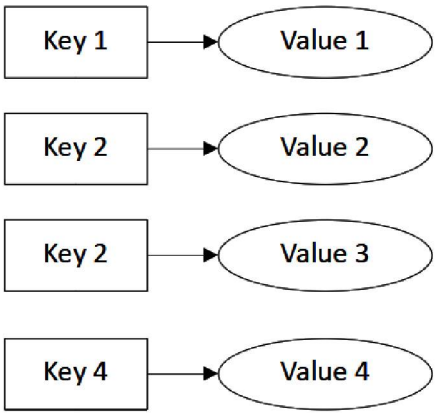


***Figure 1.2**: Sample key–value store*

For example – Redis and Memcached.

Advantages:

- Fast read–write operations
- Supports unstructured data
- Easy to scale
- High availability

- Resilient to failures

Document store

This is quite similar to the key–value store; however, the difference is that the value is usually a structured or a semi-structured data and is stored in XML, JSON, or BSON format.

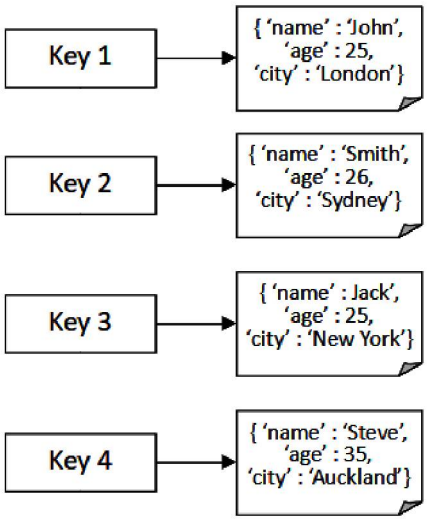


***Figure 1.3**: Sample document store*

For example – MongoDB and CouchDB.

Advantages:

- Flexible – the structure of the document need not be consistent
- Prior knowledge on data schema is not required
- Information can be added, changed, deleted, and updated easily like in a relational database
- Easily scalable
- High availability
- Easy to recover from failures

Graph database

A graph database consists of a collection of nodes and edges. A node represents an object and an edge is the connection between the two objects. Each node has an associated unique identifier that expresses the key–value pairs. Similarly, an edge

is also defined by a unique identifier that contains information about a starting or an ending node and properties like direction, parent–child relationships, actions, ownership, and so on.

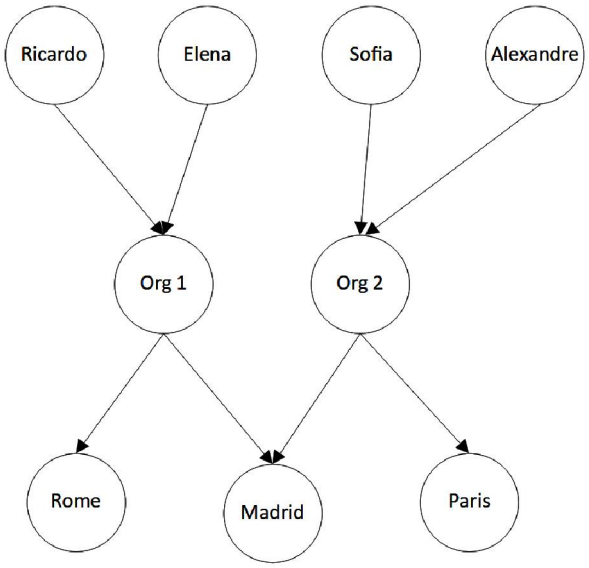


Figure 1.4: *Data relationship modeled in a graph database*

This example is of a graph data model that can be stored in a graph database. The three different set of nodes are persons, the organizations they worked in, and the location of the organizations.

Advantages:

- For intensive data relationship handling, graph databases improve the performance by several orders of magnitude.
- Flexibility – instead of modeling a domain ahead of time, data can be added to the existing graph structure without endangering the current functionality.

For example – JanusGraph and Neo4J.

Time–series database

As the name suggests, time–series databases are designed to store data that change with time. The data can be of any kind, which is periodically collected over time. Usually, they are the metrics collected from some systems. Although the time–series data can be stored in traditional relational databases, the key difference is that the records are appended and updates and deletes are not done. The time–series databases are optimized for large amounts of data ingestion and aggregation of the recorded metrics.

Advantages:

- Optimized to accumulate data periodically at a larger scale
- In-built data aggregation functionality
- Functions and operations common to time–series data analysis
- Scalable
- High availability
- Easy to recover from failures

For example – InfluxDB, Prometheus, and Graphite.

Transactional and analytical systems

The data stored in the database can be put to use by either processing it (updates and deletions) or using the data available to derive insights and perform analytical operations on the data. Based on this, the software systems powered by the data can be broadly classified into the following two categories:

OLTP

Online transaction processing system manages transaction-oriented software applications (frequent database modifications are common here). The common examples of OLTP applications are online ticket booking, e-commerce applications, and so on. They handle day-to-day transactions and also regular business operations.

A transaction is a group of operations, which is usually treated as a single unit and are performed on a smaller number of records. The database modifications can be inserting a new record, deleting an old record, or updating an existing record. The OLTP queries are simpler and short and hence require a lesser resource.

Advantages

- **Concurrency**: OLTP systems allow a large number of concurrent transactions and data access.
- **Acid Compliance**: ACID is an acronym for atomicity, consistency, isolation, and durability. This is intended to guarantee data validity despite the errors and unforeseen circumstances that may affect the transactions. More details are available in *Chapter 2, Introduction to Database Design*.
- **Availability**: The latest data is available to all the users even with large transaction rates.

- **Integrity**: Since the data is normalized and stored, the integrity of data is maintained throughout. More information about normalization is available in *Chapter 2, Introduction to Database Design*.

OLAP

Online analytical processing system helps in performing analytical calculations, derive insights to make business decisions, and future planning. An example of an OLAP scenario is viewing a financial report of a company's profit based on the sales of the products stored in their database.

OLAP systems enable users to execute complex queries on a large dataset and the results are used for further analysis and deriving insights. Since data retrieval is the main operation performed, failures are not detrimental to data integrity. Transactions are less frequent in OLAP; and OLTP systems usually become the data source for OLAP. The commonly performed analytical operations from OLAP are as follows:

- **Roll-up**: Data can be summarized or consolidated along with the dimension (from days to months).
- **Drill-down**: Reverse of roll-up. For example, drilling down from months to weeks.
- **Slice**: For analysis on a particular slice/segment of data. For example, profit during the month of April.
- **Dice**: Data is selected from multiple dimensions for further analysis. For example, profit arising from sales of ice cream in a supermarket.
- **Pivot**: Data is rotated (from rows to columns with or without aggregations) and analyzed.

Advantages:

- Used for analytics and business forecasting/planning with a large amount of data.
- Business-focused complex analytics are possible.
- Data can be aggregated or represented in a detailed view based on the user's requirement.
- Good for performing time–series data analysis.
- Can perform analysis over a huge dataset (in scale of terabytes) within a short time period.

Storing the structured in database systems

Based on the different ways used to organize and store structured data, database systems can be broadly classified into:

- Row-oriented DBMS
- Column-oriented DBMS

Row-oriented DBMS

In the row-oriented database, the data is organized and stored based on records (rows), which is the traditional way of storing the data in databases. In a disk, the subsequent rows of data are stored next to each other. For example, MySQL, Postgres, and Oracle.

Row	Name	Employee ID	Department	Age
1	John	2356	Sales	32
2	Jack	4582	IT	25
3	Alice	1478	IT	28
x	...	...	...	...

***Figure 1.5**: Data organization in row-oriented database*

In a disk, the data is stored row by row and due to this, writing the data is faster as it can be added to the last row of the existing data.

John	2356	Sales	32	Jack	4582	IT	25	...	...

***Figure 1.6**: Data stored in a disk for row-oriented database*

The reading rows from the row-oriented databases are usually fast. However, for aggregations, extra data is read from the disk, which slows down the operation. In order to find out the average age of the employees in the table, the whole table will be read once and the relevant data will be used for computing the average.

Column-oriented DBMS

In column-oriented databases, the data is organized and stored based on fields (columns). Here, the data from the same column are stored next to each other in a disk and different columns are stored separately.

Name	John	Jack	Alice	...
Employee ID	2356	4582	1478	...
Department	Sales	IT	IT	...
Age	32	25	28	...

Figure 1.7: Data stored in a disk as multiple files for each column in the column-oriented database

When it comes to aggregations, the column-oriented databases offer greater speed. For example, to find out the average age of the employees, the particular column is selected and computed directly, whereas the columns that aren't required are skipped while reading the data for aggregation.

It is easier to compress the data stored in columns, which further improves the system performance. The data is also sorted while storing in the column-oriented databases, whereas in the row-oriented databases, the sorting can be done while retrieving the data based on the index, but it is rarely sorted and stored in disk.

So, the row-oriented databases are quite useful in OLTP scenarios, whereas the column-oriented databases are widely used in OLAP scenarios.

ClickHouse

According to the official website, *"ClickHouse is a fast open-source OLAP database management system. It is column-oriented and allows to generate analytical reports using SQL queries in real-time. The initial goal for ClickHouse was to remove the limitations of OLAPServer (tool used in Yandex prior to ClickHouse) and solve the problem of working with non-aggregated data for all reports. But over the years, it has grown into a general-purpose database management system suitable for a wide range of analytical tasks."*

ClickHouse was initially developed by Yandex, a Russian company for their service called **`yandex.metrica`** (the second largest web analytics platform). The development started in 2009 and was open-sourced in 2016 (Apache License 2.0).

The ClickHouse development team was recently moved to a new organization called ClicKhouse Inc under the same creators.

Features:

- Faster data retrieval
- Highly scalable
- Data compression (using LZ4, ZSTD, Delta, and so on)
- Efficient usage of the available hardware (parallel processing on multiple cores and distributed processing using multiple servers)
- Fault-tolerant and reliable (via replication)
- SQL dialect for querying
- Easy to learn and use

Based on the official website, YandexMetrica uses 374 servers, which stores over 20.3 trillion rows in ClickHouse with 12 billion events added daily. The compressed data is about 2 petabytes (2,000 terabytes) and the uncompressed data is approximately 17 petabytes (compression ratio of ~11.7 %).

When to use ClickHouse?

Being a database designed for OLAP scenarios, ClickHouse is optimized for reading the data quickly rather than the transactions. However, in the recent versions of ClickHouse, updates and deletes are supported, unlike some of the other OLAP databases. The point updates and deletes are not efficient and hence bulk updates and deletes are recommended. ClickHouse is optimized for the following scenarios:

- Queries for reading the data is common and adding the data is less common.
- A small subset of columns and large number of rows are read.
- Tables have large number of columns.
- Data is added to the database in large batches every time, rather than by adding individually or few rows at a time.
- Data doesn't need to be modified (although later versions of ClickHouse supports modifications like delete and update).
- Rate of queries are usually around 100 or less per second.
- Result from the query is a small subset of the original data.
- Result should fit in RAM of the ClickHouse server.

Onward

This book is aimed at being a practical guide to learn ClickHouse and special effort is taken to ensure that the prior knowledge required to get started is minimal. This

book follows bottom-up approach where we start by setting up a database, learn the querying part and then deep dive in to the different table engines available. The book is divided into three major parts:

The first part consists of *chapters 1, 2, and 3* that are introductory in nature and covers the environment setup and the fundamental topics in the database management systems. The second part consists of *chapters 4–10* and they cover basics of querying data from ClickHouse, table engines in ClickHouse, and integrating ClickHouse with external data sources and miscellaneous engines available in ClickHouse. *Chapters 11 and 12* cover administrative aspects like configuring the ClickHouse server and the user management in ClickHouse.

The Appendix section includes installation steps for the software that are used in the respective chapters. Recommended way to work with this book is to start by setting up the environment yourself and to work out all the examples and instructions provided. If the reader faces any challenges in following the instructions provided in the Appendix section, for example, setting up the other software, there are ample resources available online that can help the readers navigate through them.

ClickHouse software development is going at a rapid pace and, on average, there are two **Long-Term Support (LTS)** releases every year. The book covers important topics in the latest LTS version at the time of writing (21.3 LTS). There are many experimental features available, which will be covered in subsequent editions of this book. ClickHouse has a well maintained and extensive official documentation and in case a feature is not covered in the book, the official documentation may have it covered.

Conclusion

So far, we have learned the importance of data, on what a database is, different types of database management systems, OLAP and OLTP systems, and row- and column-oriented DBMS. We also had a brief introduction on ClickHouse, when to use it, and some of the prominent adopters of this technology. In the upcoming chapters, we shall go through the basics of relational databases and continue with setting up our own ClickHouse sandbox environment.

Points to remember

- Data is chunks of information stored electronically in **Database Management Systems (DBMS)**.
- Based on the way the data is stored, the DBMS are classified into row-oriented and column-oriented DBMS.
- Online transactional processing systems and online analytical processing systems are classified based on how the data is used by the system.

- Row-oriented DBMS are suitable for OLTP systems and column-oriented DBMS for OLAP systems.
- ClickHouse is superfast owing to its design (multi-core processing) and storage optimizations (columnar storage and data compression).
- ClickHouse is optimized for reading the data from large datasets. Transactions are not recommended, although supported.
- ClickHouse uses SQL dialect for querying.
- Multiple prominent companies have adopted ClickHouse for their own use cases.

Multiple choice questions

1. **Information is derived from smaller chunks of data, which has to be analyzed, and put into context.**
    a) True
    b) False

2. **No-SQL databases does not support SQL-like languages for retrieving the data.**
    a) True
    b) False

3. **Time–series data can be stored in relational databases.**
    a) True
    b) False

4. **Group of operations, which is usually treated as a single unit, performed on a smaller number of records is called as.**
    a) Mutation
    b) Transaction
    c) Roll-up
    d) Pivot

5. **Which of these are commonly performed analytical operations from OLAP systems?**
    a) Slice
    b) Dice

c) Drill-down

d) Pivot

e) All of the above

6. **Row-oriented databases are quite useful in OLAP scenarios, whereas column-oriented databases are widely used in OLTP scenarios.**

   a) True

   b) False

7. **ClickHouse is a row-oriented database management system.**

   a) True

   b) False

8. **ClickHouse is recommended when a small subset of rows and large subset of columns are read frequently.**

   a) True

   b) False

Answers

1. **a**
2. **b**
3. **a**
4. **b**
5. **e**
6. **b**
7. **b**
8. **b**

References

- **https://clickhouse.tech/docs/en/introduction/history/**
- **https://clickhouse.tech/docs/en/introduction/adopters/**
- Colliat, George. *OLAP, relational, and multidimensional database systems*. ACM Sigmod Record 25.3 (1996): 64-69

CHAPTER 2

Relational Database Model and Database Design

In the previous chapter, we have seen different types of database management systems and in this chapter we shall get into the basics of database management systems. ClickHouse stores the data in structured form and despite being a database designed for OLAP applications, it uses some important concepts from relational model. ClickHouse can also ingest data from other popular relational databases like MySQL and Postgres and knowing these important concepts from relational model can be quite helpful. This chapter aims to serve as a quick introduction for the beginners and as a refresher of important concepts in a relational database.

The relational model was first proposed by an English computer scientist named Edgar Frank Codd in the year 1969. The relational model is an approach to manage the data stored in a database; and the database that is organized based on the relational model is called a relational database.

For a comprehensive and a detailed resource on the topics discussed in this chapter, kindly refer to the book **RDBMS In-depth** authored by Dr. Madhavi Vaidya (BPB Publications).

Structure

In this chapter, we will discuss the following topics:

- Relational model

- Types of relations
- Keys and index
- Normalization
- Data integrity and transactions
- Codd's 12 rules

Objectives

By the end of this chapter, you will be able to:

- Understand the fundamental concepts pertaining to the relational data model
- Understand the different types of relationships, and relational databases
- Get to know about the Codd's rules for RDBMS

Relational model

A relational model is a commonly used data model to store and process the data. In relational data model, a database is represented as a collection of tables. An entity is a real-world object (could be either tangible or intangible) and the attributes describe the features of an entity (properties of an entity). For example, if students are considered to be the entities, then the possible attributes could be name, age, height, weight, and so on.

In a relational model, entities sharing the same set of properties are stored in an entity set, which is also called as a table. The association between the entities is called as a relationship. A key could be an individual attribute or a group of attributes that uniquely identifies an entity among the entity set (table).

In a relational model, the data and relationships are represented by a collection of inter-related tables. Each table consists of columns and rows, where a column represents an attribute of an entity and rows represent records (also called tuples). In a relational model, a key could be a column or a group of columns and the keys are helpful in uniquely identifying the rows and maintaining a relationship among the tables.

Database table relationships

There are three different types of relationships that are possible between the data stored in a relational database. Two tables are used while working with relationships: the first one is the primary table or the parent table and the second one is the related table or the child table.

One-to-one

In a one-to-one relationship, each record in the parent table relates to only one record in the child table and vice versa. For example, consider a student database with a table on student's personal information (parent) and another table on final grades (child). Each row in the personal info table is about a student and they can correspond to only one record in final grades table.

ID	Name	Age	Gender
21547	Jim	20	Male
22598	Rachel	21	Female
24785	Glenn	20	Male
....	...	...	...

***Figure 2.1**: Parent table with one-to-one relationship*

ID	Grade
21547	B
22598	A
24785	C
....	...

***Figure 2.2**: Child table with one-to-one relationship*

One-to-many

In a one-to-many relationship, a record in the parent table may relate to zero, one, or more than one record in the child table.

For example, a customer info and orders info tables from a supermarket database. Each customer can have zero, one, or many orders made from a store as illustrated in the following table. This is the most common type of relationship found in relational databases.

ID	Name	Age	Zip
2125	Bob	45	21352
1254	Dominic	36	65421
3215	Monica	64	89213
...	...	...	...

***Figure 2.3**: Parent table with one-to-many relationship*

ID	Order ID	Date
2125	8532145	2020-08-01
2125	8641253	2020-08-03
2125	8641258	2020-08-03
3215	8736545	2020-08-05
3215	8796547	2020-08-05
3215	8874568	2020-08-07

***Figure 2.4**: Child table with one-to-many relationship*

Many-to-many

In a many-to-many relationship, one record in the parent table can relate to many records in the child table and vice versa. Consider an example of a database containing tables about authors and books. A single author may have written multiple books and a single book may have been written jointly by many authors.

Author ID	Author Name	Book ID
587	Dave	5412
587	Dave	6547
619	Amelia	3212
654	Hannah	5412
785	Daniel	9547

***Figure 2.5**: Parent table with many-to-many relationship*

Author ID	Book Name	Book ID
587	ABCD	5412
587	CDEF	6547
619	QRST	3212
654	ABCD	5412
785	WXYZ	9547

***Figure 2.6**: Child table with many-to-many relationship*

Keys

Keys are used to establish a relationship among the tables in a relational database. The keys are also used to identify a record, uniquely in a table. As mentioned earlier, the database key in a relational database could be a single attribute or could be a combination of attributes. As ClickHouse uses primary keys extensively, knowing the other types of keys can be helpful while working with tables in other relational databases like MySQL or PostgreSQL. The following are the different types of keys in a relational database:

Super key

Super key is a set of columns (fields) in a table, which can be used to uniquely identify a record in that table.

Employee ID	Name	Phone	Age
3212	Martin	452145	41
3541	Johnny	478522	43
3687	Will	487452	39
...	...	...	...

***Figure 2.7**: Table for illustrating super keys*

In the preceding table, **`Employee ID`**, **`Name`**, **`Phone`**, and **`Age`** are the super keys since the combination is always unique. Combination of **`{Employee ID, Name}`**, **`{Name, Phone}`**, **`{Phone, Age}`** are also super keys for this table. Super keys may also have fields that are not required for uniquely identifying a row. A super key is the superset of candidate keys.

Candidate key

Candidate keys are individual columns (fields) or minimal set of columns that can be used to uniquely identify the records. A primary key is selected from the candidate keys.

As mentioned in the earlier example, **`Employee ID`** and **`Phone`** are the candidate keys. A candidate key cannot be **`Null`** and should contain unique values.

Class	Name	Roll Number	Age
9	Alan	2	14
10	Elysse	10	15
9	Andrew	3	14
...	...	...	...

***Figure 2.8**: Table for illustrating candidate keys*

In the preceding example of the students table, **`{Class, Roll Number}`** combination is the candidate key and there are no other individual fields that can be used to uniquely identify a row.

Primary key

Primary key is the main key of the table that is unique and a non-Null key and is chosen from the candidate keys. A primary key should be able to uniquely identify a record in a table.

In the employee table, either one of the **`Employee ID`** or **`Phone`** can be the primary key. In the students table, the only option for a primary key is the combination of **`{Class, Roll Number}`**.

Foreign key

Foreign keys and primary keys are used to create a relationship between the two tables. A foreign key in a child table is a primary key in the parent table. This is illustrated in *figure 1.1* of *Chapter 1, Introduction*.

Alternate key

The candidate keys, which are not selected as a primary key, are the alternate keys. In the employee table, if the **`Employee ID`** is chosen as a primary key, then the alternate key would be **`Phone`** and vice versa. In students table, there are no alternate keys.

Surrogate key

Surrogate keys are the artificial keys used to uniquely identify a record in a table, when there are no primary keys available. The surrogate key does not have any meaning with respect to the data and is also known as the dumb key. They are generated just before the data is inserted to the table.

Name	In Time	Out Time
Andrew	10:35	17:25
Lisa	10:45	15:03
Geoff	10:51	18:16
...	...	...

***Figure 2.9**: Table for illustrating surrogate keys*

The preceding table holds the information about the employee time in office. As there are no fields that can be used to uniquely identify the rows, surrogate keys are used while storing the data in a database. The surrogate keys are usually integers that are incremented with every new row.

Natural key

Natural keys are the individual fields or set of fields that can be used to uniquely identify a record in a table. It differs from a surrogate key as natural keys are made up of real data (attributes of an entity).

Simple, composite, and compound key

A simple key is a single column (field) that can be used to uniquely identify a record in a table.

Composite and compound keys are two or more columns (fields) that can be used to uniquely identify a record in a table. Each column (field) in the compound key is a simple key, whereas at least one column (field) is not a simple key in the composite key.

For example, in the employee table, **`{Employee ID, Phone}`** would be a compound key, whereas **`{Employee ID, Phone, Name}`** is a composite key.

Index

Indexing is a technique that enables the user to quickly retrieve the data from a database table. Instead of scanning throughout the entire dataset, indexing allows the DBMS to scan the potential locations of the data required in disk thereby speeding up the data retrieval. Indexing can be of the following types:

Primary index

Primary index is an ordered file with two fields. The first one is same as the primary key and the second one is pointed to the location of the data for the key.

The index records may be dense or sparse. In dense indexing, the records are created for every search key value, whereas in sparse index, the index record is not available for every search key. Instead, the range of index columns and the block address of the data is stored. Dense index is fast but consumes more disk space and sparse index is comparatively slower but takes up less disk space.

Primary Key 1	Data Location for Primary Key 1
Primary Key 2	Data Location for Primary Key 2
Primary Key 3	Data Location for Primary Key 3
Primary Key ...	Data Location for Primary Key ...

Figure 2.10: Dense primary index

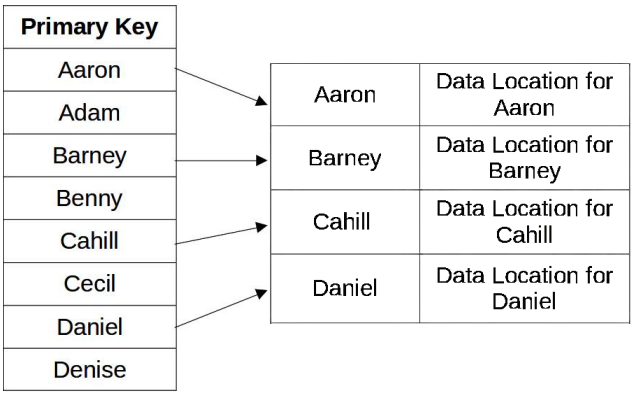


Figure 2.11: Sparse primary index

Secondary index

Secondary index is similar to primary index, but they are generated from a field (apart from the primary key), which is a candidate key. It is commonly used to speed up the data access for commonly used search parameters. The secondary indices point to buckets that hold the information on the data location pertaining to all the actual records with that particular search key value.

Multi-level index

Multi-level index is used when the primary index is too large to handle. In this method, the index is broken down into several smaller indices thereby reducing the memory requirements. In a simple 2-level indexing, a sparse index of primary keys will be the outer index and in the inner level, multiple primary indices will be present.

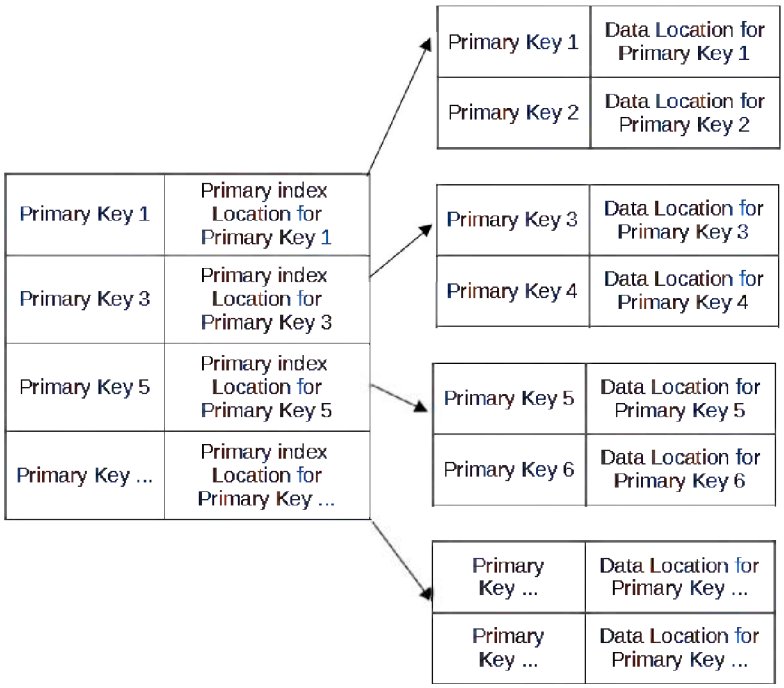


***Figure 2.12**: Multi-level indexing (2 level)*

The sparse index will point to the range of primary index keys in the inner level; and in the inner level, the primary index keys will point to the actual data location corresponding to the primary indices.

Database normalization

Database normalization is the process of organizing the data in the database tables in order to prevent data redundancy and anomalies. Although database normalization is itself a big topic, we shall restrict to the first three normal forms, which can be helpful while working with the normalized tables in other relational databases from which the data is ingested in to ClickHouse. There are three types of anomalies that can be caused as a result of not normalizing the database. For illustrating the types

of anomalies, we shall use the following table, which is not normalized. It contains information about the students in an educational institution.

ID	Name	Course Name	Postal Code
213	Marshal	Maths	30215
213	Marshal	Physics	30215
235	Sasha	Physics	32154
245	Brian	Math	33651
245	Brian	Chemistry	33651
289	Heidi	Math	30745

***Figure 2.13**: Table for illustrating database anomalies*

Update anomaly

In the preceding table, if the postal code of Marshal or Brian has to be updated, we have to make sure that the data in all the rows corresponding to Marshal or Brian are updated. Or else, it will lead to inconsistent data with respect to their postal codes.

Insert anomaly

In case a new student is enrolled only in an institution but not in any of the courses yet, we wouldn't be able to insert that student's information if the `Course Name` field doesn't allow Null values.

Delete anomaly

Let's assume that the institution stops offering chemistry course due to the underwhelming response from the students and we are required to delete the corresponding rows. In this case, the entire information about Sasha will be removed as she is enrolled only in the chemistry course.

Normal forms in database

Normal forms are used to handle redundancies in the database. There are six normal forms along with the Boyce–Codd normal form. Most of the practical applications require up to the third normal form.

1NF

The first normal form requires each table cell to have only one value and the records to be unique as shown in the following table:

Name	Phone	Age	Sport
Bob	32145	14	Cricket, Soccer
Ali	31524	13	Badminton, Basket Ball
Sara	32985	14	Tennis
Chloe	32145	15	Swimming
Peter	38563	14	Rugby

***Figure 2.14**: Table to be normalized (1NF)*

In the preceding table with children name and their sport of interest, we see that Bob and Ali have two types of sports in the sport column. 1NF form of the preceding table is given as follows:

Name	Phone	Age	Sport
Bob	32145	14	Cricket
Bob	32145	14	Soccer
Ali	31524	13	Badminton
Ali	31524	14	Basket Ball
Sara	32985	14	Tennis
Chloe	32145	15	Swimming
Peter	38563	14	Rugby

***Figure 2.15**: 1NF of table in figure 2.14*

In the preceding table, it is possible to find unique records using a combination of `{Name, Phone, Sport}` fields.

2NF

In order to normalize a database into the second normal form, it should already be in 1NF and should have single-column primary key. We can use the preceding database table in 1NF to convert it into 2NF. Since 2NF requires single-column primary key, we have to add a new field called the ID (unique ID for each child) and split the table into two (one for children info and another for sports), as shown in the following tables:

Name	Phone	Age	ID
Bob	32145	14	100
Ali	31524	13	101
Sara	32985	14	102
Chloe	32145	15	103
Peter	38563	14	104

***Figure 2.16**: 2NF of table in figure 2.15 - part 1*

ID	Sport
100	Cricket
100	Soccer
101	Badminton
101	Basket Ball
102	Tennis
103	Swimming
104	Rugby

***Figure 2.17**: 2NF of table in figure 2.15 - part 2*

3NF

For a database to be normalized into the third normal form, it should already be in 2NF and should have no transitive functional dependencies.

Name	City	Postal Code	ID
Bob	Houston	85412	100
Ali	Dallas	80541	101
Sara	Plano	82413	102
Chloe	Houston	85412	103
Peter	Dallas	80542	104

***Figure 2.18**: Table to be normalized (3NF)*

In the preceding table, the **`Postal Code`** is dependent on the ID (candidate key) and **`City`** (non-prime key, since it is not in candidate keys) is dependent on the **`Postal Code`**. This is a transitive functional dependency since **`City`** is transitively dependent on the **ID** field.

To convert this database into 3NF, we can create a new table for City and update the rest of the tables as follows:

Name	Postal Code	ID
Bob	85412	100
Ali	80541	101
Sara	82413	102
Chloe	85412	103
Peter	80542	104

City	Postal Code
Houston	85412
Dallas	80541
Plano	82413
Dallas	80542

***Figure 2.19**: 3NF of table in figure 2.18*

Data integrity

Data integrity in a database refers to the reliability and accuracy of the data throughout its life cycle. Data integrity should be ensured to prevent data from getting corrupted or being unusable, which could be either accidental or deliberate. Following are the types of data integrity:

Entity integrity

Entity integrity ensures that there are no duplicate rows in a table. This is achieved by using the primary keys, which are unique to each and every row.

Referential integrity

Referential integrity ensures that the data integrity between the two related tables, that is, foreign keys, should always match with the primary keys and there should be no orphaned record.

Domain integrity

Domain integrity ensures the presence of a valid data in the columns. For example, this can be achieved by setting proper data types to the columns (fields) and restricting possible values in the columns (fields).

User-define integrity

User-defined integrity is achieved by following the user-defined rules, which are not covered in the other three types of integrity, as discussed earlier.

Transactions

A database transaction is a group of operations that results in data modification and are performed on a smaller set of data in the database. A database transaction may consist of multiple low-level tasks. The database modifications can be either inserting a new record, deleting an old record, or updating an existing record. A transaction in a database can have any one of the following states:

Active

Transactions reach the active state when the execution of the transactional operation begins.

Partially committed

The transaction goes into the partially committed state once the final operation of that transaction is executed. The changes are not yet reflected in the database.

Failed

Transaction goes into the failed state if any of the operation is unsuccessful.

Aborted

When the transaction is unsuccessful, the system aborts that transaction and tries to bring back the data in the database to its original state.

Committed

When the transaction is successful, the changes are reflected in the database and the transaction is said to be in the committed state.

ACID properties

ACID properties of a transaction are used to maintain data integrity. **ACID** is an acronym for **Atomicity, Consistency, Isolation, and Durability**.

Atomicity

Atomicity ensures that all the transactions in the database are either executed fully or not executed at all and there should not be any partial execution.

Consistency

The data stored in the database should be consistent after the transaction is successful and committed.

Isolation

In case of simultaneous transactions in the database, every transaction should be isolated and no transaction should affect other transactions that are executed in parallel.

Durability

Once a transaction is completed successfully, the changes should be made available in the database even when there is a system failure.

Codd's rules

Edgar Codd proposed a set of rules (13 in total), which needs to be satisfied by any DBMS that can be considered as a relational DBMS. Till now, none of the available DBMS conforms to all the rules proposed by Codd. The (over)simplified version of the original rules are as follows:

Rule 0

If a DBMS system has to qualify as a relational database management system, that system must only use its relational capabilities to manage the database.

Rule 1 – information rule

All the information must be stored in the table format (with rows and columns).

Rule 2 – guaranteed access rule

Every single-cell value must be accessible via a combination of table name, primary key, and the attribute name.

Rule 3 – systematic treatment of Null values

Null values should be supported, which can represent missing or unknown values.

Rule 4 – active online catalog

The users must be able to access database's structure using the same query language, which is also used to access data stored in the database.

Rule 5 – comprehensive data sub-language rule

The system must support one relational language (for example, SQL) with linear syntax, which can be used within the application programs. The language must support operations such as data definition, view definition, data manipulation, transactions, and security operations.

The database should not allow data access without this language.

Rule 6 – view updating rule

All the views (and the records in the views) in the database should be updatable by the system.

Rule 7 – high-level insert, update, and delete

The database should support high-level insertion, updation, and deletion, which is not limited to a single record. Support for union, intersection, and minus operations must also be available.

Rule 8 – physical data independence

Changes in the physical storage structure of data must not impact the system.

Rule 9 – logical data independence

Changes in the logical level (such as table, column, rows, and so on) must not impact the user application.

Rule 10 – integrity independence

Integrity constraints should be independent of application programs and stored separately in a catalog.

Rule 11 – distribution independence

The end user must not be able to see that the data is distributed in multiple locations.

Rule 12 – non-subversion rule

If the system provides a low-level interface, it cannot be used to subvert the system. It should not allow by-passing the security and integrity constraints.

Conclusion

We have covered the important topics related to relational databases and the design part of the relational database. Although ClickHouse isn't a transactional database, most of the concepts learned here could be helpful in understanding and designing the database for different applications. In the next chapter, we shall start with setting up our ClickHouse sandbox and tools required.

Points to remember

- In a relational database, the entities are represented as records (rows) and attributes as fields (columns).
- Three types of relationships are possible between a parent and child tables.

- Different types of keys.
- Indexing a database for faster data access.
- Different types of anomalies can affect a database, which are not normalized.
- Three major forms of normalization are widely used.
- Transactions and its ACID properties.
- Codd's 12 rules.

Multiple choice questions

1. **Attributes describe the properties of an entity**
   a) True
   b) False
2. **In a relational model, columns in a table represent attributes and rows represent records.**
   a) True
   b) False
3. **There are _____ different types of relationships that are possible between the data stored in a relational database.**
   a) Two
   b) Three
   c) Four
   d) Five
4. **Super key is the minimal set of columns in a table, which can be used to uniquely identify a record in that table.**
   a) True
   b) False
5. **Candidate key is the minimal set of columns in a table, which can be used to uniquely identify a record in that table.**
   a) True
   b) False

6. **Primary key is chosen from the candidate keys and they should be able to uniquely identify a record in a table.**
   a) True
   b) False

7. **A foreign key in a parent table is a primary key in the child table.**
   a) True
   b) False

8. **Surrogate keys are artificial keys used to uniquely identify a record in a table.**
   a) True
   b) False

9. **_____ is the process of organizing the data in database tables in order to prevent data redundancy and anomalies.**
   a) Indexing
   b) Normalization
   c) Clustering

10. **Transactions are a set of operations that are treated as both single and a whole operation.**
    a) True
    b) False

Answers

1. **a**
2. **a**
3. **b**
4. **b**
5. **a**
6. **a**
7. **b**
8. **a**
9. **b**
10. **a**

CHAPTER 3
Setting Up the Environment

In the earlier chapters, we learned about the different types of databases available, how a row-oriented database differs from the column-oriented database and when to use ClickHouse. We have also seen the important and fundamental topics in relational model. In this chapter, we shall look at setting up ClickHouse, setting up a sample database, and how to use ClickHouse CLI.

ClickHouse is supported in any Linux or Mac OS X. Pre-built packages are readily available for Linux distributions such as Debian, Ubuntu, Cent OS, and so on. It can be compiled from the source in Mac OS and any Linux distributions. To get hands on with ClickHouse, we shall install Lubuntu 20.04 as the guest operating system using VirtualBox (open-source software, which supports multiple OS like Windows, Linux, and MacOS) and get started.

Structure

In this chapter, we will discuss the following topics:

- Installing ClickHouse
- Setting up the DBeaver (database tool)
- Creating a sample database in ClickHouse
- Querying the database
- ClickHouse **Command-Line Interface (CLI)**

Objectives

The aim of this chapter is to:

- Install and get started with ClickHouse
- Use a database tool for querying tables

Introduction

ClickHouse is primarily supported and can run on any Linux-based OS. As of now, Windows is not officially supported by ClickHouse. The choice of the Linux OS used in this book is Lubuntu 20.4 (long-term support version) and the ClickHouse version is 21.3.4.25 (official build, long-term support). Lubuntu is chosen since it has very nominal hardware requirements. The Lubuntu needs to be installed inside Oracle VirtualBox (version 6.1.12).

By choosing a VirtualBox-based setup we can use Linux OS inside the Windows/Mac OS environment without having to install it in a separate partition. In case the readers are comfortable with any other Linux OS, which is already installed on their hardware, then using VirtualBox may not be required. The hardware configuration used for VirtualBox setup for this book is as follows:

- 4 GB RAM
- 2 core processor
- 32 GB disk space

Higher hardware specs with a stable high-speed internet connection is recommended. Readers are requested to get comfortable with the Lubuntu OS, using the Linux terminal and basic Linux commands. Steps for installing Lubuntu in VirtualBox and basic Linux commands are available in the *Appendix* section.

Which version to use?

ClickHouse has around ten stable releases per year and two LTS (long-term support) versions on an average. LTS versions are usually supported for a year after its release. The most recent LTS version available at the time of writing this book is used (21.3 LTS). The readers can also install the latest version available to them and things should continue to work and should be able to follow the book.

Installing ClickHouse

ClickHouse is available in Ubuntu in the software repositories and can be installed directly. To install the latest LTS version, open the Linux terminal (`Start | System Tools | QTerminal` in Lubuntu 20.04) and execute the following commands:

```
$ wget https://repo.clickhouse.tech/deb/stable/main/clickhouse-
server_21.3.4.25_all.deb

$ wget https://repo.clickhouse.tech/deb/stable/main/clickhouse-
client_21.3.4.25_all.deb

$ wget https://repo.clickhouse.tech/deb/stable/main/clickhouse-common-
static_21.3.4.25_amd64.deb

$ sudo dpkg -i  clickhouse-*.deb
```

Enter the password for the **sudo** user when prompted. At the time of installation, if a screen appears that prompts you to enter the password for the default user, just leave it blank for now and press enter to proceed without any password for the default user. We will work on user roles and access control in the upcoming chapters. Once the installation is over, execute the following commands to start the ClickHouse service and the ClickHouse command-line client:

```
$ sudo service clickhouse-server start

$ clickhouse-client
```

When the **clickhouse-client** (command-line interface to ClickHouse) starts, the following information is displayed:

```
vijayanand@vijay-virtualbox:~$ clickhouse-client
ClickHouse client version 21.3.4.25 (official build).
Connecting to localhost:9000 as user default.
Connected to ClickHouse server version 21.3.4 revision 54447.
```

***Figure 3.1**: ClickHouse client*

Using the **clickhouse-client**, it is possible to execute ClickHouse SQL statements. For example, we shall see the databases that are already available and the tables inside the database. Execute the following statements inside the **clickhouse-client**:

```
:) SHOW DATABASES;
```

```
SHOW DATABASES

Query id: be526d5d-5bec-48b2-8ba1-7af953c49d75

┌─name────┐
│ default │
│ system  │
└─────────┘

2 rows in set. Elapsed: 0.002 sec.
```

***Figure 3.2**: ClickHouse databases available after fresh installation*

As this is a fresh installation, we will see that only two databases are available (default and system). Execute the following statements to get the list of tables available in the 'system' database:

```
:) USE system;
:) SHOW TABLES;
```

A long list of tables should be printed on the terminal screen. If everything goes expected until this point, we can safely assume that the installation is completed for ClickHouse.

ClickHouse CLI

ClickHouse provides a command-line interface via the `clickhouse-client` package. The client supports both batch and interactive modes.

In the interactive mode, the user is presented with a command line where queries can be executed. In the batch mode, the clickhouse client is specified with the `-query` or `-multiquery` parameter. The following parameters are accepted by the client:

- **--host, -h**: This is the server name (where ClickHouse is installed) that is the localhost by default. You can specify either the name or the IPv4/IPv6 address.
- **--port**: This is the port to connect to. The default value is `9000`. Note that the HTTP interface and the native interface use different ports that are specified in the config.
- **--user, -u**: This is the username.
- **--password**: This is the password.
- **--query, -q**: This is the query to process when using the batch mode.
- **--queries-file, -qf**: This is the file path with queries to execute.
- **--database, -d**: Select the current default database.
- **--multiline, -m**: This allows multiline queries
- **--multiquery, -n**: This allows processing multiple queries separated by semicolons.
- **--format, -f**: This uses the specified default format to output the result.
- **--vertical, -E**: This uses the vertical format by default to output the result.
- **--time, -t**: This prints the query execution time to 'stderr' in the non-interactive mode.

- **--stacktrace**: This prints the stack trace if there is any exception.
- **--config-file**: This is the name of the configuration file.
- **--secure**: This connects to the server over secure connection.
- **--history_file**: This is the file path containing the command history.

DBeaver

> **"Free multi-platform database tool for developers, database administrators, analysts and all people who need to work with databases."**
>
> *– DBeaver Website*

DBeaver is the database tool that will be used throughout the book for querying and managing our ClickHouse database. It is also possible to perform these tasks with `clickhouse-client` (CLI tool); however, for the beginners, it is easy with any database tool and they offer more features compared to `clickhouse-client`. The DBeaver community edition is an open-source software released under Apache License. DBeaver supports multiple other databases like MySQL, Postgres, and may other popular databases available. DBeaver can also be used for MySQL and PostgreSQL, which will be used in latter parts of this book.

Installation

The DBeaver community edition can be installed easily via the snap package manager in Lubuntu 20.04. Execute the following commands in the terminal window.

```
$ sudo snap install dbeaver-ce
```

Once the DBeaver is installed, start the application and if prompted to create a sample database to explore DBeaver, you can safely ignore (or try out if curious to explore) for now.

To start creating a new database connection to our newly installed ClickHouse, go to **`Database`** in the toolbar and select **`New Database Connection`** option in the toolbar as shown in the following screenshot:

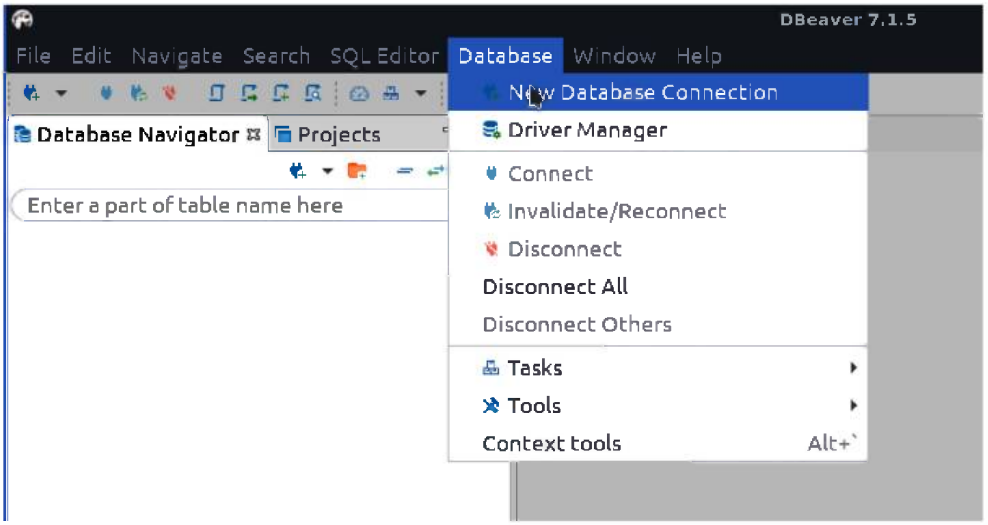


Figure 3.3: Creating a new database connection in DBeaver UI

The following window will be displayed. Select the **`All`** option in the left pane of the new window; and in the search field, type **`ClickHouse`** to get relevant results, as shown in the following screenshot:

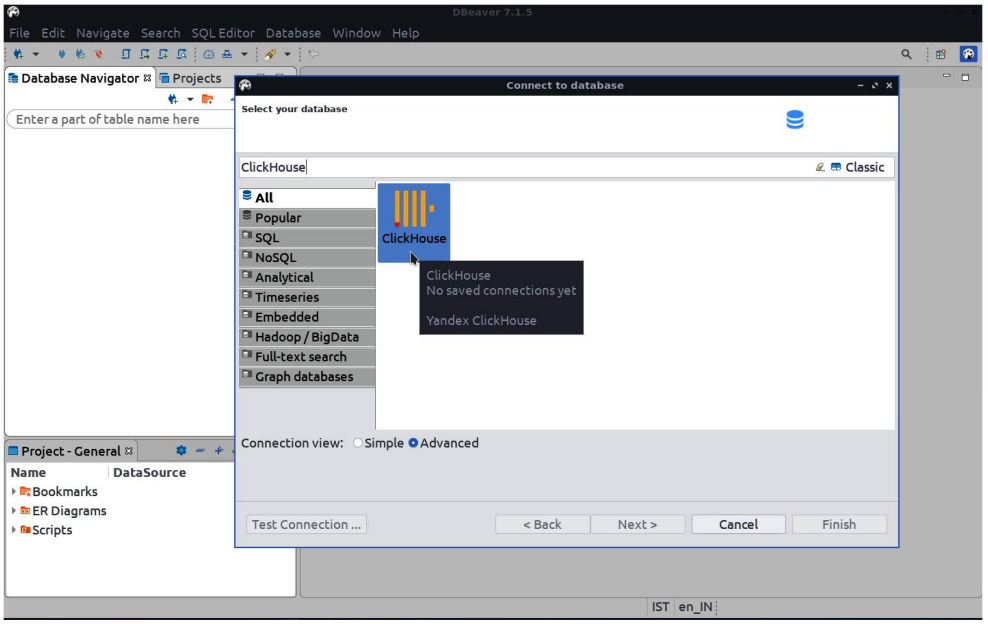


Figure 3.4: Searching for ClickHouse database to connect

Select **ClickHouse** and select the **Next >** option to navigate to the next screen, which will display the configuration settings for the ClickHouse database. Leave the settings as it is (should be configured for ClickHouse at the localhost and port **8123**).

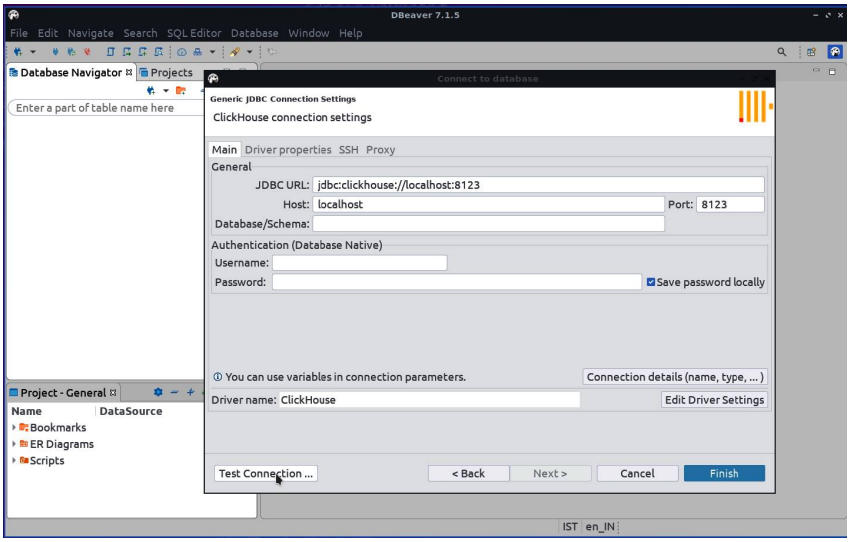


Figure 3.5: ClickHouse connection settings window

Now, select the **Test Connection** button. The following screen will be displayed. We can safely select the **Download** button to start downloading the JDBC drivers required for DBeaver to connect to ClickHouse. DBeaver provides the drivers required (also for MySQL and PostgreSQL) and it is not necessary to download them separately, as shown in the following screenshot:

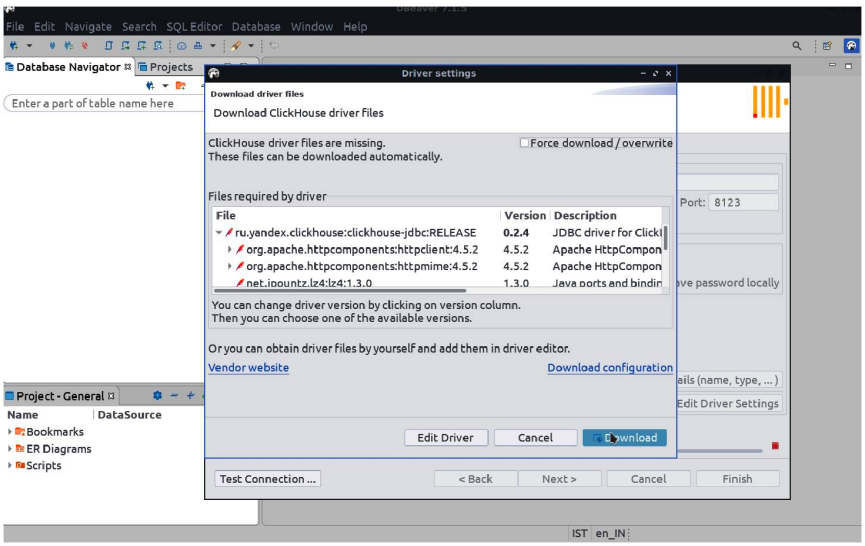


Figure 3.6: Downloading required driver files for ClickHouse connection

Once the download is complete, the following message will be displayed indicating the connection status as connected:

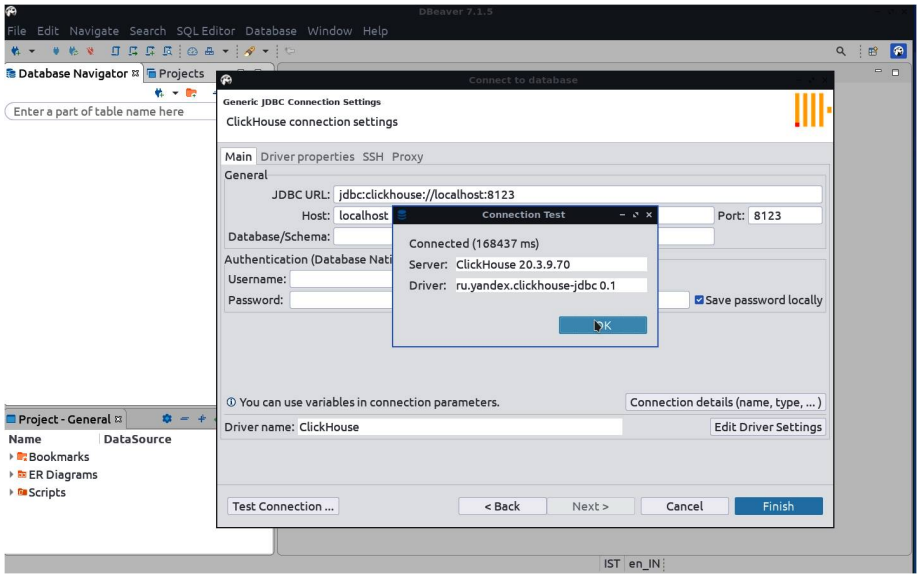


Figure 3.7: Successful connection test

Select the **OK** button and then select **Finish** to complete the connection setup. You should be able to see the newly added connection in the database navigator that is available in the main window, as shown in the following screenshot:

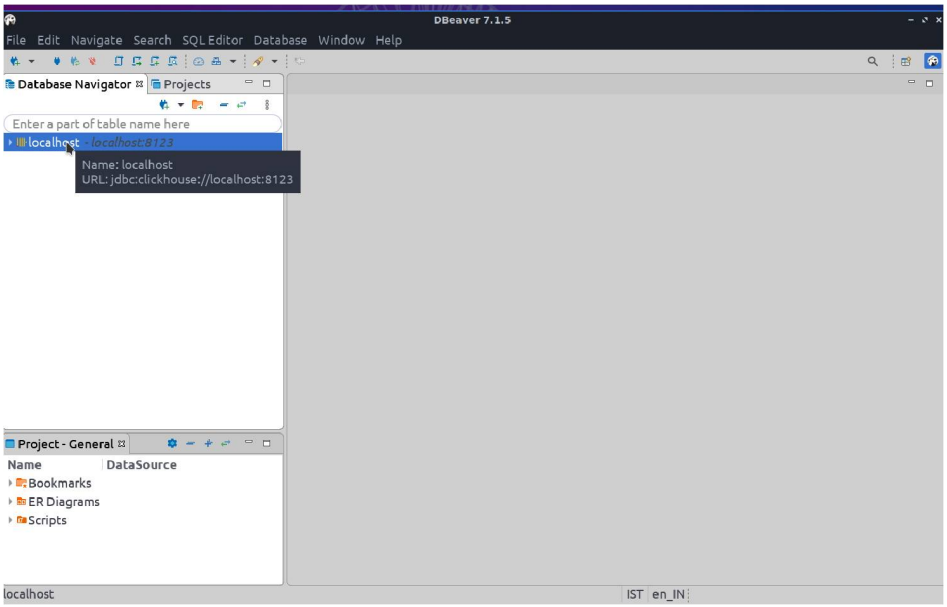


Figure 3.8: Database navigator in DBeaver showing ClickHouse connection

Now, press the `F3` button to open an SQL editor for the database using which we can query the database. We are ready with our setup and we shall start testing with a sample dataset.

Creating a sample database and table

To test the functionality of ClickHouse, we shall create a sample database and a table, populate it with some real-world data, and execute few simple SQL queries on it.

Also, we will use the individual household electric power consumption dataset, which is available at the UCI Machine Learning Repository. The dataset contains more than two million measurements collected from a single house between December 2006 and November 2010. The dataset is processed and updated with a new `DateTime` column before inserting it into the database. The columns available in the raw dataset are as follows:

- `date`: Date (dd/mm/yyyy).
- `time`: Timestamp (hh:mm:ss).
- `global_active_power`: Household global minute-averaged active power (kilowatt).
- `global_reactive_power`: Household global minute-averaged reactive power (kilowatt).
- `voltage`: Minute-averaged voltage (Volts).
- `global_intensity`: Household global minute-averaged current intensity (Amperes).
- `sub_metering_1`: Energy consumed in the kitchen.
- `sub_metering_2`: Energy consumed in the laundry room.
- `sub_metering_3`: Energy consumed by the water heater and the air conditioner.

To begin with, we will start by creating a new database and a table in it. Go to the SQL editor in DBeaver and execute the following query to create a new database.

Note: The dataset is available under "Creative Commons Attribution 4.0 International (CC BY 4.0)" license. The sources as per the website are as follows:

Georges Hebrail, Senior Researcher, EDF R&D, Clamart

FranceAlice Berard, TELECOM ParisTech Master of Engineering Internship at EDF R&D, Clamart, France

URL: https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption

```
CREATE DATABASE electricity;
```

Once the query is typed, select the query using the mouse and press **`Ctrl + Enter`** to execute the query, as shown in the following screenshot:

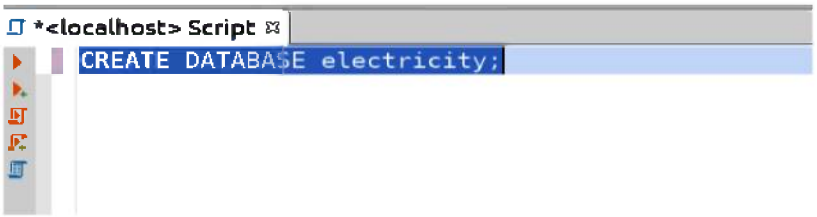


***Figure 3.9**: Selecting and executing the SQL statement in DBeaver's SQL editor*

Once the database is successfully created, execute the following query to create a new table in which the data is populated.

```
CREATE TABLE electricity.consumption (
`Date` Date,
`DateTime` DateTime,
`Global_reactive_power` Nullable(Float64),
`Global_active_power` Nullable(Float64),
`Voltage` Nullable(Float64),
`Global_intensity` Nullable(Float64),
`Sub_metering_1` Nullable(Float64),
`Sub_metering_2` Nullable(Float64),
`Sub_metering_3` Nullable(Float64))
ENGINE = MergeTree()
PARTITION BY toYYYYMMDD(Date)
ORDER BY (DateTime)
SETTINGS index_granularity = 8192;
```

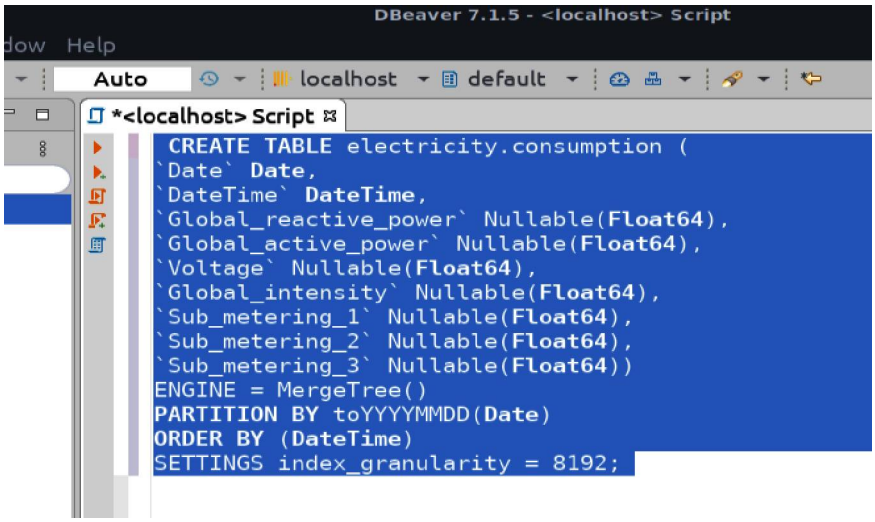


***Figure 3.10**: Executing multiline SQL statement in SQL editor*

Similarly, select the whole script and press **`Ctrl`** + **`Enter`** to execute it. The table should be created. The query syntax will be covered in the upcoming chapters. Next step is to download and push the data into the table. For this, you can make use of the Python script, which is available in the Appendix section. Let us begin by installing the software dependencies and execute the following commands in the terminal window:

```
$ sudo apt install wget curl python3-pip
```

```
$ pip3 install pandas requests clickhouse-driver
```

The second line has the Python packages required to download and insert the data in ClickHouse. Next step is to run the Python script. The script is available at:

https://github.com/vj-1988/ClickHouse/blob/master/download_insert_electricity_data.py

Or

https://raw.githubusercontent.com/vj-1988/ClickHouse/master/download_insert_electricity_data.py

The script can be copied from the source URL (or can be typed manually, referring the Appendix section) and paste it in any of the location in the Lubuntu OS. Navigate to the saved location and execute the script using the following command:

```
$ python3 download_insert_electricity_data.py
```

After the file has successfully run, the data should be available in our ClickHouse. We shall test this via DBeaver. Go to the SQL editor and execute the following SQL statement:

```
SELECT COUNT() FROM electricity.consumption;
```

This will return the total number of rows in the table. It should be more than 2 million, if the script has run successfully. Let us take a look at some data available in the table. Execute the following SQL statement:

```
$ SELECT * FROM electricity.consumption LIMIT 5;
```

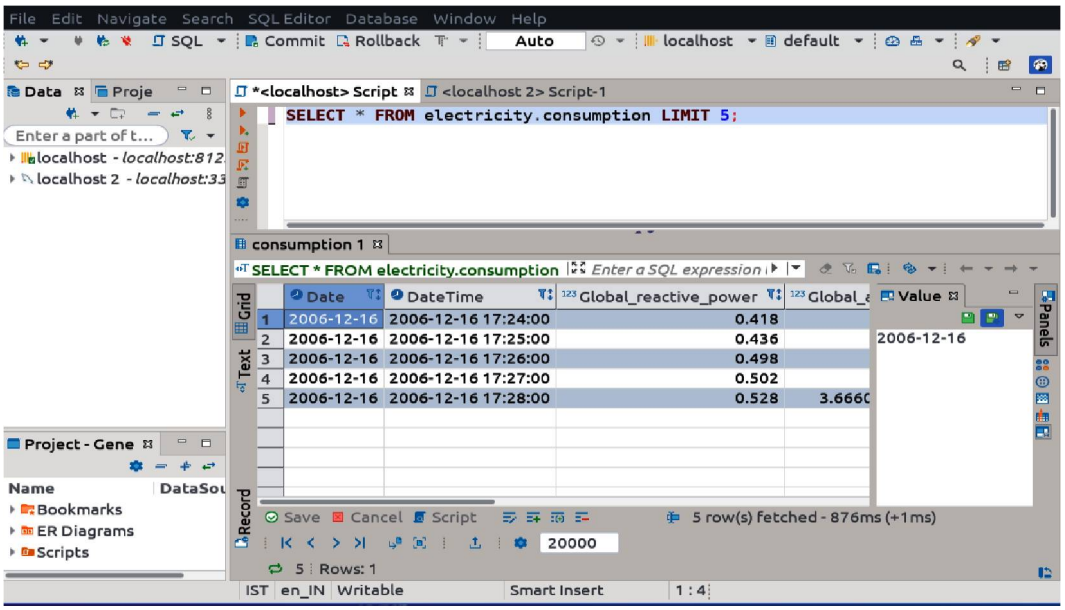


Figure 3.11: Results from the sample database

Conclusion

In this chapter, we have successfully installed ClickHouse, **`clickhouse-client`**, and have set up DBeaver. We were able to create a sample database and table and load a sample dataset into it and we were also able to query it. In the next chapter, we shall take a look at some fundamental, but important relational database concepts before diving into the SQL for ClickHouse and features of ClickHouse.

Points to remember

- ClickHouse is installed (long-term support version) in Lubuntu 20.04.
- Created a new database and table in ClickHouse and populated the table with data.
- Installed DBeaver and connected it to ClickHouse for executing SQL statements.

CHAPTER 4
ClickHouse SQL

In the preceding chapters, we learned about the fundamental concepts of relational databases and to set up ClickHouse sandbox. In this chapter, we shall learn the basics of ClickHouse SQL. **Structured Query Language** (**SQL**) is a domain-specific language, which is used to operate databases (more commonly on relational databases). SQL can be used to query data, manipulate it, and define the data in database/tables and for data access control. The initial version of SQL (called **Structured English Query Language** also known as **SEQUEL**) was first developed by David Chamberlain and Ronald Boyce. Towards the end of the 1970s and early 1980s, multiple commercial products based on the relational model started emerging and by 1986, ANSI and ISO had adopted the standard SQL definitions.

ClickHouse supports a declarative query language that is similar to ANSI SQL. Despite the similarities and being shipped with lot of features and functionalities, the interoperability of the queries with other databases is not guaranteed and some of the functionalities defined in the SQL standard may not be available.

Structure

In this chapter, we will discuss the following topics:

- SQL syntax in ClickHouse
- Operators supported in ClickHouse

- Datatypes in ClickHouse
- ClickHouse SQL
- Joins in ClickHouse
- Unions in ClickHouse

Objectives

The objectives of this chapter are to:

- Learn the SQL basics
- Know about different SQL statements
- Query and manipulate the ClickHouse database
- Perform joins and unions in ClickHouse

SQL syntax in ClickHouse

ClickHouse comes with an extended SQL-like language that supports a variety of data types and has a lot of in-built functions. ClickHouse has two different parsers built in it to process the SQL statements. A full SQL parser (a recursive descent parser) and the data format parser (a fast stream parser) are available in the application. The full SQL parser is used all the time except for `INSERT` queries to avoid problems with large `INSERT` queries. ClickHouse SQL is case insensitive. Any number of space symbols (space, tab, line feed, CR, and form feed) are allowed in a query, including at the beginning and the end of a query. An SQL statement may have the following elements:

Keywords

Keywords consist of words that are part of programming syntax. Keywords are not reserved in ClickHouse (such as `SELECT`, `INSERT`, `CREATE`, and so on).

Identifiers

Identifiers are the names of the database, tables, schema, or columns/aliases. An identifier cannot be a keyword unless they are enclosed by double quotes or backticks (for example, `SELECT` is an invalid identifier, although `SELECT` is valid).

Clauses

Clauses are part of the SQL statement and they are used to specify the conditions using which the data is retrieved from the table.

```
SELECT * FROM table WHERE x=1;
```

In this SQL statement, a **WHERE** clause is used.

Expressions

Expressions are a combination of operators, values, and functions that evaluate to a Boolean, numeric, or date value. In the previous example, the results of rows evaluate the expression (x=1) to True.

Queries

Queries read and return data from the database based on the criteria specified (for example, **SELECT * FROM table**).

Statements

Statements act upon the data stored in the database tables and can have a persistent effect on the data (for example, **DELETE FROM table WHERE x=10;**).

Comments

SQL or C styled (**/*** to ***/**) comments are allowed in ClickHouse SQL. SQL-style comments start with -- and continue to the end of the line.

Operators in ClickHouse SQL

The operators in ClickHouse SQL are transformed into their corresponding functions during the query parsing stage according to their priority, precedence, and associativity predefined in the application.

Arithmetic operators

Arithmetic operators are used to perform simple arithmetic operations. The following are the arithmetic operators available in ClickHouse SQL.

Syntax	Description	Function
a + b	Addition	plus(a, b)
a – b	Subtraction	minus(a, b)
a * b	Multiplication	multiply(a, b)
a / b	Division	divide(a, b)
a % b	Modulo	modulo(a, b)

Table 4.1: Arithmetic operators in ClickHouse SQL

Comparison operators

Comparison operators are used to compare the values, and the result of the operation can be True, False, or Unknown.

Syntax	Description	Function
a == b, a = b	Equal to	equals(a, b)
a != b, a <> b	Not equal to	notEquals (a, b)
a > b	Greater than	greater(a, b)
a < b	Less than	less(a, b)
a >= b	Greater than or equal to	greaterOrEquals (a, b)
a <= b	Less than or equal to	lessOrEquals(a, b)
a LIKE b	Like	like(a, b)
a NOT LIKE b	Not like	notLike(a, b)
a BETWEEN b AND c	Between	a >= b AND a <= c
a NOT BETWEEN b AND c	Not between	a < b OR a > c

Table 4.2: Comparison operators in ClickHouse SQL

Logical operators

Logical operators in SQL are used to make a decision based on multiple conditions and the conditions should be evaluated to True or False. Based on the value of the condition, the final value of the logical operator is decided.

Syntax	Description	Function
Not a	Negation	not(a)
a AND b	Logical AND	and(a, b)
a OR b	Logical OR	or(a, b)

Table 4.3: Logical operators in ClickHouse SQL

Checking for NULL

ClickHouse has `IS NULL` and `IS NOT NULL` operators to check for `NULL` types. Since ClickHouse does not have a Boolean data type, the values 0 and 1 are used to represent the False and True values, respectively.

IS NULL

The `IS NULL` operator returns `1` (True) if the value evaluated is NULL and `0` (False) for Nullable type values. For any other values (such as string, array, and so on), the operator always returns `0`.

IS NOT NULL

The `IS NOT NULL` operator is just the opposite of `IS NULL` operator.

Data types in ClickHouse

ClickHouse supports multiple types of data to accommodate numbers, strings, date and time, and Boolean data type. Apart from these regular data types, ClickHouse also supports array, tuples, and nested data structures.

We shall go through the most commonly used data types in this chapter.

Numeric data types

ClickHouse supports integers (both signed and unsigned), floating-point values and, signed fixed-point numbers (decimal).

Integers

The following are the types of fixed length signed and unsigned integers and their range:

- `Int8`: -128 to 127
- `Int16` :- -32768 to 32767
- `Int32`: -2147483648 to 2147483647
- `Int64`: -9223372036854775808 to 9223372036854775807
- `Int128`: -170141183460469231731687303715884105728 to 170141183460469231731687303715884105727
- `Int256`: - 57896044618658097711785492504343953926634992332820282019728792003956564819968 to 57896044618658097711785492504343953926634992332820282019728792003956564819967

- **UInt8**: 0 to 255
- **UInt16**: 0 to 65535
- **UInt32**: 0 to 4294967295
- **UInt64**: 0 to 18446744073709551615
- **UInt256**: 0 to 115792089237316195423570985008687907853269984665640564039457584007913129639935

Floats

ClickHouse supports Float32 and Float64 to store and process fractional numbers (for example, 3.14 and 1.414). Computations with floating-point numbers are prone to rounding errors. In addition to finite fractional numbers supported in standard SQL, ClickHouse supports other categories of floating-point numbers such as infinity (both positive and negative) and not-a-number (nan).

```
SELECT toFloat32(20);
```

The preceding query will output **20.0**.

inf and -inf

ClickHouse supports datatype to represent both the positive and negative infinity.

```
SELECT 3.14/0;
SELECT -3.14/0;
```

The output will be **inf** and **-inf**, respectively.

nan

ClickHouse supports nan value to represent not-a-number.

```
SELECT 0/0;
```

The preceding SQL statement will return **nan**.

Decimal

Decimals are signed fixed-point numbers. They preserve precision while performing mathematical operations such as addition, subtraction, and multiplication. For division, the least significant digits are discarded. Decimal can be of Decimal32(S), Decimal64(S), and Decimal128(S) type. S is a parameter that determines the number of decimal digits that are allowed in fraction. The decimal value ranges are given as follows:

- **Decimal32(S)**: -1 * 10^(9 - S) to 1 * 10^(9 - S)
- **Decimal64(S)**: -1 * 10^(18 - S) to 1 * 10^(18 - S)

- **Decimal128(S)**: -1 * 10^(38 - S) to 1 * 10^(38 - S)
- **Decimal256(S)**: -1 * 10^(76 - S) to 1 * 10^(76 - S)

```
SELECT toDecimal128(20,6);
```

The preceding query will print **`20.000000`**.

Boolean

ClickHouse does not have a dedicated data type for Boolean values. For representing **`Boolean`** data type, ClickHouse uses **`UInt8`** datatype, and values 1 and 0 are used to represent True and False, respectively.

String

Unlike many other DBMS, which have multiple data types to represent a string and its various forms (such as **`character`**, **`text`**, **`blob`**, **`varchar`**, and so on), ClickHouse uses string data type replacing them all. String data type in ClickHouse can be of any length. Strings are not encoded and are stored as a set of bytes. Strings are enclosed in single quotes while querying as shown in the following statement:

```
SELECT 'Hello world!';
```

FixedString

This data type is used to represent a string of fixed length (N Bytes). If the values are less than N bytes at the time of inserting the data in the table, ClickHouse will automatically add Null bytes and store them. If the length is greater than N bytes, an exception is thrown.

At the time of querying, if the search string is shorter than the defined length (N bytes), the Null bytes should be included in the query to match the fixed string length. For example, if a column is defined as type **`FixedString(3)`** and named a, to search for **`ch`**, it should be padded with one more Null byte and searched, as it is shorter than the defined length. So the query should be:

```
SELECT * FROM table WHERE a = 'ch\0';
```

Date

ClickHouse stores date as the number of days since 1970-01-01 (unix epoch). The upper limit for the date is until the year 2105. The date values are stored in two bytes. The following queries can convert the string representation to Date data type:

```
SELECT toDate('2020-01-01');
```

```
SELECT toDate('2107-01-01');
SELECT toDate('1969-12-31');
```

The first query will output the date that we have given and the next two queries will return 1970-01-01 since they are out of range for valid dates.

DateTime

DateTime type stores the timestamp and is expressed as the calendar date and time of the day. The resolution is 1 second and milliseconds are not supported in DateTime type. A time zone can also be defined for the entire column in the table.

By default, ClickHouse outputs DateTime as YYYY-MM-DD hh:mm:ss text format. If the time zone is not specified, the ClickHouse uses the server time zone. The following query converts the string representation of date and time into ClickHouse:

```
SELECT toDateTime('2020-01-01 00:00:00');
```

DateTime64

DateTime64 type stores the timestamp similar to the DateTime type. The difference is that the milliseconds are supported in the **`DateTime64`** type as shown in the following query. Millisecond precision and the time zone can also be defined for the entire column in the table:

```
SELECT toDateTime64('2020-01-01 00:00:00.0000', 4, ‘Europe/London’);
```

Arrays

Arrays are a collection of items. ClickHouse arrays can be created using the **`array()`** function or enclosing the array elements in square brackets as shown in the following statements:

```
SELECT array('a', 'b', 'c', NULL) AS x;
SELECT ['a', 'b', 'c', NULL] AS x;
SELECT [1,2,3,4,5.0, NULL] AS x;
SELECT array('a', 'b', 'c', 1, NULL) AS x;
```

The first two statements will return an output, whereas the last statement will throw an exception since the array elements are of incompatible datatype. The third statement will run fine since they are of the numeric datatype (compatible).

Tuples

Tuples are a collection of items like an array but can hold incompatible types as well. Tuples can be created using the **tuple()** function. The following example demonstrates the tuple creation with different data types:

```
SELECT tuple(1,2,'a', toDate('2020-01-01'), Null) AS x, toTypeName(x);
```

The statement will output the tuple and the data types of the tuple elements in separate columns.

Nested

Nested data structures are rare in relational databases, but ClickHouse supports nested data structures that can be quite useful while storing unstructured information in the structured database table. To illustrate the Nested data type, we shall create a table with a nested column, insert a sample data, and query the table to understand how the nested data is stored in the table as shown in the following statements:

```
CREATE TABLE nested_example
(
dt Date,
ID UInt32,
code Nested
(
code_id String,
count UInt32
)
)Engine=MergeTree(dt,ID,8192);
```

This table has a column with a nested data type called code. The nested column in turn has two columns called **code_id** and **count** in it. Let us insert sample data in it using the following query:

```
INSERT INTO nested_example VALUES ('2020-11-01', 77, ['1', '2', '3'],
[125, 654, 785]);
```

Once the data is inserted successfully, we shall query the table and see how the data is stored in the table with a nested data type. We can use the following query to see the inserted row:

```
SELECT  * FROM nested_example;
```

The following information is displayed in DBeaver's query results section (*Figure 4.1*):

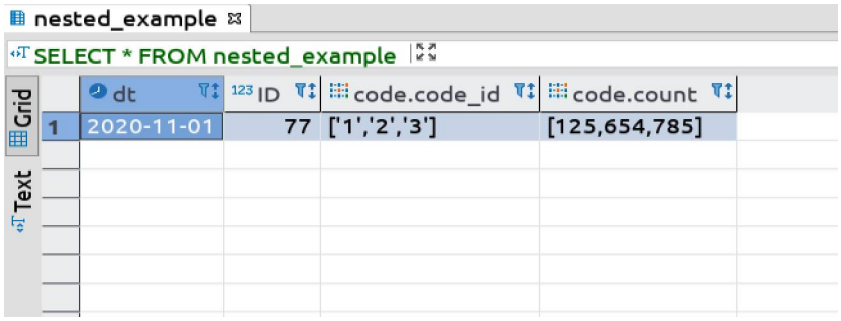


***Figure 4.1**: Query results with nested data types*

ClickHouse supports a single level of nesting by default. The column array of nested data structure should be of equal length. In this example, the length of the `code.code_id` should be equal to `code.count` within a single nested data structure. Execute the following queries that illustrate this:

```
INSERT INTO nested_example VALUES ('2020-11-02',76,['1','2'],[35, 49]);

INSERT INTO nested_example VALUES ('2020-11-02',76,['1','2'],[35, 49,
77]);
```

The first query will be executed successfully, whereas the second one will throw an exception. Now, execute the following query to see the table with added rows:

```
SELECT  * FROM nested_example;
```

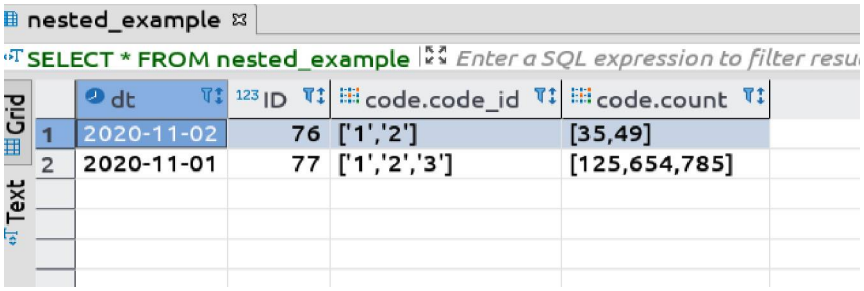


***Figure 4.2**: Query results with added rows and nested columns*

Enum

Enumerated datatype consists of named values like a dictionary and is declared with string (name) and a corresponding value (8 bit or 16-bit integer) for the name. The columns declared with enumerated data type can accept only the names or values declared in the type definition while creating the table. The purpose of this datatype

is to store and retrieve low-cardinal columns (columns with more number of unique elements) efficiently. Let us create a table with **enum** datatype:

```
CREATE TABLE enum_example
(col1 Enum('a'=1, 'b'=2))
ENGINE TinyLog;
```

This table has a single column with the **enum** datatype and can accept values that are either **a/b** and **1/2**. After the table is created, let us insert some sample data, as shown in the following statement:

```
INSERT INTO enum_example VALUES ('a'), ('b'), ('a'), ('b');
INSERT INTO enum_example VALUES (1), (2), (1), (1);
Once the data is inserted, let us query the data that was inserted:
SELECT * FROM enum_example;
```

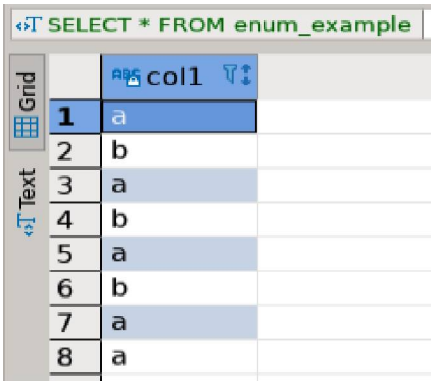


***Figure 4.3**: Query results on the table with Enum datatype*

LowCardinality

LowCardinality is not a data type by itself but a superstructure on top of other data types, which allows dictionary encoding for those data types. This is helpful in columns with low cardinality and tremendously increases the speed of data retrieval from such columns. LowCardinality is supported in Integers, Floats, Strings, Date, and DateTime types. The following syntax is used to create a table with a LowCardinality data type:

```
CREATE TABLE low_cardinality
(col1 LowCardinality(String))
ENGINE TinyLog;
```

ClickHouse SQL

We can work with the ClickHouse database using ClickHouse SQL. The ClickHouse dialect is quite similar to the standard SQL dialect commonly used in relational databases. Let us begin with querying the table, which we created while setting up the environment in *Chapter 3, Setting up the Environment*.

SELECT

The **SELECT** statement is used to select and retrieve data from the database. The syntax for the simplest **SELECT** statement is as follows:

```
SELECT column1, column2, …. FROM tablename;
```

Let us try this with the data that is already available in our ClickHouse installation. Execute the following query and observe the results:

```
SELECT "DateTime", Voltage FROM electricity.consumption;
```

Since DateTime is a keyword, we have to enclose them in double quotes while executing the query. This query will fetch the DateTime and Voltage column from the table. The **FROM** clause will specify the source from which the data is to be read and retrieved.

To select all the columns, * can be used in place of the column names as shown in the following query:

```
SELECT  *  FROM electricity.consumption;
```

LIMIT

To limit the number of rows in the result set, a **LIMIT** clause can be used. **LIMIT 10** will return the first 10 rows of the result and **LIMIT 10,5** will skip the first 5 rows and return the next 10 rows. Consider the following example:

```
SELECT  *  FROM electricity.consumption LIMIT 10;
```

DISTINCT

To select rows containing distinct values from a column, a **DISTINCT** clause can be used. Execute the following statements that has the **DISTINCT** clause.

```
SELECT  DISTINCT(Sub_metering_3) FROM electricity.consumption;
```

The first query returns distinct values from the column **Sub_metering_3**.

SAMPLE and OFFSET

ClickHouse allows sampling using the **SAMPLE** clause, which can make the query process a subset of data and return the results. Data to be considered for querying can be offset using the **OFFSET** clause. The syntax for the query with sampling and offset is as follows:

```
SELECT AVG(column_name) FROM table_name SAMPLE 8/10 OFFSET 1/2;
```

In the preceding query, the average value of a column from 80 percentage of data is taken from the second half of data in the table.

WHERE

To filter the result rows based on condition, the **WHERE** clause is used. The **WHERE** clause usually has one or more expressions with logical or comparison operators.

```
SELECT "Date" FROM electricity.consumption WHERE Voltage > 254.0;
```

The preceding query will return the dates on which the voltage value had exceeded 254.0.

The **WHERE** clause can be used in multiple expressions using **AND**/**OR** /**NOT** operators. The following example uses multiple expressions and an **AND** operator:

```
SELECT "Date", Global_intensity FROM electricity.consumption WHERE
Voltage >= 250.0 AND Global_intensity<4.6;
```

GROUP BY

The **GROUP BY** is generally used along with aggregate functions, and they are used to group rows with the identical values. Consider the following example:

```
SELECT AVG(Voltage), "Date" FROM electricity.consumption GROUP BY
"Date";
```

The preceding query will group the rows with similar values (in this case, date), and finds the average value of voltage for each group. So, the result will be the daily average of Voltage. The columns selected from the table should be either inside the aggregate function or in **GROUP BY** keys. Multiple columns can be used for grouping and the data is grouped based on the unique combination of these columns.

In the **SELECT** statement, the **GROUP BY** clause comes after the **WHERE** clause and the **ORDER BY** clause follows the **GROUP BY** clause (if the results need to be sorted).

ORDER BY

To sort the results from a **SELECT** query, the **ORDER BY** clause is used. The results can be ordered in ascending or descending order. In case the order is not specified, ClickHouse sorts them in ascending order by default.

```
SELECT AVG(Voltage), "Date" FROM electricity.consumption GROUP BY "Date"
ORDER BY "Date" DESC;
```

The preceding query will order the results in descending order.

CREATE

The **CREATE** statement can be used to create a database, table, user, and views. To create a database, the following syntax can be used:

```
CREATE DATABASE [IF NOT EXISTS] database_name [ON CLUSTER cluster_name]
[ENGINE = engine(...)]
```

To create a database using a native engine, the following query can be used:

```
CREATE DATABASE IF NOT EXISTS testing;
```

A new database will be created and will be displayed in DBeaver after refreshing. To create a table, the syntax would be:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER
cluster_name]
(
    column1 [datatype1] [DEFAULT|MATERIALIZED|ALIAS expr1] [compression_
codec] [TTL expr1],
    column2 [datatype2] [DEFAULT|MATERIALIZED|ALIAS expr2] [compression_
codec] [TTL expr2],
    ...
) ENGINE = engine
```

The **CREATE** statement will be extensively dealt in the next section, which covers different table engines and integrates with external sources. To create a simple table (using MergeTree engine), the following SQL statement can be used:

```
CREATE TABLE IF NOT EXISTS testing.test_table (
      ID String,
      Name String DEFAULT ID CODEC(LZ4),
      Quantity UInt32 CODEC(ZSTD),
      Total Float32 MATERIALIZED Quantity*3.14,
```

```
        Bill Float32 ALIAS Total*1,
        "Date" Date,
        CONSTRAINT quantity_constraint CHECK Quantity>0)
ENGINE = MergeTree()
PARTITION BY Date
ORDER BY (ID, Name, Date);
```

A new table named **test_table** is created under the database testing (created earlier). We have six columns (**ID**, **Name**, **Quantity**, **Total**, **Bill**, and **Date**) and the table is partitioned by date and ordered by **ID**, **Name**, and **Date**. While inserting the data, if a value for the Name column is not available, **ID** will be the default value for the name. The total column is materialized (calculated from other columns) and we cannot insert data manually using the **INSERT** statement. Bill is an alias and will not be stored in the disk. By default, ClickHouse uses LZ4 compression for columns, but other compression codecs are also supported. The **Name** column uses **LZ4**, whereas the **Quantity** column uses **ZSTD** compression.

Views in ClickHouse

Views consist of rows and columns just like tables, but they are based on other tables in the database and its contents are defined based on SQL query. ClickHouse supports normal as well as materialized views. Normal views don't store data and read from another table. A materialized view stores the data, which is processed by the **SELECT** query. A materialized view can be used to populate data from a temporary table into a permanent table (materialized views are widely used in populating data from a Kafka source and will be covered in the section Engines in ClickHouse) or to store computed or aggregated values from other tables. The following syntax is used to create a view in ClickHouse:

```
CREATE [MATERIALIZED] VIEW [IF NOT EXISTS] [database_name.]view_name
[TO[database_name.]name] [ENGINE = engine] [POPULATE] AS SELECT ...
```

The result set of the **SELECT** statement will be available in the view and if TO keyword is used, the data held in the view is inserted in the table. Let us create a normal view from the table **testing.test_table**.

```
CREATE VIEW testing.test_table_view AS SELECT ID,Name,"Date" FROM
testing.test_table;
```

The newly created view will have the columns **ID**, **Name**, and **Date**, and any data inserted in the **testing.test_table** will be reflected in the newly created view. To create a materialized version of the view, the following query can be used:

```
CREATE MATERIALIZED VIEW testing.test_table_materialized_view
```

```
ENGINE = MergeTree()
ORDER BY  (ID, Name, Date)
AS SELECT ID,Name,"Date" FROM testing.test_table;
```

INSERT INTO

As we have seen in the earlier examples, the **`INSERT INTO`** statement is used to insert new records in the table. The syntax for the **`INSERT INTO`** statement is as follows:

```
INSERT INTO [database_name.]table_name [(column1, column2, column3)]
VALUES (value11, value12, value13), (value21, value22, value23), ...
```

The elements enclosed in square braces are optional. To insert a record in the **`testing.test_table`**, which was created earlier, the following query can be used:

```
INSERT INTO testing.test_table (ID, Name, Quantity, "Date") VALUES
('123a', 'John', 5, '2020-09-05'), ('123b', 'Jimmy', 7, '2020-09-05');
```

Once the data is successfully inserted, we shall query the data from the table and the views associated with the table:

```
SELECT * FROM testing.test_table;
```

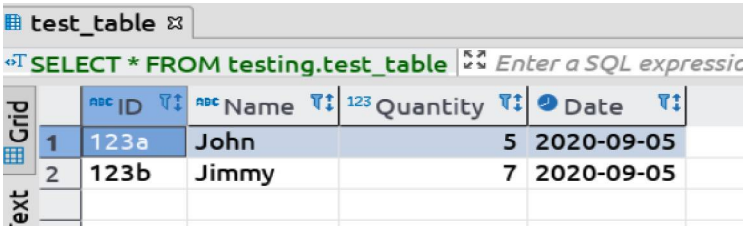


Figure 4.4: Query results showing the data that was inserted

We can see that, although we have declared columns named Total and Bill, they aren't displayed. Let us execute the following query and view the result:

```
SELECT Total, Bill FROM testing.test_table;
```

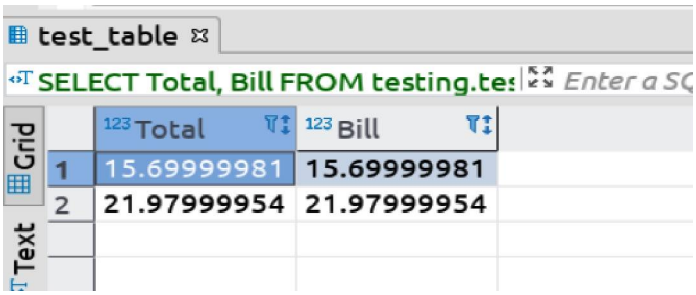


Figure 4.5: Query results showing the data from materialized and alias column

Let us query the views created based on this table. Execute the following queries and see the results. Replace the view name to **test_table_materialized_view** to fetch the results from the materialized view, as shown in the following screenshot:

```
SELECT * FROM testing.test_table_view;
```

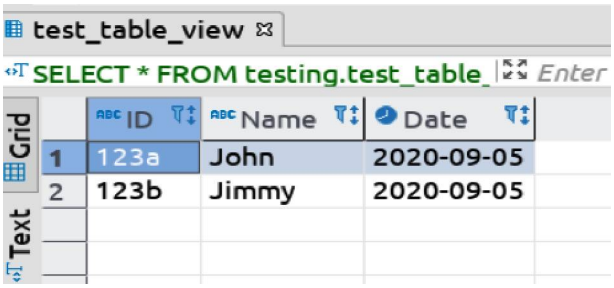


***Figure 4.6**: Query results showing the data from a view*

DROP

The **DROP** statement is used to drop an existing entity in ClickHouse, which could be a database, table, user, view, dictionary, and so on. The following syntax is used to drop a table:

```
DROP TABLE [IF EXISTS] [database_name.]table_name [ON CLUSTER cluster]
```

The following syntax can be used to drop a database in ClickHouse:

```
DROP DATABASE [IF EXISTS] database_nameb [ON CLUSTER cluster]
```

The **DROP** statements should be used with utmost caution since accidental usage may cause loss of data. **DROP** statements can also be used to drop user and user roles.

ALTER

The **ALTER** statements are used to modify the table and the data in the table. Using the **ALTER** statement, it is possible to manipulate the columns, update, and delete data in the tables.

Although OLTP-oriented databases support update and delete operations via **UPDATE** and **DELETE** statements, ClickHouse has recently added the functionality for updating and deleting the records via the **ALTER** statement.

Columns can be added, deleted, and cleared of values stored; and the datatype and default expressions can be modified using **ALTER** statements. The following syntax can be used to altera column in a ClickHouse table:

```
ALTER TABLE database_name.table_name [ON CLUSTER cluster]
ADD|DROP|CLEAR|COMMENT|MODIFY column_name
```

Let us create a column on the **test_table**, clear the default values, and modify the column type. Execute the following queries to add columns via the **ALTER** statement:

```
ALTER TABLE testing.test_table ADD COLUMN new_column1 Nullable(String)
DEFAULT toString('default1');

ALTER TABLE testing.test_table ADD COLUMN new_column2 Nullable(String);

SELECT * FROM testing.test_table;
```

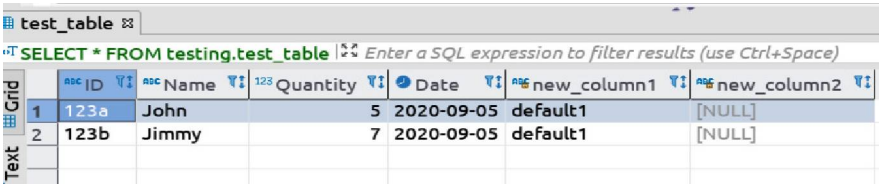


Figure 4.7: Newly added columns with the default values

The statements should add two new columns of string data type, and fill them with the specified default values. Now, let us add another record to the table and clear the contents of one of the columns that was just added.

```
INSERT INTO testing.test_table (ID, Name, Quantity, "Date", new_column1,
new_column2) VALUES ('123c', 'Jack', 5, '2020-09-05', 'Jack1', 'Jack2');

SELECT * FROM testing.test_table;
```

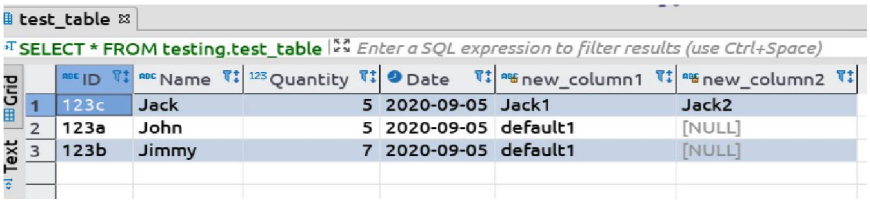


Figure 4.8: Newly added records

The following statement can be used to clear the contents of a column:

```
ALTER TABLE testing.test_table CLEAR COLUMN new_column2 IN PARTITION
tuple('2020-09-05');
```

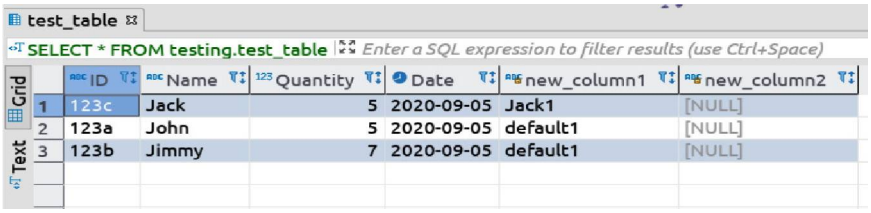


Figure 4.9: Cleared values in new_column2

The partition information is mandatory here and it is not possible to clear the column contents without partition information. This table is partitioned by the **Date** column (information is available in the **CREATE** statement section in this chapter) and the partition values are required to clear the columns. To drop (delete) the **new_column2** from the table, the following SQL statement can be used:

```
ALTER  TABLE testing.test_table DROP COLUMN new_column2;

SELECT * FROM testing.test_table;
```

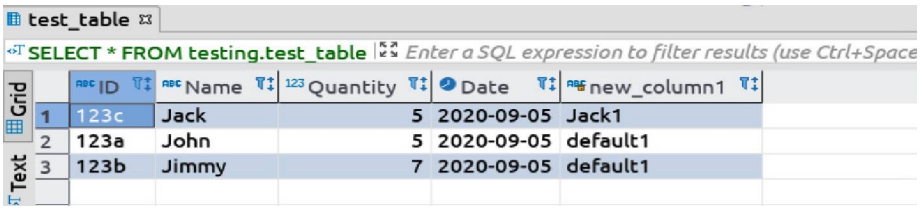


Figure 4.10: test_table after dropping a column

```
ALTER TABLE testing.test_table MODIFY COLUMN IF EXISTS Quantity Float64
DEFAULT -1;
```

This will change the **Quantity** column to Float64, which was Int32 earlier. Let us convert back to Int32 type again, as shown in the following statement:

```
ALTER TABLE testing.test_table MODIFY COLUMN IF EXISTS Quantity Float32
DEFAULT -1;
```

Updates and deletes

ClickHouse supports updates and deletes data via the **ALTER** statement. These operations are batch operations and are performed asynchronously. These operations are called mutations in ClickHouse. The following syntax can be used for updating specific values in rows in the ClickHouse table:

```
ALTER TABLE [database_name.]table_name UPDATE col1 = value1 [, ...]
WHERE expression;
```

The following statement can update the values in existing records:

```
ALTER TABLE testing.test_table UPDATE new_column1 = 'Jimmy Anderson'
WHERE Name='Jimmy';
```

```
SELECT * FROM testing.test_table;
```

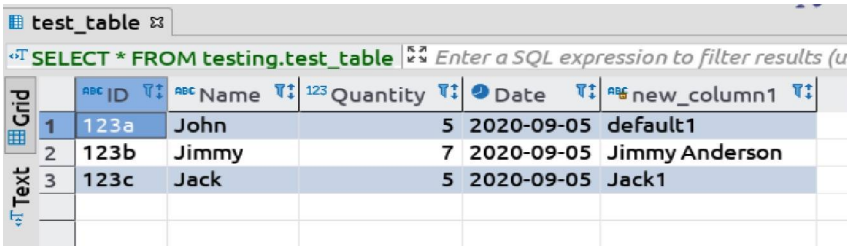


Figure 4.11: Updated values in new_column1

The following syntax can be used for deleting records from a ClickHouse table:

```
ALTER TABLE [database_name.]table_name DELETE WHERE expression;
```

The following syntax is for deleting rows based on the **WHERE** clause:

```
ALTER TABLE testing.test_table DELETE WHERE Name='Jack';
SELECT * FROM testing.test_table;
```

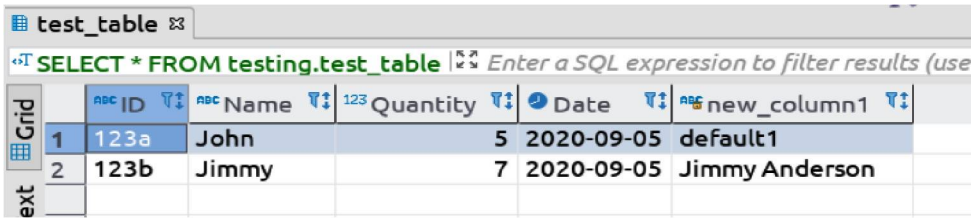


Figure 4.12: test_table after deleting the record

The preceding statement will delete the rows matching the **WHERE** clause (rows where the **Name** column has Jack as values in it).

SHOW

The **SHOW** statement can be used to display the information from ClickHouse, which includes the list of databases, tables, processes running, and the query used to create the table. The following syntax is to display the query used to create a table in ClickHouse:

```
SHOW CREATE [TEMPORARY] [TABLE|DICTIONARY] [database_name.]table_name
SHOW CREATE TABLE testing.test_table;
```

The following query can be used to show the databases and tables in ClickHouse:

```
SHOW DATABASES;
SHOW TABLES [FROM | IN database_nameb] [LIKE '<pattern>' | WHERE filter_
expression];
```

```
SHOW TABLES FROM testing WHERE name LIKE '%view%';
```

To show the process that is running, the following query can be used:

```
SHOW PROCESSLIST;
```

RENAME

As the name suggests, this statement is used to rename the tables. The syntax for the rename query is as follows:

```
RENAME TABLE [database_name1.]table_name1 TO database_name2.]table_name2
[ON CLUSTER cluster]
```

USE

The **USE** statement allows us to choose the database for the session. The database set to be used via the **USE** statement will be used for searching tables if the database is not explicitly mentioned in the query. The syntax is:

```
USE database_name;
```

SQL Joins in ClickHouse

The joins are used to combine two or more tables based on a related column in these tables using the **JOIN** clause. The following joins are supported in ClickHouse:

- Inner join
- Left join
- Right join
- Full join
- Cross join

Joins can be performed on more than two tables. For the sake of simplicity, the examples here are restricted to two tables. Let us begin by creating a sample database and tables, to begin with. Execute the following SQL statements to create and insert sample records in new tables meant for illustrating joins in ClickHouse:

```
CREATE database joins_example;
CREATE TABLE joins_example.customers(
customer_id String,
name String,
city String,
birth_year String)
```

```
ENGINE = MergeTree()
PRIMARY KEY customer_id
ORDER BY tuple(customer_id);

CREATE  TABLE joins_example.orders(
order_id String,
customer_id String,
quantity UInt64,
bill_amount Float64)
ENGINE = MergeTree()
PRIMARY KEY order_id
ORDER BY tuple(order_id);

INSERT INTO joins_example.customers VALUES ('1a', 'Chloe', 'Manchester',
'1982'),
('1b', 'Richard', 'Leeds', '1996'),
('1c', 'Emily', 'London', '1963'),
('1d', 'William', 'Bristol', '1974'),
('1e', 'Henry', 'Leeds', '1992'),
('1f', 'Philip', 'Liverpool', '1987'),
('1g', 'Lisa', 'Southampton', '1979'),
('2a', 'Stephen', 'Norwich', '1979');

INSERT INTO joins_example.orders VALUES ('a21', '1a', 7, 35.87),
('a21', '1a', 105, 852.77),
('a31', '1b', 12, 65.85),
('a27', '1c', 45, 325.68),
('a22', '1d', 87, 687.52),
('b21', '1e', 72, 602.32),
('g71', '1g', 32, 258.69),
('h22', '1g', 23, 212.74),
('q87', '1h', 49, 423.12),
('q87', '1i', 55, 487.25),
('q87', '1j', 68, 521.15);
```

Inner join

Inner join query joins and returns records combined from both the tables, where the join condition is satisfied. The query compares the rows with both the tables, finds the rows that satisfied the join condition, and combines them into single result set.

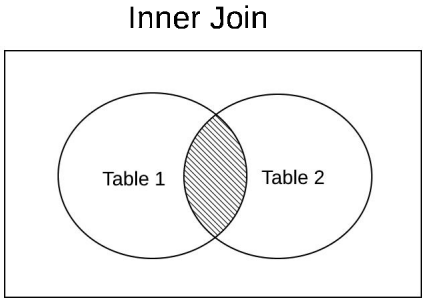


***Figure 4.13**: Inner join between two tables*

The following query performs an inner join on the recently created tables, based on the **`customer_id`** column:

```
SELECT orders.order_id, orders.quantity, customers.name, customers.
birth_year
FROM joins_example.orders as orders
INNER JOIN joins_example.customers as customers
ON orders.customer_id = customers.customer_id;
```

orders

SELECT orders.order_id, orders.quantity, Enter a SQL expression to f

Grid | Text

	order_id	quantity	name	birth_year
1	a21	7	Chloe	1982
2	a21	105	Chloe	1982
3	a22	87	William	1974
4	a27	45	Emily	1963
5	a31	12	Richard	1996
6	b21	72	Henry	1992
7	g71	32	Lisa	1979
8	h22	23	Lisa	1979

***Figure 4.14**: Result set for inner join performed on orders and customers table*

Left join

In left join, the query joins and returns all the records from the left table, the matched values from the right table, and in case there are no matches in the right table, Null values are returned for the columns from the right table.

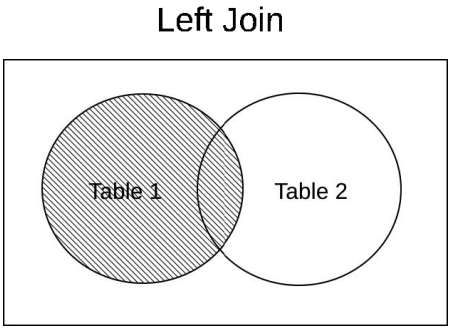


Figure 4.15: Left join between two tables

The following query performs a left join on the two tables based on the **customer_id** column:

```
SELECT orders.order_id, orders.quantity, customers.name, customers.
birth_year
FROM joins_example.orders as orders
LEFT JOIN joins_example.customers as customers
ON orders.customer_id = customers.customer_id;
```

orders

SELECT orders.order_id, orders.quantity, Enter a SQL expression to filte

	order_id	quantity	name	birth_year
1	a21	7	Chloe	1982
2	a21	105	Chloe	1982
3	a22	87	William	1974
4	a27	45	Emily	1963
5	a31	12	Richard	1996
6	b21	72	Henry	1992
7	g71	32	Lisa	1979
8	h22	23	Lisa	1979
9	q87	49		
10	q87	55		
11	q87	68		

Figure 4.16: Result set for left join performed on orders and customers table

Right join

Right join is similar to left join, which joins and returns all the records from the right table, the matched values from the left table, and in case there are no matches in the left table, Null values are returned for the columns from the left table.

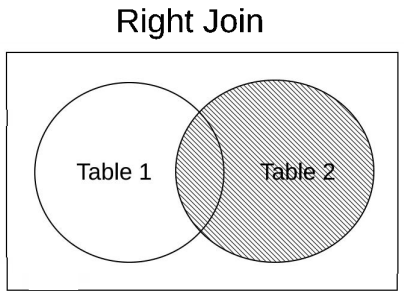


***Figure 4.17**: Right join between two tables*

The following query performs a right join on the two tables based on the **`customer_id`** column:

```
SELECT orders.order_id, orders.quantity, customers.name, customers.
birth_year
FROM joins_example.orders as orders
RIGHT JOIN joins_example.customers as customers
ON orders.customer_id = customers.customer_id;
```

orders

SELECT orders.order_id, orders.quantity, Enter a SQL expression to filter resu

Grid | Text

	order_id	quantity	name	birth_year
1	a21	7	Chloe	1982
2	a21	105	Chloe	1982
3	a22	87	William	1974
4	a27	45	Emily	1963
5	a31	12	Richard	1996
6	b21	72	Henry	1992
7	g71	32	Lisa	1979
8	h22	23	Lisa	1979
9		0	Philip	1987
10		0	Stephen	1979

***Figure 4.18**: Result set for Right Join performed on orders and customers table*

Full join

Full join will return the joined records from both the tables and the missing matches from both the table are filled with Null.

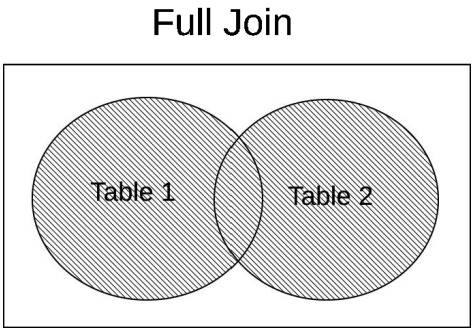


Figure 4.19: Full join between two tables

The following query performs a full join between the two tables:

```
SELECT orders.order_id, orders.quantity, customers.name, customers.
birth_year
FROM joins_example.orders as orders
FULL JOIN joins_example.customers as customers
ON orders.customer_id = customers.customer_id;
```

unknown

SELECT orders.order_id, orders.quantity — Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text | Record

	order_id	quantity	name	birth_year
1	a21	7	Chloe	1982
2	a21	105	Chloe	1982
3	a22	87	William	1974
4	a27	45	Emily	1963
5	a31	12	Richard	1996
6	b21	72	Henry	1992
7	g71	32	Lisa	1979
8	h22	23	Lisa	1979
9	q87	49		
10	q87	55		
11	q87	68		
12		0	Philip	1987
13		0	Stephen	1979

Figure 4.20: Result set for Full Join performed on orders and customers table

Cross join

Cross join performs the Cartesian product of both the tables involved. Each row from the left table is matched with the rows from the right table and the result set is returned. In most cases, a cross join may not produce any meaningful results.

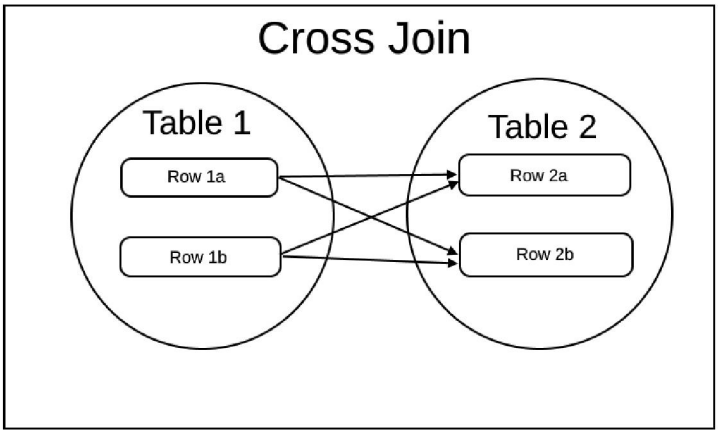


***Figure 4.21**: Cross join between two tables*

```
SELECT * FROM joins_example.orders
CROSS JOIN joins_example.customers;
```

orders CROSS

SELECT * FROM joins_example.orders CR | Enter a SQL expression to filter results (use Ctrl+Space)

	order_id	customer_id	quantity	bill_amount	customers.customer_id	name	city	birth_year
1	a21	1a	7	35.87	2a	Stephen	Norwich	1979
2	a21	1a	7	35.87	1a	Chloe	Manchester	1982
3	a21	1a	7	35.87	1b	Richard	Leeds	1996
4	a21	1a	7	35.87	1c	Emily	London	1963
5	a21	1a	7	35.87	1d	William	Bristol	1974
6	a21	1a	7	35.87	1e	Henry	Leeds	1992
7	a21	1a	7	35.87	1f	Philip	Liverpool	1987
8	a21	1a	7	35.87	1g	Lisa	Southampton	1979
9	a21	1a	105	852.77	2a	Stephen	Norwich	1979
10	a21	1a	105	852.77	1a	Chloe	Manchester	1982
11	a21	1a	105	852.77	1b	Richard	Leeds	1996
12	a21	1a	105	852.77	1c	Emily	London	1963
13	a21	1a	105	852.77	1d	William	Bristol	1974

***Figure 4.22**: Result set (partial) for cross Join performed on orders and customers table*

Union

ClickHouse supports the **`UNION ALL`** clause, which can be used to combine the result set from multiple **`SELECT`** statements. ClickHouse supports only **`UNION ALL`** and regular **`UNION`** (that return only distinct values) is not directly supported. However, it can be achieved by using the **`DISTINCT`** clause in the subqueries.

The columns in the **SELECT** statements must be of the same data type and the same number and in the same order as they appear in the query. The following query combines the **customer_id** and **order_id** from both the tables:

```
SELECT order_id as id FROM joins_example.orders
UNION ALL
SELECT customer_id as id FROM joins_example.customers;
```

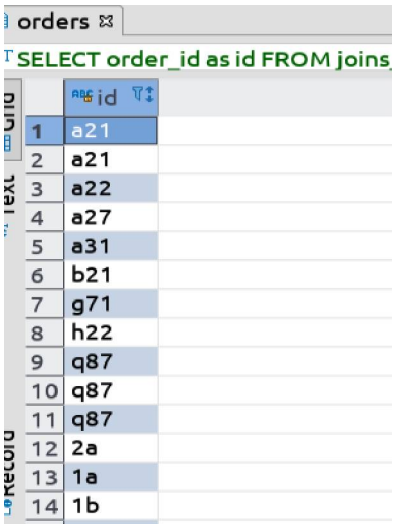


***Figure 4.23**: Result set (partial) for UNION ALL clause on order_id and customer_id from the respective tables*

Conclusion

In this chapter, we have learned ClickHouse SQL for performing common operations on data in ClickHouse. Despite being designed and developed for OLAP scenarios, ClickHouse supports almost all the operations and functionalities supported in a relational database including joins and unions.

There are even more SQL clauses and statements, which are used for tasks, such as managing user role-based access controls and administrative tasks. In the next chapter, we shall take a look at the in-built functions that are used to manipulate and process the data.

Points to remember

- ClickHouse supports arithmetic, logical, and comparison operators.
- ClickHouse supports numeric, string, and DateTime datatypes.

- ClickHouse supports arrays and tuples.
- The nested data type is supported in ClickHouse.
- ClickHouse has feature-rich SQL and supports SQL statements including `SELECT`, `INSERT`, `DROP`, `CREATE`, `ALTER`, and so on.
- ClickHouse also supports updates and deletes via `ALTER` statement.
- Joins and unions are also supported.

Multiple choice questions

1. **ClickHouse SQL is case-sensitive.**
   a) True
   b) False

2. **Keywords are reserved in ClickHouse.**
   a) True
   b) False

3. **The operators in ClickHouse SQL are transformed into their corresponding functions during the query parsing stage.**
   a) True
   b) False

4. **Range of UInt64 is _____.**
   a) 0 to 8446744073709551615
   b) 0 to 18446744073709551615
   c) 0 to 28446744073709551615
   d) 0 to 17446744073709551615

5. **Decimal data types preserve precision while performing operations such as addition, subtraction, and division.**
   a) True
   b) False

6. **ClickHouse uses UInt8 datatype and values 0 and 1 are used to represent True and False, respectively.**

   a) True

   b) False

7. **At the time of storing fixed string data, if the value to be stored is less than N (defined at the time of table creation) bytes, ClickHouse will throw an exception.**

   a) True

   b) False

8. **Resolution of DateTime data type is ____.**

   a) 1 millisecond

   b) 1 microsecond

   c) 1 second

   d) 1 minute

9. **An array can hold different but compatible data types, whereas tuples can hold incompatible data types in it.**

   a) True

   b) False

10. **The nested data type can hold columns of different data types.**

    a) True

    b) False

11. **ClickHouse supports up to _____ levels of nesting.**

    a) 3

    b) 4

    c) 2

    d) 1

12. **In the SELECT statement, the GROUP BY clause comes after the WHERE clause, and the ORDER BY clause follows the GROUP BY clause.**

    a) True

    b) False

13. **Materialized views don't store data and read from another table based on an SQL query.**
    a) True
    b) False

14. **ClickHouse supports synchronous updates and deletes.**
    a) True
    b) False

Answers

1. **b**
2. **b**
3. **a**
4. **b**
5. **b**
6. **a**
7. **b**
8. **c**
9. **a**
10. **a**
11. **d**
12. **a**
13. **b**
14. **b**

CHAPTER 5
SQL Functions in ClickHouse

ClickHouse has a rich set of in-built functions that can be used in SQL queries. ClickHouse has two different sets of functions, which the developers of ClickHouse classify as 'regular functions' and aggregate functions. The regular functions are provided with one or multiple rows of input and each row is processed independently and the results are returned. The aggregate functions process the entire set of rows and the results are dependent on the entire set.

In this chapter, we shall look at the commonly used regular functions, which will be referred as functions, from here on. The aggregate functions are covered in the next chapter.

Structure

In this chapter, we will discuss the following topics:

- Basics of ClickHouse functions
- Functions to convert data types
- Mathematical functions
- String functions
- DateTime functions
- Array functions

Objectives

This chapter aims to:

- Learn about in-built SQL functions in ClickHouse
- Get hands-on with in-built functions in ClickHouse

ClickHouse SQL functions

The ClickHouse SQL functions are reusable small programs that may have zero or more input parameters and return only one result. The ClickHouse SQL functions have the following properties:

- The functions return a single return value.
- The functions are incapable of changing the values of the arguments.
- The result data type is dependent on the data type of input values.
- ClickHouse uses strong typing. So, appropriate type conversion must be done by the users or applications before passing the values to the function.
- If any one of the arguments is Null, then the functions return Null as output.
- If invalid data is passed to the function, then the function throws an exception, the query is canceled and the error text is returned to the client.
- Constants as arguments and constant expressions are also supported.

Data type conversion

Type conversion functions are used to convert data from one data type to another in ClickHouse. Care must be taken by the user since the conversion of data type is prone to data loss. This is common when data of a larger type is converted to data of a smaller type (for example, from Float64 to Int32).

Integers

The following functions are available to convert to Int/UInt data type. There are separate functions available to convert to Null or Zero in case the conversion is not possible:

- `toInt[8/16/32/64](expression)`
- `toUInt[8/16/32/64]( expression)`
- `toInt[8/16/32/64]OrZero(expression)` – string as argument
- `toUInt[8/16/32/64]OrZero(expression)` – string as argument

- `toInt[8/16/32/64]OrNull(expression)` – string as argument
- `toUInt[8/16/32/64]OrNull(expression)` – string as argument

The following queries illustrate the usage and the result of the integer type conversion functions in SQL statements:

```
SELECT  toInt8(-322.8), toInt16(322.8), toInt32('-322'),
toInt64('322002220');
```

Figure 5.1: Converting to signed integer type

```
SELECT  toUInt8(322.8), toUInt16(322.8), toUInt32('322'),
toUInt64('322002220');
```

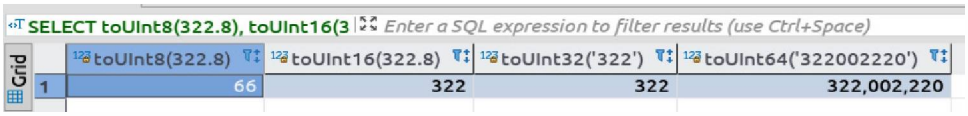


Figure 5.2: Converting to un-signed integer type

```
SELECT  toInt32OrZero('-322.0'), toInt64OrZero('322002220');
```

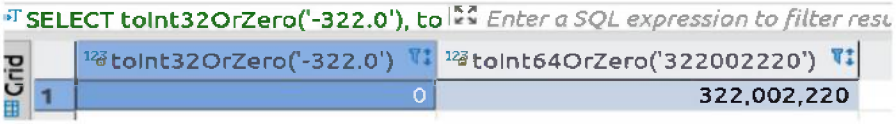


Figure 5.3: Converting to signed integer type or zero

```
SELECT  toUInt32OrZero('-322'), toUInt64OrZero('322002220');
```

Figure 5.4: Converting to unsigned integer type or zero

```
SELECT  toInt32OrNull('-322.0'), toInt64OrNull('322002220');
```

Figure 5.5: Converting to signed integer type or Null

```
SELECT  toUInt32OrNull('-322'), toUInt64OrNull('322002220');
```

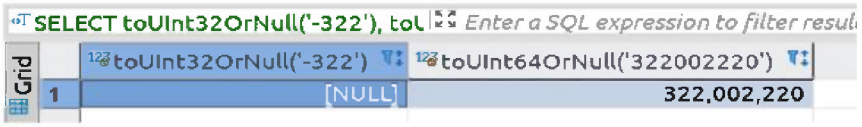


Figure 5.6: Converting to un-signed integer type or Null

An exception is thrown if an incompatible data type is passed or alphabets (and unsupported characters) are present in the strings that are provided as arguments.

Float

Similar to integer conversion functions, float conversion can be done using the following functions:

- `toFloat[32/64](expression)`
- `toFloatOrZero[32/64](expression)`
- `toFloatOrNull[32/64](expression)`

The following SQL statements illustrate the usage and the results of the float conversion functions:

```
SELECT  toFloat32(-322+225.67), toFloat64('322002220.87');
```

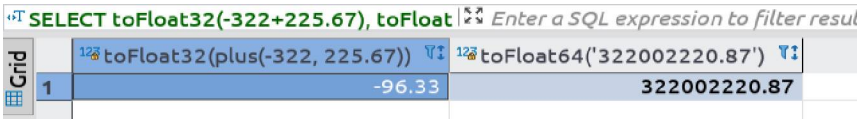


Figure 5.7: Converting to floating point data type

```
SELECT  toFloat32OrZero('-322.0a'), toFloat64OrZero('322002220.87');
```

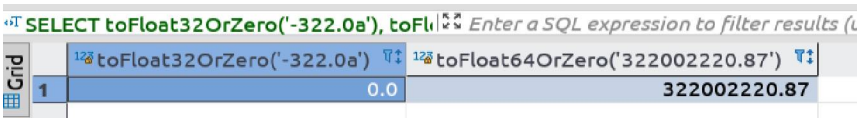


Figure 5.8: Converting to floating point data type or zero

```
SELECT  toFloat32OrNull('-322.0a'), toFloat64OrNull('322002220.87');
```

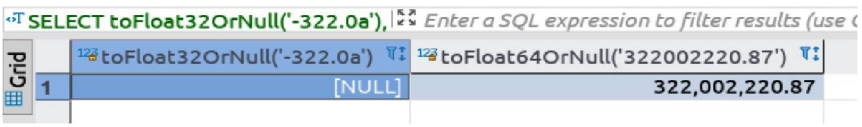


Figure 5.9: Converting to floating point data type or Null

Decimal

The decimal data type can be 32,64 or 128 bit in ClickHouse. The functions to convert to decimal data type are as follows:

- `toDecimal[32/64/128](value_to_be_converted, decimal_places)`
- `toDecimal[32/64/128]OrZero(value_to_be_converted, decimal_places)`
- `toDecimal[32/64/128]OrNull(value_to_be_converted, decimal_places)`

The first argument for these functions is either numbers or valid strings. The second argument is an integer that determines the number of decimal places. The following examples illustrate the usage of these functions:

```
SELECT  toDecimal32(3.1412, 8), toDecimal64('2548.87',8);
```

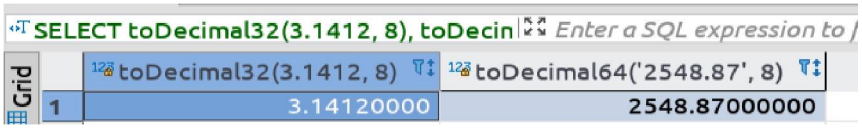


Figure 5.10: Converting to decimal data type

```
SELECT  toDecimal32OrZero('-322.0a',8), toDecimal64OrZero('207.32',12);
```

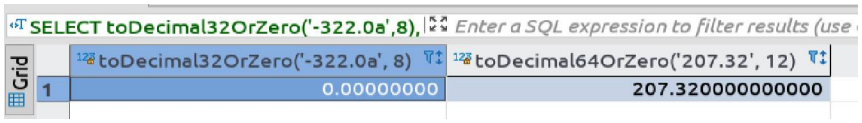


Figure 5.11: Converting to decimal data type or zero

```
SELECT  toDecimal32OrNull('-322.0a',8), toDecimal64OrNull('207.32',12);
```

Figure 5.12: Converting to decimal data type or Null

Note: To display floating point numbers properly in the DBeaver result editor, go to preferences > Data formats and check the native numeric and DateTime format options. This will force the DBeaver to display the data as stored in the database.**

Date and DateTime

As we know that, ClickHouse supports different data types for representing date and time (along with date). The type conversion function for these types are similar to each other. The following functions can be used to convert other formats to Date and DateTime types:

- `toDate(value)`
- `toDateTime(value)`
- `toDateOrZero(value)`
- `toDateTimeOrZero(value)`
- `toDateOrNull(value)`
- `toDateTimeOrNull(value)`

The `toDate()` and `toDateTime()` functions accept any non-zero integer, floats, or strings as input. The string input should be of supported Date/DateTime format (refer to documentation for further information on supported formats). The following statements illustrate the usage of these functions:

```
SELECT toDate('2020-08-01'), toDate(-100), toDate(100), toDate(100.5);
```

Figure 5.13: Converting to date type

```
SELECT  toDateTime('2020-08-01 00:00:00'), toDateTime(-2),
toDateTime(2), toDateTime(-2.2);
```

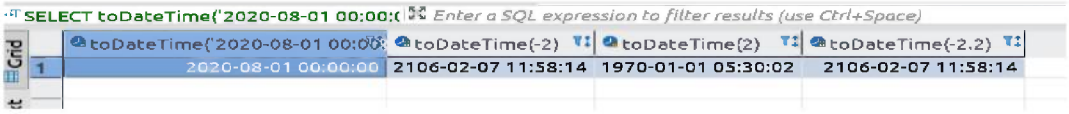


Figure 5.14: Converting to DateTime type

```
SELECT toDateOrZero('2020-08-01'), toDateOrZero('1847-08-01');
```

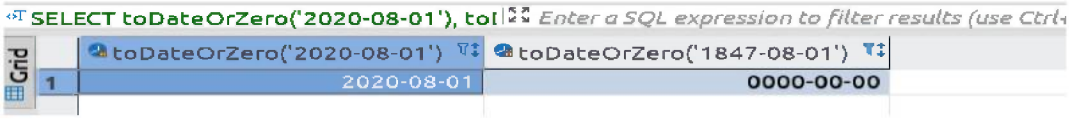


Figure 5.15: Converting to date type or zero

```
SELECT toDateTimeOrZero('2020-08-01 00:00:00'),
toDateTimeOrZero('1847-08-01');
```

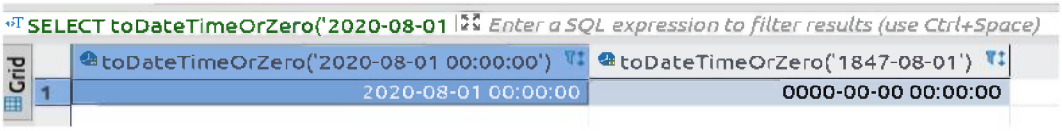


Figure 5.16: Converting to DateTime type or zero

String

In ClickHouse, it is possible to convert from multiple data types to string type. The following functions can be used to convert other data types to string type:

- **`toString(value)`**.
- **`toFixedString(value, length)`** – string as argument type and output type is fixed string with constant length.
- **`toStringCutToZero(value)`** – string or fixed string type as argument type.

The **`toFixedString()`** converts strings of arbitrary length to fixed-length string type. If the argument string is less than the specified fixed length, then the string is appended with Null bytes and if it is greater than the length specified, an exception is thrown. The **`toStringCutToZero()`** will truncate the string to the first zero bytes. The following examples illustrate the usage of the string conversion function:

```
SELECT  toString(77), toString(32.1), toString(toDate('2002-08-01'));
```

Figure 5.17: Converting to string type

```
SELECT toFixedString(toString(123), 5);
```

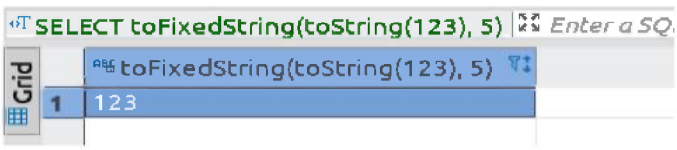


Figure 5.18: Converting to fixed string type

```
SELECT toStringCutToZero('abc\0def\0g');
```

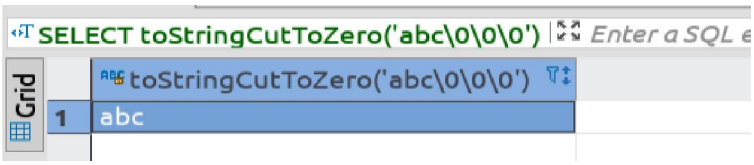


Figure 5.19: String truncated on the first zero byte

Working with numbers

Apart from the type conversion functions, ClickHouse supports in-built mathematical functions and a set of functions for rounding off the numbers.

Mathematical functions

Mathematical functions accept a numeric input (with `pi()` and `e()` functions being exceptions, which returns a constant) and return a number of Float64 types. The following examples illustrate the usage of the mathematical functions in ClickHouse:

```
SELECT e(), pi();
```

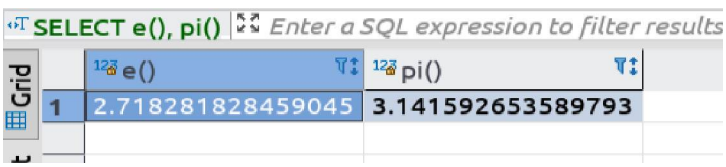


Figure 5.20: Euler's Number and pi constants

```
SELECT exp(0), exp2(3.14),  exp10(3.14);
```

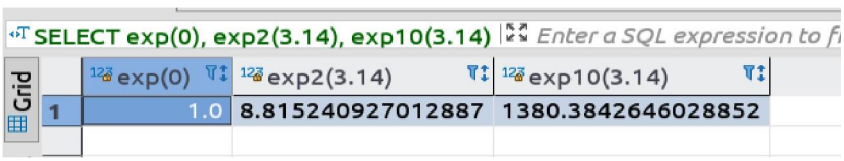


Figure 5.21: Exponents for different bases

```
SELECT log(1), ln(2.0), log2(3.14), log10(3.14);
```

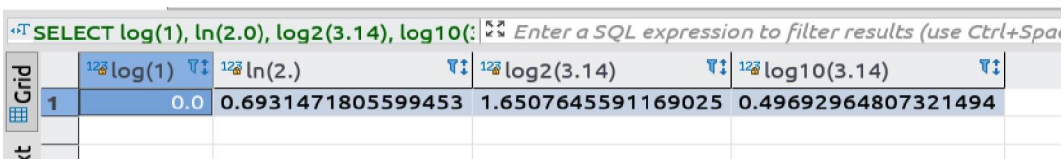


Figure 5.22: Logarithms for different bases

```
SELECT sqrt(2.0), cbrt(8.0), power(3,3);
```

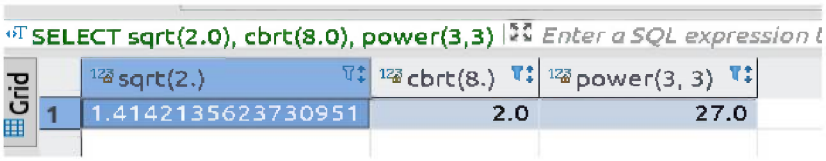


Figure 5.23: Square root, cube root and power functions

```
SELECT sin(90), cos(270), tan(45);
```

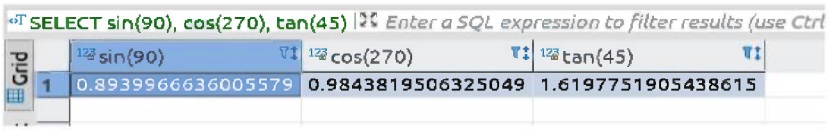


Figure 5.24: Trigonometric functions

The trigonometric functions take arguments in degrees.

```
SELECT asin(0.99), acos(0.2), atan(0.1);
```

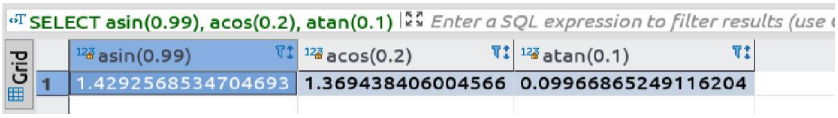


Figure 5.25: Inverse trigonometric functions

```
SELECT erf(3), erfc(3);
```

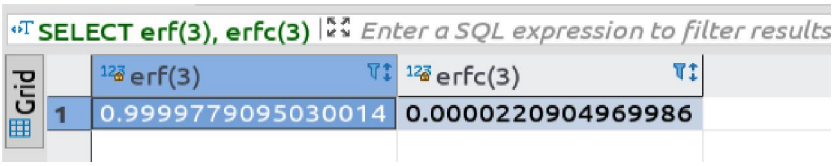


Figure 5.26: Error function and complementary error functions

```
SELECT tgamma(10), lgamma(10);
```

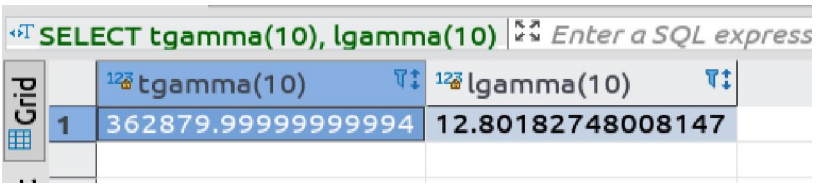


Figure 5.27: Gamma and Log gamma functions

Rounding functions

ClickHouse has in-built functions for rounding off the floating point numbers. The following functions can be used to perform rounding operations:

- `round(value, [decimal_places])`
- `ceil(value, [decimal_places])`
- `floor(value, [decimal_places])`

The round() function rounds off the number automatically to the nearest integer. The ceil() function rounds off to the nearest integer above its current value. The floor function rounds off to the nearest integer below its value. The number of decimal places is an optional argument and its implications can be found by modifying the sample queries given as follows:

```
SELECT round(10.5354), round(10.535, 2);
```

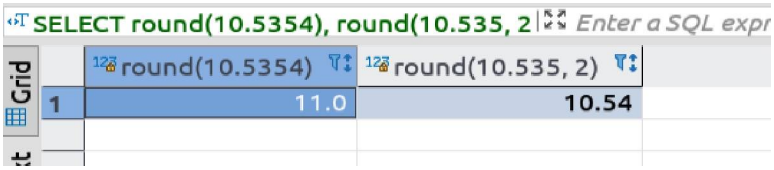


Figure 5.28: Rounding off using round function

```
SELECT floor(10.534), floor(10.535,1);
```

Figure 5.29: Rounding off using floor function

```
SELECT ceil(10.534), ceil(10.535,2);
```

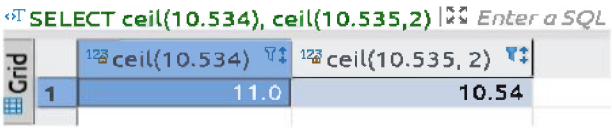


Figure 5.30: Rounding off using ceil function

Working with Date/DateTime

ClickHouse has numerous in-built functions for working with Date and DateTime data types and manipulating them. Except for `now()`, `today()`, and `yesterday()`,

all the other functions accept Date/DateTime as an argument, and in few functions, the time zone can also be passed, which is again purely optional. The following functions do not accept any arguments and their usage is illustrated in the following examples:

- `now()`: Returns current DateTime
- `today()`: Returns today's date
- `yesterday()`: Returns yesterday's date

```
SELECT now(), today(), yesterday();
```

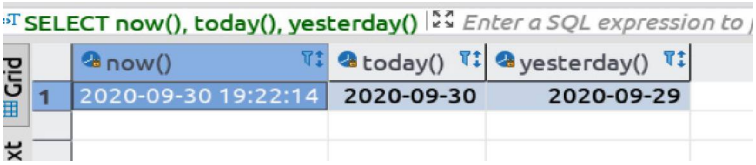


Figure 5.31: Date/DateTime functions without arguments

Converting to different time units

It is possible to extract the time units such as minutes, seconds, and hours or extract information like month, quarter, and year from DateTime in ClickHouse. The following example uses functions that help us to extract the year, quarter, month, and the week from the Date/DateTime:

```
SELECT toDateTime('2020-05-05 07:06:51') as ts, toYear(ts) as year,
toQuarter(ts) as quarter, toMonth(ts) as month, toWeek(ts) as week;
```

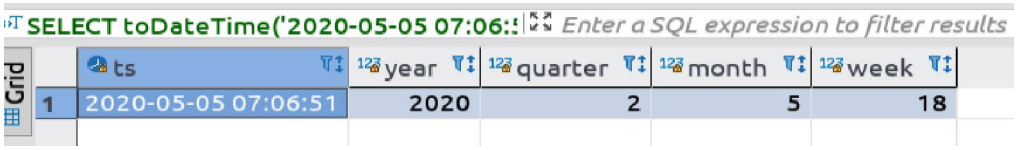


Figure 5.32: Date/DateTime functions to get the year, quarter, month, and week

The following is an example usage of functions to extract hours, minutes, and seconds from DateTime:

```
SELECT toDateTime('2020-05-01 02:35:44') as ts, toHour(ts) as hour,
toMinute(ts) as minutes, toSecond(ts) as seconds;
```

Figure 5.33: DateTime functions to extract hours, minutes, and seconds

The functions used in the following example convert the date to day of the week (1 – 7 (Monday to Sunday), day of the month (1-31), and day of the year (1-366)). These functions accept date as well as date–time data types as input argument data type.

```
SELECT toDate('2020-05-01') as dt, toDayOfWeek(dt) as day_week,
toDayOfMonth(dt) as day_month, toDayOfYear(dt) as day_year;
```

Figure 5.34: Date functions to extract the day of the week, month, and year

To convert the time zone of date time in ClickHouse, the **`toTimeZone()`** function can be used.

```
SELECT toDateTime('2020-05-01 02:35:44', 'US/Samoa') as samoa_ts,
toTimeZone(samoa_ts, 'Asia/Kolkata') as asia_ts;
```

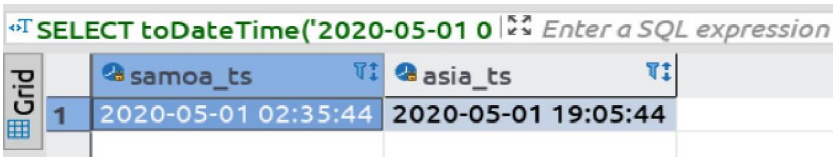


Figure 5.35: Date functions to convert time zone

`toUnixTimeStamp()` can be used to convert the ClickHouse DateTime format to Unix time format (number of seconds since Unix epoch, which is 00:00:00 on Jan 1, 1970).

```
SELECT toUnixTimestamp('2020-05-01 02:35:44');
```

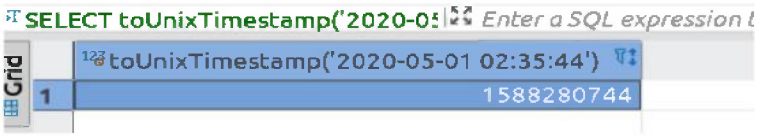


Figure 5.36: Converting to Unix timestamp

Rounding functions

Rounding functions for Date/DateTime data types are also available in ClickHouse. The rounding functions accept Date and/or DateTime data types and string representations should be converted before being passed as an argument to these functions. The following examples use the type conversion functions and their usage can be inferred from them:

```
SELECT toDateTime('2020-05-05 07:06:51') as ts, toStartOfYear(ts) as
year_start, toStartOfISOYear(ts) as start_iso_year, toStartOfQuarter(ts)
as quarter_start, toStartOfMonth(ts) as month_start;
```

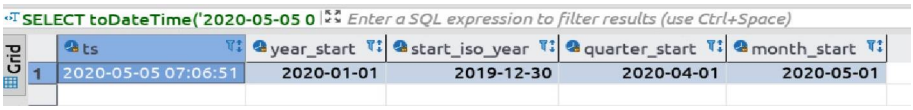


***Figure 5.37**: Rounding off the timestamp to year, quarter, and month*

```
SELECT toDateTime('2020-05-05 07:06:51') as ts, toMonday(ts) as nearest_
monday, toStartOfWeek(ts) as week_start;
```

***Figure 5.38**: Rounding off the timestamp to nearest Monday and starting day of the week (Sunday)*

```
SELECT toDateTime('2020-05-05 07:06:51') as ts, toStartOfDay (ts) as
day_start, toStartOfHour (ts) as hour_start, toStartOfMinute(ts) as
minute_start;
```

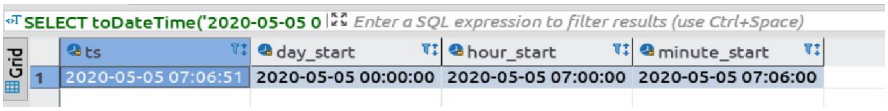


***Figure 5.39**: Rounding off the timestamp to start of the day, hour, and minute*

```
SELECT toDateTime('2020-05-05 07:16:51') as ts,
toStartOfFiveMinute(ts) five_mins, toStartOfTenMinutes(ts) as ten_mins,
toStartOfFifteenMinutes(ts) as fifteen_mins;
```

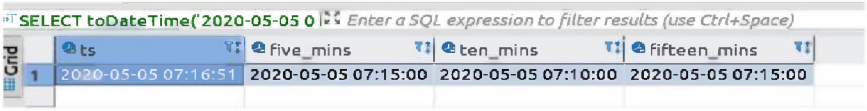


***Figure 5.40**: Rounding off the timestamp to nearest 5, 10, and 15 minutes interval*

To convert the Date/DateTime to an integer while still preserving human readability (unlike ISO format), the following functions can be used:

```
SELECT toDateTime('2020-05-05 07:06:51') as ts, toYYYYMM(ts) as YYYYMM,
toYYYYMMDD(ts)YYYYMMDD, toYYYYMMDDhhmmss(ts) as YYYYMMDDhhmmss;
```

***Figure 5.41**: Converting timestamp to integer format*

Date/DateTime arithmetic

ClickHouse supports functions to perform simple addition and subtraction operations on Date/DateTime data types. To get the date difference between two Date/DateTime values, **`dateDiff('unit', start, end, [timezone])`** function can be used. The first argument is the time unit in which the output is returned. The second and third arguments are the start and end Date/DateTime. Time Zone is an optional argument here.

```
SELECT dateDiff('hour', toDateTime('2020-02-08 10:00:00'),
toDateTime('2020-02-08 16:40:00')) as difference;
```

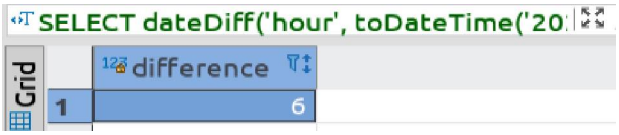


***Figure 5.42**: Date/Datetime difference*

The following time units can be passed as an argument to the **`dateDiff()`** function:

- **second**
- **minute**
- **hour**
- **day**
- **week**
- **month**
- **quarter**
- **year**

Subtracting and adding time units from Date/DateTime

The functions used to add and subtract time units accept two arguments. The first one is the Date/DateTime value on which the operation is performed and the time units to be added or subtracted.

Subtracting time units from Date/DateTime

```
SELECT toDateTime('2020-02-08 16:40:00') as ts, subtractYears(ts,7) as
sub_years, subtractMonths(ts,7) as sub_months, subtractWeeks(ts,7) as
sub_weeks, subtractDays(ts,7) as sub_days;
```

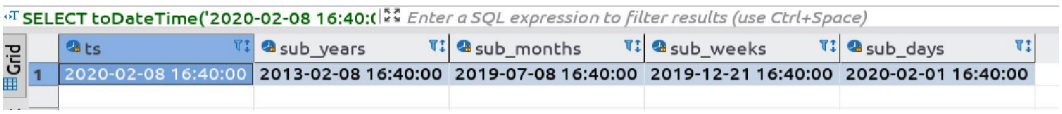


***Figure 5.43**: Subtracting time units from Date/DateTime*

```
SELECT toDate('2020-02-08') as ts, subtractHours (ts,7) as sub_hours,
subtractMinutes(ts,7) as sub_minutes, subtractSeconds(ts,7) as sub_
seconds;
```

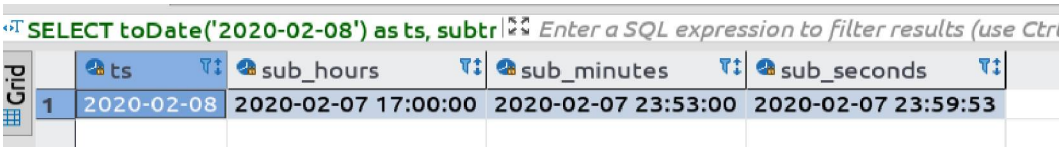


Figure 5.44: Subtracting time units from Date/DateTime

Adding time units from Date/DateTime

```
SELECT toDateTime('2020-02-08 16:40:00') as ts, addYears(ts,7) as
add_years, addMonths(ts,7) as add_months, addWeeks(ts,7) as add_weeks,
addDays(ts,7) as add_days;
```

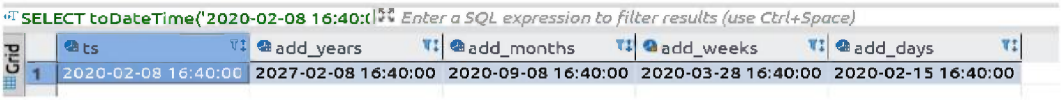


Figure 5.45: Adding time units to Date/DateTime

```
SELECT toDate('2020-02-08') as ts, addHours(ts,7) as add_hours,
addMinutes(ts,7) as add_minutes, addSeconds(ts,7) as add_seconds;
```

Figure 5.46: Adding time units to Date/DateTime

Date/DateTime formatting

The `formatDateTime()` function formats Date/DateTime to the user-specified string format. It is recommended to refer to the official documentation for a complete list of format modifiers.

```
SELECT formatDateTime(toDate('1996-11-01'), '%d,%m,%Y') as formatted_
date;
```

Figure 5.47: Formatting Date/DateTime

Working with strings

Strings are not encoded in ClickHouse and are stored as bytes. The following examples have functions to find the length of the strings in ClickHouse:

```
SELECT empty(''), empty('Hi!');
```

Figure 5.48: Checking for Empty/Non-empty strings

```
SELECT notEmpty(''), notEmpty('Hi!');
```

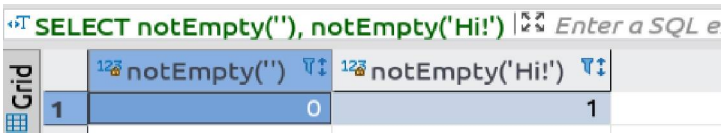


Figure 5.49: Checking for Empty/Non-empty strings

The following functions are used to find the length of the string. The `length()` function returns length in character bytes, whereas the rest of the functions returns in Unicode code points:

```
SELECT 'ŠĤb_123' as text, length(text) as length, char_length(text) as
char_length,lengthUTF8(text) as lengthUTF8, character_length(text) as
character_length;
```

Figure 5.50: Finding the length of the string

Case conversion

There are separate functions available in ClickHouse to convert the case of the string. The following example shows how to convert a string (with ASCII symbols) from upper or lower case:

```
SELECT 'Rain' as text, lower(text) as lower, upper(text) as upper;
```

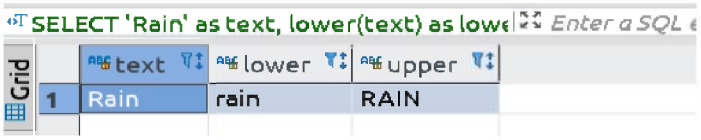


Figure 5.51: Converting ASCII symbols in a string to lower and upper case

```
SELECT '\x52\x61\x69\x6E' as text, lowerUTF8(text) as lower,
upperUTF8(text) as upper;
```

Figure 5.52: Converting Unicode to lower and upper case

String manipulation

ClickHouse has in-built functions to perform operations such as repetition, reversal, and concatenation.

```
SELECT 'air' as text, repeat(text,3) as repeat, reverse(text) as
reverse, concat(text, 'water') as concat;
```

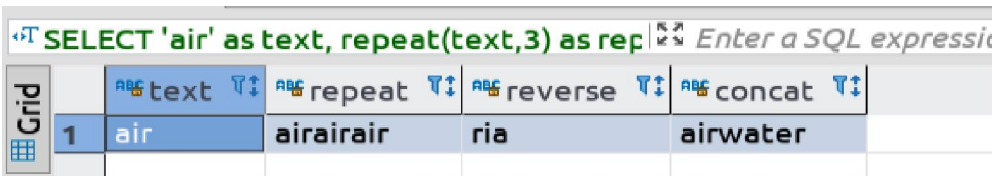


Figure 5.53: String manipulation in ClickHouse

To extract a substring from a long string, **`substring()`**/**`substr()`**/**`mid()`** function can be used. The first argument is the source string from which the substring is to be extracted, the second argument is the starting position in the source string from which the substring is extracted, and the third is the length of the substring to be extracted.

```
SELECT 'Manhattan' as text,  substring(text, 4, 3) as substring,
substr(text, 4, 3) as substr, mid(text, 4, 3) as mid;
```

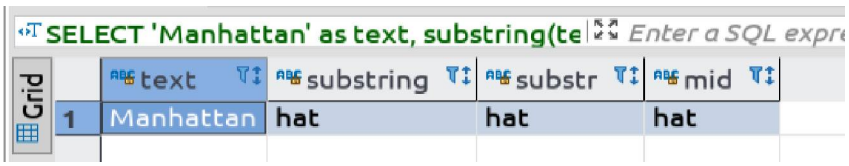


Figure 5.54: Getting sub-string from a source string

Removing the leading and trailing whitespaces (ASCII character 32) can be done using **`trimLeft()`**, **`trimRight()`**, and **`trimBoth()`** functions.

```
SELECT '      Hello      ' as txt, trimLeft(txt) as trim_left,
trimRight(txt) as trim_right, trimBoth(txt) trim_both;
```

Figure 5.55: Trimming whitespaces from a string

To remove any sequence of characters from the beginning to the end of the string, the **`trim()`** function can be used. So the source string and the characters to be trimmed are passed as arguments to this function.

```
SELECT trim(BOTH ' ' FROM '   Hello, world!   ') as trimmed;
```

Figure 5.56: Trimming whitespaces from a string

Trimming the leading and trailing characters can be achieved by using the **`LEADING`**/**`TRAILING`** option in the query instead of **`BOTH`**. To check if the string starts with a particular prefix or set of characters, the **`startsWith()`** function can be used.

```
SELECT 'Hello' as txt, startsWith(txt, 'H') starts_with_H,
startsWith(txt, 'A') starts_with_A;
```

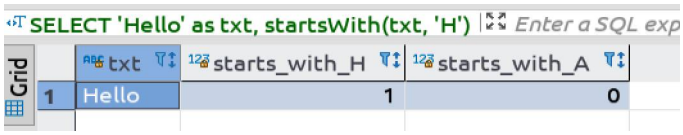


Figure 5.57: Checking if a string starts with a specific set of characters

Similarly, to find if a string ends with a specific set of characters, the **`endsWith()`** function can be used.

```
SELECT 'Hello' as txt, endsWith(txt, 'o') ends_with_o,  endsWith(txt,
'a') ends_with_a;
```

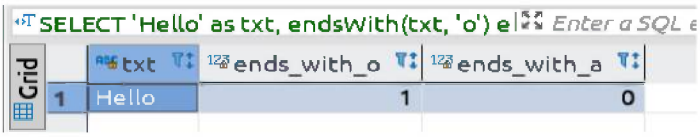


Figure 5.58: Checking if a string ends with a specific set of characters

Searching in strings

ClickHouse provides a wide variety of functions to perform search operations in strings. The search operations are case-sensitive by default. Separate functions are available for the Unicode and case insensitive search.

Searching for positions

The **`position()`** and **`multiSearchAllPositions()`** functions are used to find the position (byte number) of the substring in the source string. The first argument is the source string and the second argument is the substring in **`position()`** function or array of substrings in the **`multiSearchAllPositions()`** function. In case no match is found, 0 is returned.

```
SELECT 'She sells sea shells, on the sea shore!' as txt, position(txt,
'sea') as position, multiSearchAllPositions(txt, ['sea', 'shore']) as
multiple_positions;
```

Figure 5.59: Searching for substring positions in a string

For case-insensitive search, **`positionCaseInsensitive()`** function can be used. The UTF variants of these functions are **`positionUTF8()`** and **`positionCaseInsensitiveUTF8()`**, respectively.

For the second function, the case-insensitive counterpart is **`multiSearchAllPositionsCaseInsensitive()`**. The UTF variants for the **`multiSearchAllPositions()`** are **`multiSearchAllPositionsUTF8()`** and **`multiSearchAllPositionsCaseInsensitiveUTF8()`**, respectively.

Searching from a set of substrings

To find the position of the substring that occurs first from a set of substrings, **`multiSearchFirstPosition()`** can be used.

```
SELECT 'She sells sea shells, on the sea shore!' as txt,
multiSearchFirstPosition(txt, ['sea', 'She']) as first_pos;
```

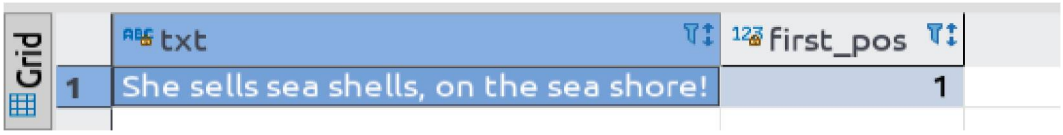


Figure 5.60: Finding the position of the first occurring substring from the source string

Since the substring **She** occurs before **sea**, the position of the first occurring substring is returned. **multiSearchFirstPositionCaseInsensitive()** is the case-insensitive version of this function.

UTF variants

Like other string search functions, the UTF variants for **multiSearchFirstPosition** functions are as follows:

- **multiSearchFirstPositionUTF8()**
- **multiSearchFirstPositionCaseInsensitiveUTF8()**

To find the index of the substring in an array of substrings that occur first in a source string, **multiSearchFirstIndex()** can be used.

```
SELECT 'She sells sea shells, on the sea shore!' as txt,
multiSearchFirstIndex(txt, ['shop', 'shore', 'sea']) as first_ind;
```

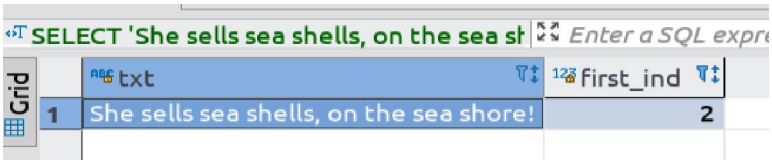


Figure 5.61: Finding the index of the first occurring substring in the source string

In this case, **shop** is absent in the source string, although **shore** and **sea** are present. As **shore** appears first in the substring array, although **sea** occurs first in the source string, the index of **shore** is returned. **multiSearchFirstIndexCaseInsensitive()** is the case-insensitive function of this version.

UTF variants

The UTF variants for **multiSearchFirstIndex** functions are:

- **multiSearchFirstIndexUTF8()**
- **multiSearchFirstIndexCaseInsensitiveUTF8()**

Searching for the occurrence of at least one substring

The function **multiSearchAny()** returns **1** if any one of the substring matches or else **0**. Index or position is not returned.

```
SELECT 'She sells sea shells, on the sea shore!' as txt,
multiSearchAny(txt, ['shop', 'shore', 'sea']) as search_any_yes,
multiSearchAny(txt, ['shop', 'share', 'see']) as search_any_no;
```

Figure 5.62: Searching for the occurrence of at least one substring in the source string

The case-insensitive variant is `multiSearchAnyCaseInsensitive()`.

UTF variants

The UTF variants for `multiSearchAny` functions are:

- `multiSearchAnyUTF8`
- `multiSearchAnyCaseInsensitiveUTF8`

Searching with regular expression patterns

There are functions implemented in ClickHouse to match `re2` regular expressions. A regular expression match considers string as a set of bytes.

match()

This function matches a regular expression pattern to the source string. It returns `1` if a match is found or `0` if a match is not found.

```
SELECT 'She sells sea shells, on the sea shore!' as txt, match(txt,
'he.') as matched, match(txt, '[\d]') as not_match;
```

Figure 5.63: Matching using regular expressions

multiMatchAny()

This function tries to match a set of regular expression patterns and returns `1` if any of the patterns match in the source string.

```
SELECT 'She sells sea shells, on the sea shore!' as txt,
multiMatchAny(txt,['[\d]', 'sea.']) as match, multiMatchAny(txt,['[\d]',
'john']) as no_match;
```

Figure 5.64: Matching any one regular expression pattern in the source string

multiMatchAllIndices()

This function matches a set of regular expression patterns and returns an array containing the index of the matching patterns in the set.

```
SELECT 'She sells sea shells, on the sea shore!' as txt,
multiMatchAllIndices(txt,['[\d]', 'sea.', 'she.']) as match;
```

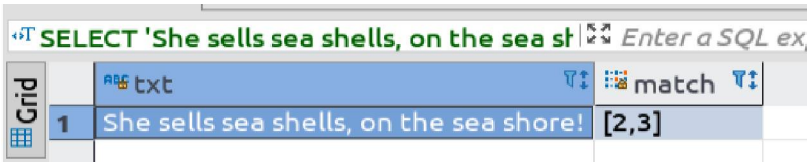


Figure 5.65: Matching multiple regular expression patterns in a source string

Matching simple regular expressions

To match simple regular expressions, **`like()`** and **`notLike()`** functions can be used.

```
SELECT 'She sells sea shells, on the sea shore!' as txt, like(txt,
'%sea%') as like, notLike(txt, '%sea%') as not_like;
```

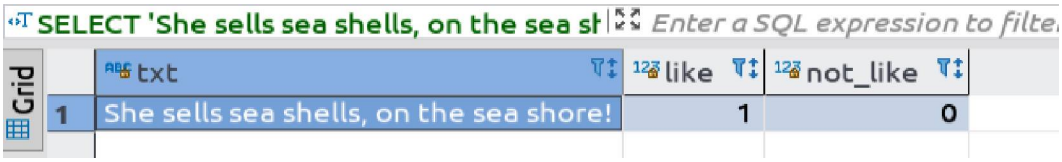


Figure 5.66: Matching simple regular expression pattern in a source string

Extracting substrings using regular expressions

Extracting substrings from source string using regex pattern can be achieved via **`extract()`** and **`extractAll()`** functions.

```
SELECT 'She sells sea shells, on the sea shore!' as txt, extract(txt,
's...s') as extract, extractAll(txt, 's...s') as extract_all;
```

Figure 5.67: Extracting pattern in source string using regular expressions

Replacing substrings from the source string

It is possible to search for substring and replace them in the strings either by searching for a substring directly or using regular expressions. These functions have three arguments (source string, replace pattern, and replace substring).

replaceOne() and replaceAll()

```
SELECT 'She sells sea shells, on the sea shore!' as txt, replaceOne(txt,
'sea', 'SEA') as replace_one, replaceAll(txt, 'sea', 'SEA') as replace_all;
```

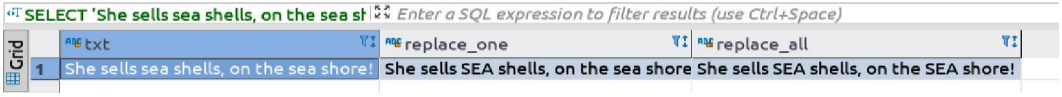


Figure 5.68: Replacing substring in the source string

replaceRegexpOne(), replaceRegexpAll()

```
SELECT 'She sells sea shells, on the sea shore!' as txt,
replaceRegexpOne(txt, 's..', 'SEA'), replaceRegexpAll(txt, 's.a',
'SEA');
```

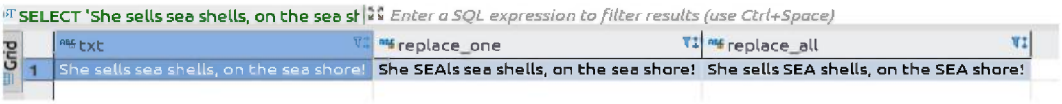


Figure 5.69: Replacing substring in source string using regex

Array functions

Arrays in ClickHouse have similar functions as strings have for data manipulation.

Array length functions

```
SELECT  array(1,2,3) as arr, empty(arr) as is_empty, notEmpty(arr) as
is_not_empty, length(arr) as length;
```

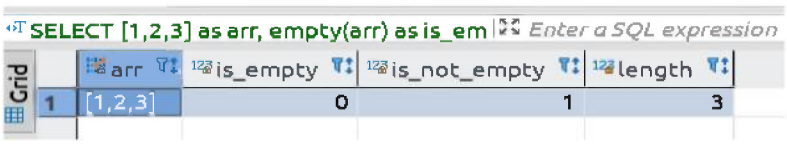


Figure 5.70: Finding the array length and empty arrays

Creating empty arrays

```
SELECT emptyArrayUInt8() as int_arr, emptyArrayFloat32() as float_arr,
emptyArrayString() as string_arr, emptyArrayDateTime() as date_time_arr;
```

Figure 5.71: Creating empty arrays

Array concatenation

```
SELECT arrayConcat ([1,2,3], [7,8,9], [4,5,6]) as concat_array;
```

***Figure 5.72**: Array concatenation*

Accessing the array elements

```
SELECT array('a','b','c') as arr, arrayElement(arr, 2) array_element,
has(arr, 'e') as has_element, hasAll(arr, ['a','b']) as hasAll,
hasAny(arr,['c','d','e']) as has_any;
```

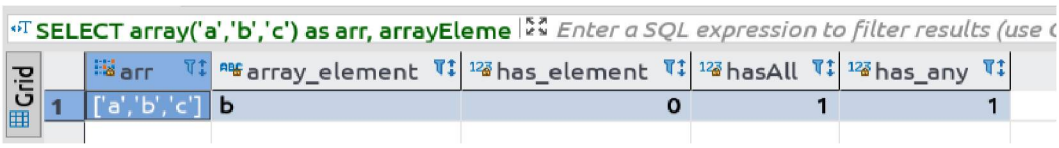


***Figure 5.73**: Accessing and checking array elements*

Finding and counting elements

```
SELECT array(3,1,2,4,5,1,2,3) as arr, indexOf(arr,2) as index,
countEqual(arr,2) as count;
```

***Figure 5.74**: Finding and counting array elements*

Push and pop operations

```
SELECT  array(1,2,4) as arr, arrayPushFront(arr,7) as  push_front,
arrayPushBack(arr,7) as push_back, arrayPopFront(arr) as pop_front,
arrayPopBack(arr) as pop_back;
```

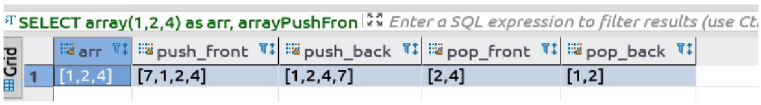


***Figure 5.75**: Push and pop operations in an array*

Slicing and resizing

The **arraySlice()** function has three arguments. The first one is the array to be sliced, the second one is the offset or the starting index of the elements in the source array from which the slicing operation is performed, and the third optional argument is the length of the slice. Similarly, for the **arrayResize()** function, the first argument is the source array, the second one is the new size, and the third optional argument is the default value to be filled if the new size is bigger than the old one.

```
SELECT  array(1,2,3,4,5) as arr, arraySlice(arr, 3, 2) as slice,
arrayResize(arr, 7, 7) as resize;
```

Figure 5.76: Slicing and resizing the arrays

Sorting the array

```
SELECT  array(3,1,8,4,5,2) as arr, arraySort(arr) as sort,
arrayReverseSort(arr) as rev_sort;
```

Figure 5.77: Sorting the array

Unique elements in an array

To find the unique elements in an array, **arrayDistinct()** can be used. To find the total number of unique elements, the **arrayUniq()** function can be used.

```
SELECT  array(3,1,2,4,5,1,2,3) as arr, arrayUniq(arr) as unique_count,
arrayDistinct(arr) as unique_elements;
```

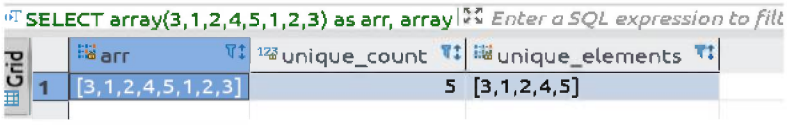


Figure 5.78: Unique elements count and unique count in array

Splitting and merging arrays and strings

The string can be split into an array using a character or string separator.

```
SELECT 'Hello Hi Hello Hi' as txt, splitByChar(' ', txt) as split_char,
splitByString('Hello',txt) as split_string;
```

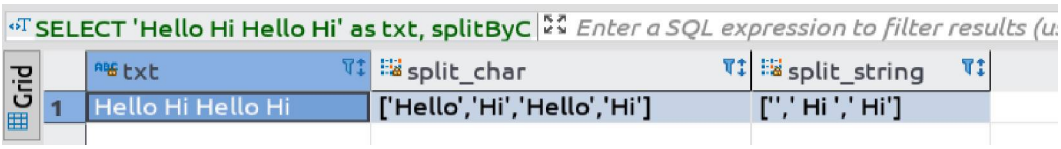


Figure 5.79: Splitting strings

To merge an array of strings, **`arrayStringConcat()`** can be used. A separator can be passed as an optional argument.

```
SELECT array('a','p','p','l','e') as arr, arrayStringConcat(arr,':')
arr_str;
```

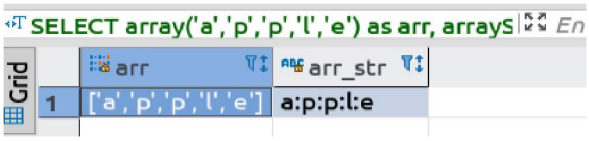


Figure 5.80: Array to string

Conclusion

ClickHouse has a rich set of in-built functions that can be used for data type conversion, working with multiple data types like numbers, strings, Date/DateTime, and arrays. There are many useful functions to search and replace in strings and manipulate arrays.

In the next chapter, we shall see the aggregate functions, which are different from the ones that were covered in this chapter. Aggregate functions are quite useful in performing analytical tasks over the datasets.

Points to remember

- ClickHouse has regular and aggregate functions.
- There are in-built functions to convert data types in ClickHouse.
- There are functions available to work with integers, floats, and Date/DateTime types.

- ClickHouse has a lot of utility functions to work with Date/DateTime arithmetic operations.
- String search and manipulations are possible with the functions available.
- Array manipulation functions are also available as part of ClickHouse.

Multiple- choice questions

1. **The function toInt64OrZero() accepts the following datatype as input argument.**
   a) Date
   b) Decimal
   c) String
   d) All

2. **The output of toString(toDate('2010-11-01') function is _____.**
   a) Error
   b) '2010-11-01'

3. **Trigonometric functions take input in ____.**
   a) Radians
   b) Degrees
   c) Both

4. **The first argument of dateDiff() function is _____.**
   a) Time unit
   b) Timestamp
   c) Fall-back value

5. **The two arguments of subtractYears() function are _____.**
   a) DateTime, DateTime
   b) Years to be subtracted and DateTime
   c) DateTime and Years to be subtracted

6. **The output of following query "SELECT empty("") is _____.**
   a) 0
   b) 1

c) True

d) False

7. **The arguments of string repetition function repeat() are _____.**

   a) String to be repeated and number of times to repeat

   b) Number of times to repeat and string to be repeated

   c) Number of times to repeat and string to be repeated and optional string to be added at the end

8. **The arguments of substring() function are _____.**

   a) Starting position, ending position, and source string

   b) Starting position, length, and source string

   c) Source string, starting position, and length

9. **arrayElement(arr, 2) returns _____.**

   a) Third element

   b) Second element

   c) Index of occurrence of the number "2"

10. **The arguments of the arraySlice() function are _____.**

    a) Array to be sliced, length of the slice, and offset

    b) Offset, length of the slice, and array to be sliced

    c) Array to be sliced, offset, and length of the slice

Answers

1. **c**
2. **b**
3. **b**
4. **a**
5. **c**
6. **b**
7. **a**
8. **c**
9. **b**
10. **c**

CHAPTER 6

SQL Functions for Data Aggregation

An aggregate function uses a set of values as input and produces a single output value. For example, the sum() function takes a set of rows as the input and returns a single value for a column. The output is calculated ignoring the Null values. The aggregate functions in ClickHouse are similar to the regular functions in usage, except for the number of outputs that are returned from the function. Aggregate functions are frequently used along with the `GROUP BY` clause.

In this chapter, we shall look at the commonly used aggregate functions, which are available in ClickHouse. We shall make use of the electricity consumption database, which was already set up in the earlier chapters, to illustrate the usage of commonly used aggregate functions in ClickHouse.

Structure

In this chapter, we will discuss the following topics:

- Aggregate functions in ClickHouse
- Combinators

Objectives

After reading this chapter, the reader will get to know about the various aggregate functions and it's variants in ClickHouse.

Aggregate functions

The aggregate functions ignore the Null values on the columns and perform the calculation on the non-Null values. The following are the commonly used aggregate functions available in ClickHouse. For an extensive list of available functions, it is recommended to refer the documentation. We shall make use of the electricity data, which we populated in *Chapter 3 – Setting up the Environment*, for most of the illustrations.

COUNT

The **`COUNT()`** function in ClickHouse returns the number of rows when no parameters are passed and the number of non-Null values if an expression is passed. The data type of the returned value is **`UInt64`**.

```
SELECT  COUNT() FROM electricity.consumption ;
```

***Figure 6.1**: Counting the number of rows in a table using count() function*

```
SELECT  COUNT(DISTINCT `Date`) FROM electricity.consumption ;
```

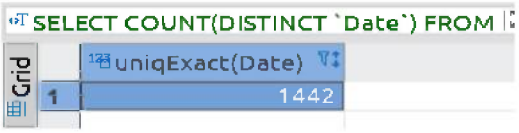


***Figure 6.2**: Counting the number of unique values in a column using count() function*

Any

The **`any()`** function returns the first encountered value in the mentioned column. The other variants of this function are **`anyHeavy()`** (returns a frequently occurring value chosen via Heavy Hitters algorithm) and **`anyLast()`** (returns the last encountered value).

```
SELECT any(`Date`) FROM electricity.consumption;
```

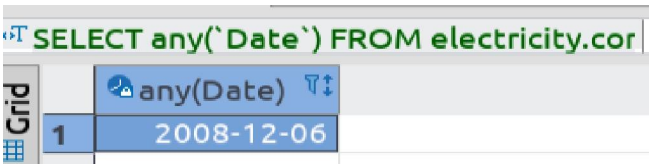


***Figure 6.3**: Choosing a value using any() function*

```
SELECT anyHeavy(`Date`) FROM electricity.consumption;
```

***Figure 6.4**: Choosing a value using anyHeavy() function*

```
SELECT anyLast(`Date`) FROM electricity.consumption;
```

***Figure 6.5**: Choosing a value using anyLast() function*

Min/max

As the name suggests, to find the minimum and maximum values from a column, **`min()`** and **`max()`** functions can be used.

```
SELECT min(Voltage), max(Voltage) FROM electricity.consumption;
```

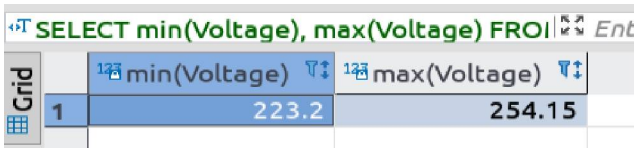


***Figure 6.6**: Finding minimum and maximum values using min() and max() functions*

Argmin/argmax

These functions are used to find the records for which the minimum or maximum values are encountered for the specified column. These functions accept two arguments: the first one is the column whose value will be returned and the second

argument is the column name whose minimum or the maximum value will be used to select the records.

```
SELECT argMin(`Date` ,Voltage), argMin(Voltage, Voltage) FROM
electricity.consumption;
```

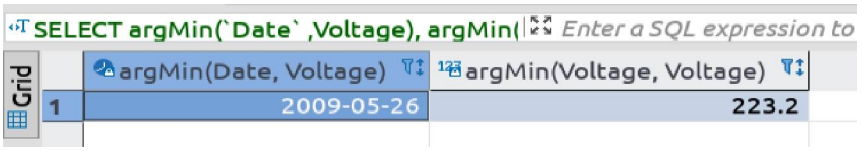


Figure 6.7: argMin() function and it's usage

```
SELECT argMax(`Date` ,Voltage), argMax(Voltage, Voltage) FROM
electricity.consumption;
```

Figure 6.8: argMax() function and it's usage

Sum

To find the sum of any numeric columns, **`sum()`** function can be used.

```
SELECT sum(Voltage) FROM electricity.consumption;
```

Figure 6.9: sum() function and it's usage

Average

To find the average of a numeric column, **`avg()`** function can be used in ClickHouse.

```
SELECT avg(Voltage) FROM electricity.consumption;
```

Figure 6.10: Finding the average of a numeric column

Quantile

Quantiles are points that divide a distribution into intervals of equal probability. Median is placed in a probability distribution in such a way that exactly half of the data is lower than the median and half of the data is above the median.

The quantile function accepts two arguments: the first one is the quantile level (which is purely optional and can accept any value between 0.0 and 1.0. At 0.5, median is calculated) and the second one is the expression over the column on which the quantile is calculated.

```
SELECT quantile(0.5)(Voltage), quantile(0.7)(Global_active_power)  FROM
electricity.consumption;
```

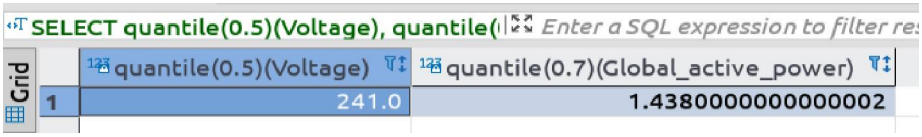


Figure 6.11: Finding the quantiles of a numeric column

Apart from the `quantile()` function, the following functions are available to compute the quantiles:

- **quantileExact()**
- **quantileExactWeighted()**
- **quantileTiming()**
- **quantileTimingWeighted()**
- **quantileTDigest()**
- **quantileTDigestWeighted()**
- **quantileDeterministic()**

The difference between these functions is left for the readers as an exercise. Details about these are available in the official documentation.

The `median()` function acts as alias to quantile functions and the following list of functions have the median family functions and their corresponding aliases:

- `medianDeterministic()` – `quantileDeterministic()`
- `medianExact()` – `quantileExact()`
- `medianExactWeighted()` – `quantileExactWeighted()`
- `medianTiming()` – `quantileTiming()`
- `medianTimingWeighted()` – `quantileTimingWeighted()`

- `medianTDigest()` – `quantileTDigest()`
- `medianTDigestWeighted()` – `quantileTDigestWeighted()`

Variance and standard deviation

The variance is mathematically defined as the average of the squared differences from the mean; and the standard deviation is the square root of variance.

```
SELECT varSamp(Voltage), varPop(Voltage) FROM electricity.consumption;
```

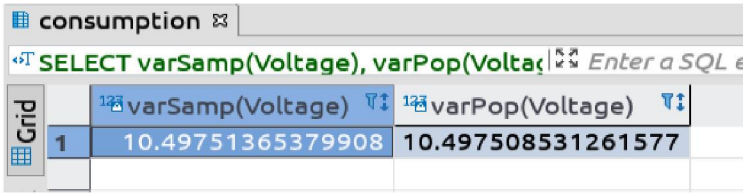


***Figure 6.12**: Finding the sample and population variance*

```
SELECT stddevSamp(Voltage), stddevPop(Voltage) FROM electricity.
consumption;
```

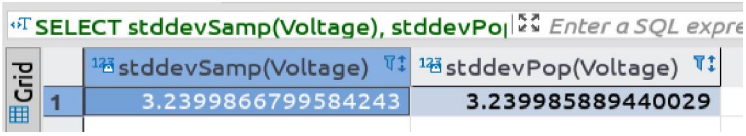


***Figure 6.13**: Finding the sample and population standard deviation*

Covariance

Covariance is a measure of joint variability of two variables. Higher covariance indicates a stronger relationship between the variables and vice versa.

```
SELECT covarSamp(Voltage, Global_active_power) AS sample_covar,
covarPop(Voltage, Global_active_power) AS population_covar FROM
electricity.consumption;
```

***Figure 6.14**: Finding the covariance of sample and population*

Correlation

Correlation is a statistical measure of the linear dependency between the two variables. Pearson's correlation coefficient can be calculated in ClickHouse using the `corr()` function.*

```
SELECT corr(Voltage, Global_active_power) FROM electricity.consumption;
```

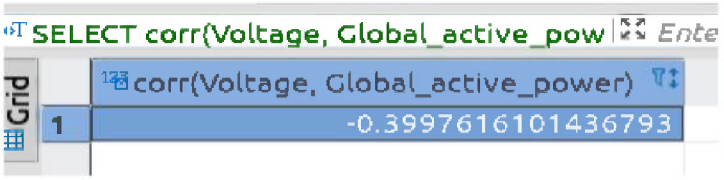


***Figure 6.15**: Finding the correlation coefficient between two numeric columns*

Skewness

The asymmetry of the distribution, which is otherwise known as skewness, can be found using the `skewPop()` and `skewSamp()` functions in ClickHouse. This works on numeric columns.

```
SELECT skewPop(Voltage) FROM electricity.consumption;
```

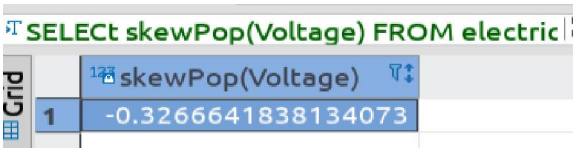


***Figure 6.16**: Finding the skewness of the population*

```
SELECT skewSamp(Voltage) FROM electricity.consumption;
```

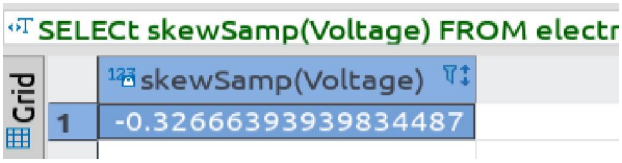


***Figure 6.17**: Finding the skewness of the sample population*

Kurtosis

Kurtosis is a measure that identifies if the tails of a given distribution contain extreme values and differ from the normal distribution. In ClickHouse, `kurtPop()` and `kurtSamp()` can be used to find the kurtosis.

```
SELECT kurtPop(Voltage) FROM electricity.consumption;
```

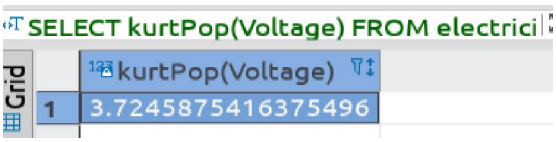


***Figure 6.18**: Finding the kurtosis of the population*

```
SELECT kurtSamp(Voltage) FROM electricity.consumption;
```

***Figure 6.19**: Finding the kurtosis of the sample population*

Combinators

Combinators are suffixed to the aggregate functions and they modify the working of the aggregate functions. The following combinators are allowed in ClickHouse:

If

The `If` combinator can be added to any aggregate function. The new aggregate function with `If` suffix accepts an additional argument, which is a conditional statement.

```
SELECT avgIf(Voltage, Voltage>245), medianIf(Voltage, Voltage>250) FROM
electricity.consumption;
```

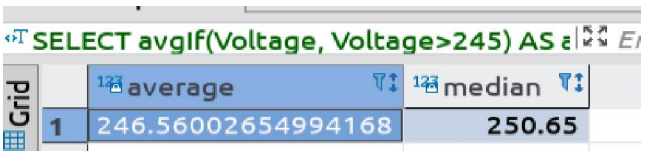


***Figure 6.20**: If combinator for avg() and median() function*

Array

To use the aggregate function on arrays, the **Array** combinator can be used with any aggregate function.

```
SELECT array(1,2,3,4,5) AS arr, avgArray(arr) as avg_arr, sumArray(arr)
as sum_arr;
```

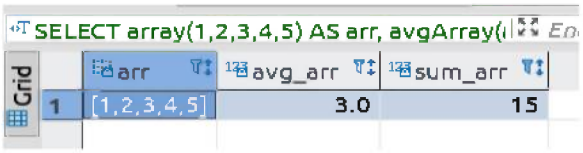


Figure 6.21: Array combinator functions

State

The **State** combinator when used along with an aggregate function returns an intermediate aggregation state instead of the final aggregation result. This can be used to finish the aggregation later. This combinator is commonly used along with:

- **AggregatingMergeTree** table engine
- **finalizeAggregation** function
- **runningAccumulate** function
- **Merge** combinator
- **MergeState** combinator

The following example shows the usage of the state combinator:

```
SELECT avgState(Voltage) as state, `Date` FROM electricity.consumption
GROUP BY `Date` LIMIT 5;
```

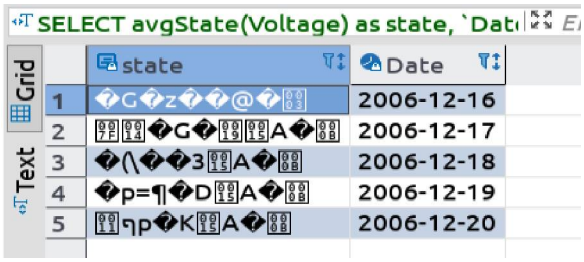


Figure 6.22: State combinator

The aggregation state may not be human readable, but can be used to process and get the final result using the **Merge** combinator.

Merge

An aggregate function with the **Merge** combinator takes the intermediate aggregation state as an argument, combines the states, and returns the resulting value.

The following example shows the usage of a **Merge** combinator. For this purpose, we create a materialized view, store the state combinator, and get the final result from there.

```
CREATE MATERIALIZED VIEW electricity.state_merge
ENGINE = MergeTree()
ORDER BY  (`Date`)
AS SELECT
avgState(Voltage) as state,
`Date` FROM electricity.consumption
GROUP BY `Date`;

INSERT INTO electricity.state_merge
SELECT
avgState(Voltage) as state,
`Date` FROM electricity.consumption
GROUP BY `Date`;

SELECT avgMerge(state) as final_result,`Date` FROM electricity.state_
merge GROUP BY `Date` LIMIT 5;
```

SELECT avgMerge(state) as final_result,`

	final_result	Date
1	236.24376262626276	2006-12-16
2	240.08702777777793	2006-12-17
3	241.23169444444474	2006-12-18
4	241.99931250000026	2006-12-19
5	242.30806250000026	2006-12-20

Figure 6.23: Merge combinator

MergeState

Similar to a **Merge** combinator, the **MergeState** combinator merges the intermediate aggregation state. However, instead of the final result, it produces another intermediate state like the state combinator.

```
SELECT avgMergeState(state) as final_result,`Date` FROM electricity.
state_merge GROUP BY `Date` LIMIT 5;
```

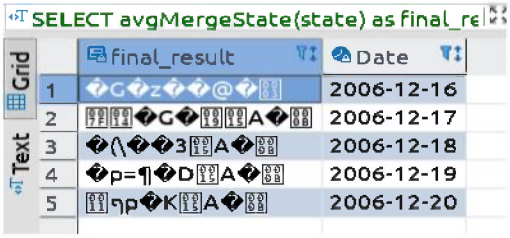


Figure 6.24: MergeState combinator

ForEach

The `ForEach` combinator used along with an aggregate function works on arrays and performs aggregation on individual array items.

```
SELECT avgForEach(arr)
FROM
(SELECT [1,2,3] as arr
UNION ALL
SELECT [4,5,6] as arr);
```

Figure 6.25: ForEach combinator

OrDefault

The `OrDefault` combinator is used along with an aggregate function whenever it is required to return the default value of the function's return type, if there is nothing to aggregate.

```
SELECT avg(number), avgOrDefault(number) FROM numbers(0);
```

Figure 6.26: OrDefault combinator

OrNull

The `OrNull` combinator is used along with an aggregate function whenever it is required to return the Null value, if there is nothing to aggregate.

```
SELECT avg(number), avgOrNull(number) FROM numbers(0);
```

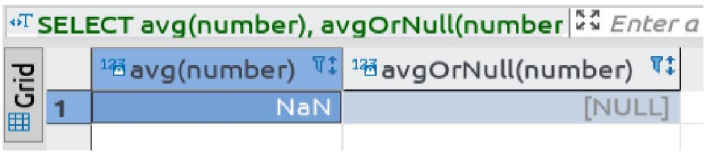


Figure 6.27: OrNull combinator

Conclusion

ClickHouse ships with a lot of useful aggregate functions to perform aggregation on the data. Apart from the regular aggregate functions, combinators are also available that expand the capability of the aggregate functions.

So far, we have seen the basics of ClickHouse, SQL capabilities, in-built functions and aggregate functions in ClickHouse, which could be useful in getting a decent hands on with ClickHouse. In the next section of this book, we shall cover the different table engines like mergetree and log. Apart from this, we shall also cover the integration of ClickHouse with different databases and data sources.

Points to remember

- Aggregate functions usually accept a set of values as input and produce a single output.
- ClickHouse has a lot of in-built aggregate functions.
- Combinators are used to modify the working of aggregate functions.

Multiple choice questions

1. **Aggregate functions usually accept one or many values as input and produces _____ output.**

   a) One

   b) Many

   c) Zero

2. **When an aggregate function encounters a Null value in the input.**
   a) Exception is raised
   b) Ignores the Null value and continues with the non-Null value

3. **Combinators are _____ to the aggregate functions.**
   a) Prefixed
   b) Suffixed

4. **To use the aggregate function on arrays, the `Array` combinator is used.**
   a) True
   b) False

5. **Input to the aggregate function with the MergeState combinator is the output of another aggregate function with the _____ combinator.**
   a) Merge
   b) State
   c) ForEach

Answers

1. **a**
2. **b**
3. **b**
4. **a**
5. **a**

CHAPTER 7

Table Engines – MergeTree Family

Storage engines (also known as table engines in ClickHouse) are the software modules in a database management system that determine the way the data is stored in disk, replication, supported queries, and other features inherent to the engine.

In this chapter, we shall look at the commonly used table engine family in ClickHouse, which is the MergeTree family. This engine is highly recommended for high-load tasks and they support data replication and partitioning. The other prominent engine family is the log family, which will be covered in the next chapter.

Structure

In this chapter, we will discuss the following topics:

- MergeTree engine
- ReplacingMergeTree engine
- SummingMergeTree engine
- AggregatingMergeTree engine
- CollapsingMergeTree engine
- VersionedCollapsingMergeTree engine

- ReplicatedMergeTree engine

Objectives

The main objectives of this chapter are to:

- Learn about the MergeTree engine and the common properties of the MergeTree family.
- Understand the different engines belonging to the MergeTree family and their properties.

MergeTree

MergeTree is the commonly used family of table engines in ClickHouse which comes loaded with lots of features. They are designed to support heavy-load tasks and offer a wide range of benefits like data resilience, custom partitioning, and super-fast data retrieval. The important features supported in MergeTree family are:

- Data replication via Zookeeper
- Data deduplication at the time of ingestion
- Partitioning the data
- Data is sorted by the primary key and stored
- Data sampling

The different engines in the MergeTree family add a unique extra functionality on top of the base MergeTree table engine. MergeTree table engine is practical for single-node ClickHouse setup and therefore it is used in the sample tables that were created earlier.

Understanding MergeTree engine

For illustrating the working of a MergeTree engine, let us use a sample table with columns named **`Name`**, **`Shop_ID`**, **`Price`**, and **`Quantity`** with **`Name`** and **`Price`** as the primary key columns. Let us say that we have inserted the following data into a MergeTree table. The SQL statement to create this table is as follows:

```
CREATE TABLE IF NOT EXISTS mergetree_sample_table (
    Name String,
    Price UInt3),
    Shop_ID UInt32 ,
```

```
    Quantity UInt32)
ENGINE = MergeTree()
PRIMARY KEY (Name, Price)
SETTINGS index_granularity=2;
```

Let us insert some sample data using the following SQL statement:

```
INSERT INTO mergetree_sample_table
('Apple',2,1,40)
('Apple',2,3,35)
('Apple',3,2,45)
('Apple',3,4,35)
('Orange',1,2,40)
('Orange',3,3,50)
('Banana',2,1,25)
('Banana',2,2,55);
```

Name	Price	Shop_ID	Quantity
Apple	2	1	40
Apple	2	3	35
Apple	3	2	45
Apple	3	4	35
Orange	1	2	40
Orange	3	3	50
Banana	2	1	25
Banana	2	2	55

***Figure 7.1**: Sample data inserted in MergeTree table engine*

While inserting the data, the data is written to disk after being sorted based on the primary key(s) (since we have not specified the columns to be sorted using the **`ORDER BY`** clause). The default data path where the data will be written on to the disk is **`/var/lib/clickhouse/data/<database_name>/<table_name>/<partition_name>/`**.

The MergeTree engine supports data partitioning (logical combination of records based on the user-specified columns). If a partition column is specified at the time of creating the table, a separate folder is created for each unique partition value and the corresponding data is stored inside those folders. For example, in the electricity consumption dataset, since we had specified partitioning the data by the **`Date`** column, there will be a separate column for each unique date value.

Within each partition folder, the data is stored in either wide (different files in the disk for different columns (<column_name>.bin), which is the default configuration) or compact (multiple columns in a single file) format, based on the configuration.

The data stored in the table is divided into granules and while reading the data, ClickHouse reads the whole data set within a granule. The granule size is specified using the **`index_granularity`** and **`index_granularity_bytes`** settings while creating the table (**`2`** in the table we just created).

ClickHouse uses sparse indexing which significantly reduces the index file size thereby enabling the system to load the entire index of a huge table in RAM. The index is maintained in the **`primary.idx`** file in each partition. The file contains the primary key value for the starting row corresponding to every granule. For example, if the **`index_granularity`** is **`4`** and the total number of rows are 10, the index file will contain the primary key values for the first, fifth, and ninth row, respectively.

For each column file (in wide format), there will be another file named **`<column_name>.mrk`**. This file is called a mark file and holds the information of data offsets corresponding to every granule. Coming back to the table that we had created, the data stored in files are as illustrated in the following figure. MergeTree supports multiple rows of data for the same primary key, unlike the traditional relational databases.

Figure 7.2: Sample data stored using MergeTree table engine

As we are aware that the primary keys are the **`Name`** and **`Price`** column in the table, and the index granularity is 2, the index file will contain the primary key values for the first, third, fifth, and seventh row. The mark file for the **`Name`** column will contain data offset for every granule (2 rows). If compression is used, the offset will contain two values (offset in compressed data and decompressed data).

While reading the data, if the query specifies **`Name='Apple'`**, the first and second granule will be read into memory and processed further. So, instead of scanning

the whole data from the disk, only a part of the data is read thereby speeding up the reading process. And if data is partitioned, the data read from the disk is even reduced further.

Creating a table

To create a table using the **MergeTree()** engine, the following syntax can be used:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER
cluster_name]
(
    column1 [datatype1] [DEFAULT|MATERIALIZED|ALIAS expr1] [TTL expr1],
    column2 [datatype2] [DEFAULT|MATERIALIZED|ALIAS expr2] [TTL expr2],
    ...
       ...
) ENGINE = MergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[PRIMARY KEY expr]
[SAMPLE BY expr]
[TTL expr]
[SETTINGS name=value, ...]
```

The syntax is already covered in *Chapter 4* (in **CREATE** statement section). The partitioning column, primary key expression, sampling settings, and time to live expressions can be specified along with the **CREATE** statement as query clause. All the query clauses are optional, but there should be at least either **PRIMARY KEY** or **ORDER BY** clause in the **CREATE** statement, if not both. Based on the preceding syntax, let us create a sample table and look at the details:

```
CREATE DATABASE mergetree_testing;


CREATE TABLE IF NOT EXISTS mergetree_testing.mergetree (
       ID String,
       Name String DEFAULT ID CODEC(LZ4),
       Quantity UInt32 CODEC(ZSTD),
       "Date" Date,


ENGINE = MergeTree()
```

```
PARTITION BY toYYYYMMDD(Date)
PRIMARY KEY (ID)
ORDER BY (ID, Name, Date)
SETTINGS
index_granularity=1024;
```

The preceding query creates a table named **mergetree** under the **mergetree_testing** database. The columns in the table are **ID**, **Name**, **Quantity**, and **Date**.

The table is partitioned by the date column. The primary key is the column ID. The table is ordered by **ID**, **Name**, and **Date**. The following settings parameters can be specified for the MergeTree table engine:

- **index_granularity**: Default value is **8192**. This value determines the number of rows between marks of an index.
- **index_granularity_bytes**: Default value is **10 Mb**. This value determines the maximum size of each granule.
- **enable_mixed_granularity_parts**: If enabled, the granule size is controlled by **index_granularity_bytes**, or else it is controlled by **index_granularity** settings. Enabling this for tables with huge row sizes can improve the efficiency of **SELECT** queries on the table.
- **use_minimalistic_part_header_in_zookeeper**: If enabled, this setting stores headers of data part in a single **znode** in Zookeeper. This setting is for replicated tables.
- **min_merge_bytes_to_use_direct_io**: Based on the storage size of data that is merged, ClickHouse uses direct I/O.
- **merge_with_ttl_timeout**: Default value: **86400** (units – seconds, which is equal to one day). Minimum time delay in seconds before repeating a merge with TTL.
- **write_final_mark**: Default value is **1**. Toggles writing the final index mark at the end of the data part.
- **merge_max_block_size**: Default value is **8192**. A maximum number of rows in a block for merge operations.

Let us insert some sample data and see how it is stored in the table. The following set of queries can be executed to insert and view the inserted data:

```
INSERT INTO mergetree_testing.mergetree VALUES
('aa','a', 1, '2020-01-01'),
```

```
('aa','a', 2, '2020-01-02'),
('ba','b', 1, '2020-01-03');


SELECT * FROM mergetree_testing.mergetree;
```

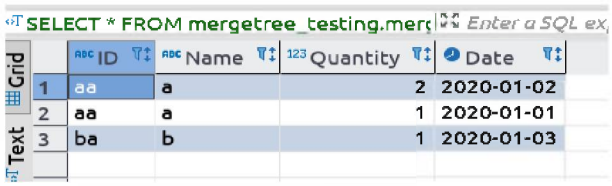


***Figure 7.3**: Test data stored in MergeTree table engine*

ReplacingMergeTree()

This engine from the MergeTree family is mainly used where duplicates are not desired. The removal of duplicates is based on the columns specified in the **`ORDER BY`** clause while creating the table.

The removal of duplicates occurs during the merge operation. The end-user will not have any control over data deduplication since the merge operation runs automatically in the background by ClickHouse. The **`OPTIMIZE`** statement can be executed to perform merge manually but since it reads and writes a large amount of data, it is not recommended to be run frequently.

The **`ReplacingMergeTree`** engine has the same set of clauses and clause requirements as the MergeTree table engine. But the **`ReplacingMergeTree`** accepts an optional parameter (in **`ENGINE = ReplacingMergeTree([column_name])`** clause in **`CREATE`** statement), which determines the logic for replacing the duplicate rows. The column name can be specified as a parameter (should be of **`UInt*`**, **`Date`**, or **`DateTime`** data type) and if it is specified, the row with the maximum value for that column will be retained. If it is not specified, the last row returned on the **`SELECT`** query for the primary / sorting key is retained.

Creating a table

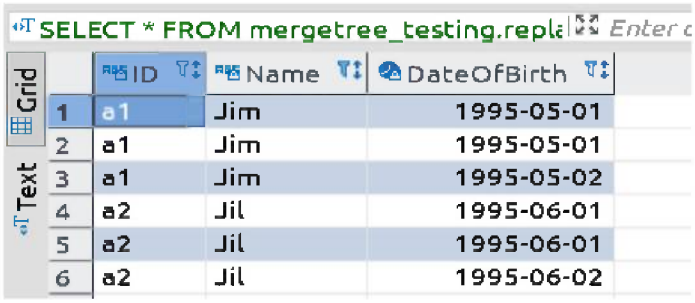
The create statement is similar to the one for the MergeTree table engine. The following query can be used to create a **`ReplacingMergeTree`** table:

```
CREATE TABLE IF NOT EXISTS mergetree_testing.replacingmergetree (
        ID String,
        Name String,
        DateOfBirth Date)
```

```
ENGINE = ReplacingMergeTree()
PARTITION BY ID
ORDER BY (ID, DateOfBirth)
SETTINGS
index_granularity=1024;
```

The table has three columns (**ID**, **Name**, and **DateOfBirth**), and is partitioned by the ID column. The data is ordered by **ID** and **DateOfBirth**. Let us now insert some sample data to test the table. The following query can be executed to insert some test data into the table:

```
INSERT INTO mergetree_testing.replacingmergetree VALUES
('a1', 'Jim', '1995-05-01'),
('a1', 'Jim', '1995-05-01'),
('a1', 'Jim', '1995-05-02'),
('a2', 'Jil', '1995-06-01'),
('a2', 'Jil', '1995-06-01'),
('a2', 'Jil', '1995-06-02');


SELECT * FROM mergetree_testing.replacingmergetree;
```



	ID	Name	DateOfBirth
1	a1	Jim	1995-05-01
2	a1	Jim	1995-05-01
3	a1	Jim	1995-05-02
4	a2	Jil	1995-06-01
5	a2	Jil	1995-06-01
6	a2	Jil	1995-06-02

***Figure 7.4**: Test data stored in ReplacingMergeTree table engine*

Since the merge may not have happened immediately after inserting the data, we can run **OPTIMIZE** query to manually perform the merge operation.

```
OPTIMIZE TABLE mergetree_testing.replacingmergetree FINAL;
```

Once the **OPTIMIZE** query is executed successfully, we should be able to see the final table now:

```
SELECT * FROM mergetree_testing.replacingmergetree;
```

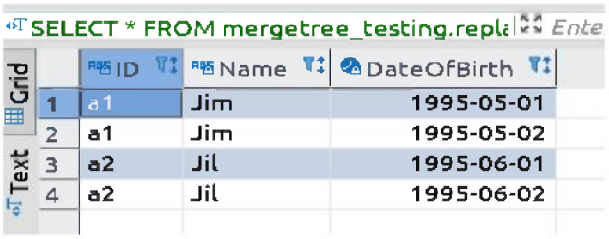


***Figure 7.5**: Deduplicated data in ReplacingMergeTree table engine*

Now, let us try this example by providing the optional column parameter. The column specified as a parameter should be of **`UInt`**, **`Date`**, or **`DateTime`** type. In our case, we shall use the **`DateOfBirth`** column.

```
CREATE TABLE IF NOT EXISTS mergetree_testing.replacingmergetree_1 (
       ID String,
       Name String,
       DateOfBirth Date)
ENGINE = ReplacingMergeTree(DateOfBirth)
ORDER BY (ID, DateOfBirth)
SETTINGS
index_granularity=1024;

INSERT INTO mergetree_testing.replacingmergetree_1 VALUES
('a1', 'Jim', '1995-05-01'),
('a1', 'Jimmy', '1995-05-02'),
('a1', 'Jim', '1995-05-02'),
('a2', 'Jil', '1995-06-01'),
('a2', 'Jil', '1995-06-01'),
('a2', 'Jil', '1995-06-02');

SELECT * FROM mergetree_testing.replacingmergetree_1;
```

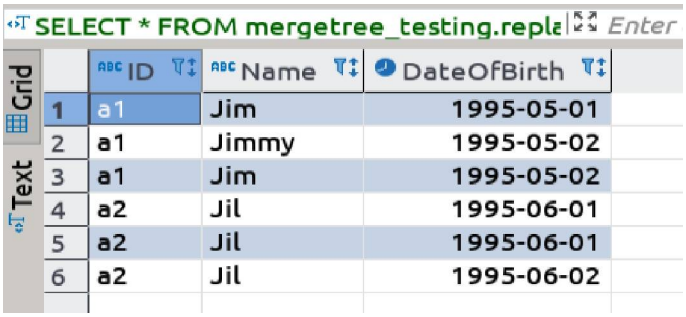


***Figure 7.6**: Test data stored in ReplacingMergeTree table engine*

```
OPTIMIZE TABLE mergetree_testing.replacingmergetree_1 FINAL;

SELECT * FROM mergetree_testing.replacingmergetree_1;
```

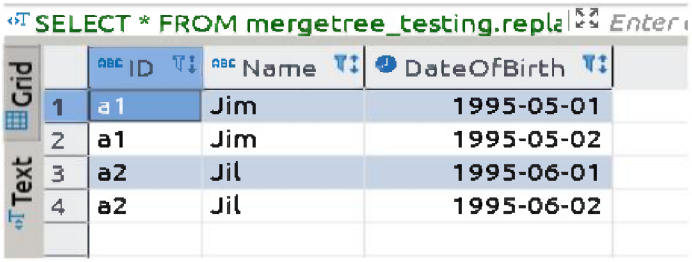


Figure 7.7: Deduplicated data in ReplacingMergeTree table engine

SummingMergeTree

SummingMergeTree table engine is quite similar to **ReplacingMergeTree** engine. But instead of replacing the duplicate row, the numeric columns belonging to the rows with the same primary/sorting key will be summed.

Similar to **ReplacingMergeTree**, the data merge is taken care by ClickHouse and the summing of numeric columns happens at the time of merge. Therefore, there are chances for the table to hold rows with the same primary/sorting key. So, it is recommended to use the **sum()** function on numeric columns and group by the primary/sorting key while reading the data.

The **SummingMergeTree** accepts an optional parameter, which is a tuple of columns and if specified, the summing is based on the rows with the same value in the specified columns instead of the primary/sorting key. The specified columns must not be in primary key columns.

Creating a table

The following query can be used to create a **SummingMergeTree** table:

```
CREATE TABLE IF NOT EXISTS mergetree_testing.summingmergetree (
       Name String,
       Quantity UInt32,
       Weight UInt32)
ENGINE = SummingMergeTree()
ORDER BY (Name)
SETTINGS
index_granularity=1024;
```

This will create a table with three columns (**Name**, **Quantity**, and **Weight**). The table is sorted by the **Name** column. Let us now insert some sample data to test the table. The following query can be executed to insert some test data into the table and view the inserted data:

```
INSERT INTO mergetree_testing.summingmergetree VALUES
('Apple', 10, 25 ),
('Apple', 8, 15 ),
('Apple', 7, 14 ),
('Orange', 10, 25 ),
('Orange', 12, 35 ),
('Orange', 14, 42 );


SELECT * FROM mergetree_testing.summingmergetree;
```

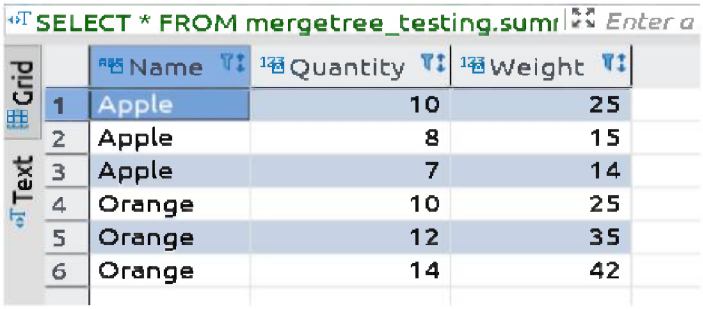


***Figure 7.8**: Test data in SummingMergeTree table engine*

To manually merge the data parts, the **OPTIMIZE** statement can be executed. Let us take a look at the final data after merge by executing the following set of queries:

```
OPTIMIZE TABLE mergetree_testing.summingmergetree FINAL;
SELECT * FROM mergetree_testing.summingmergetree;
```

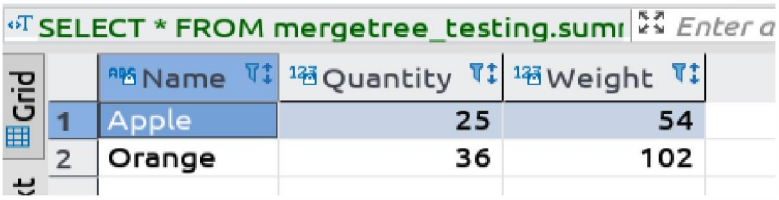
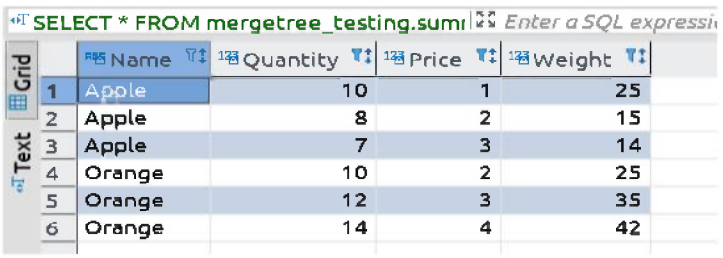


***Figure 7.9**: Data in SummingMergeTree table engine after merging data parts*

Let us now create a similar table with summing columns specified as parameter:

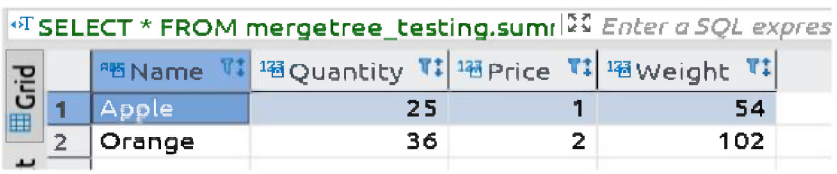
```
CREATE TABLE IF NOT EXISTS mergetree_testing.summingmergetree_1 (
       Name String,
       Quantity UInt32,
       Price UInt32,
       Weight UInt32)
ENGINE = SummingMergeTree((Quantity, Weight))
ORDER BY (Name)
SETTINGS
index_granularity=1024;

INSERT INTO mergetree_testing.summingmergetree_1 VALUES
('Apple', 10, 1, 25 ),
('Apple', 8, 2, 15 ),
('Apple', 7, 3, 14 ),
('Orange', 10, 2, 25 ),
('Orange', 12, 3, 35 ),
('Orange', 14, 4, 42 );

SELECT * FROM mergetree_testing.summingmergetree_1;
```



	Name	Quantity	Price	Weight
1	Apple	10	1	25
2	Apple	8	2	15
3	Apple	7	3	14
4	Orange	10	2	25
5	Orange	12	3	35
6	Orange	14	4	42

***Figure 7.10**: Test data in SummingMergeTree table engine*

```
OPTIMIZE TABLE mergetree_testing.summingmergetree_1 FINAL;
SELECT * FROM mergetree_testing.summingmergetree_1;
```



	Name	Quantity	Price	Weight
1	Apple	25	1	54
2	Orange	36	2	102

***Figure 7.11**: Data in SummingMergeTree table engine after merging data parts*

When summing columns are specified, if other numeric columns are still present in the table, an arbitrary value is selected for that numeric column.

AggregatingMergeTree

ClickHouse provides this table engine to store aggregating states of columns, which can be used to get aggregated results while querying. **AggregatingMergeTree** engine replaces the rows with the same primary/sorting key with the combination of states of aggregate functions, instead of the final aggregated value itself. The clauses used to create the table with the **AggregatingMergeTree** are same as the ones used in MergeTree.

The usage of the **AggregatingMergeTree** engine can be inferred from the following example. Let us first create a table using the **AggregatingMergeTree** engine, insert some data, and get the final aggregated results via query. To create the table, the following query can be executed:

```
CREATE TABLE IF NOT EXISTS mergetree_testing.aggregatingmergetree(
      Name String,
      Quantity_Average AggregateFunction(avg, Float64),
      Quantity_Minimum AggregateFunction(min, Float64),
      Quantity_Maximum AggregateFunction(max, Float64))
ENGINE = AggregatingMergeTree()
ORDER BY (Name);
```

The preceding query creates a table with a **Name** column and three columns meant for aggregation (avg, min, and max). The table is ordered by name. For populating the data, we shall use the table **mergetree_testing.mergetree**, which was created earlier. The following **INSERT** statement can be executed to populate the data in the newly created table:

```
INSERT INTO mergetree_testing.aggregatingmergetree
SELECT Name,
avgState(toFloat64(Quantity)) as Quantity_Average,
minState(toFloat64(Quantity)) as Quantity_Minimum,
maxState(toFloat64(Quantity)) as Quantity_Maximum
FROM mergetree_testing.mergetree GROUP BY Name;
```

We are inserting the name column and aggregation state columns from the **mergetree_testing.mergetree**, which is grouped by the **Name** column. So, the average, minimum, and maximum aggregation states of the **Quantity** column, for each distinct value in the **Name** column will be available in the newly created table.

```
SELECT * FROM mergetree_testing.aggregatingmergetree ;
```

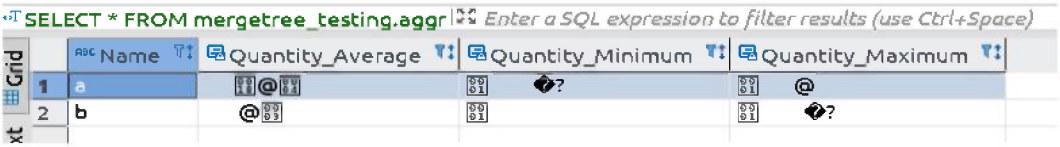


Figure 7.12: Test data in AggregatingMergeTree table engine

Non-readable characters are displayed in the columns storing aggregation states. They are just the intermediate states of aggregation. To get the final aggregated result, we use aggregate functions suffixed with **Merge**. The following example shows how to extract the final aggregated result:

```
SELECT Name, avgMerge(Quantity_Average) as average, minMerge(Quantity_
Minimum) as minimum,
maxMerge(Quantity_Maximum) as maximum
FROM mergetree_testing.aggregatingmergetree
GROUP BY Name;
```

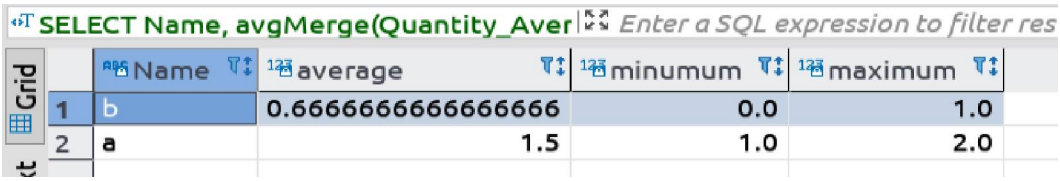


Figure 7.13: Aggregated results from AggregatingMergeTree table engine

It is also possible to create a materialized view with the **AggregatingMergeTree** engine. This view will keep watching the source table and will perform aggregation whenever a new set of data is ingested in the source table. The following statement can be used to create a materialized view:

```
CREATE MATERIALIZED VIEW mergetree_testing.aggregatingmergetree_mat_view
ENGINE = AggregatingMergeTree()
ORDER BY (Name)
AS SELECT
Name,
avgState(toFloat64(Quantity)) as Quantity_Average,
minState(toFloat64(Quantity)) as Quantity_Minimum,
maxState(toFloat64(Quantity)) as Quantity_Maximum
FROM mergetree_testing.mergetree GROUP BY Name;
```

```
SELECT * FROM mergetree_testing.aggregatingmergetree_mat_view ;
```

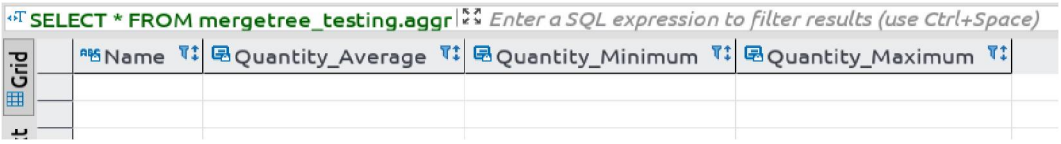


Figure 7.14: Data in newly created materialized view using AggregatingMergeTree table engine

Let us now insert a new set of data in the source table, which is **mergetree_testing.mergetree**, and verify the data in the materialized view after insertion.

```
INSERT INTO mergetree_testing.mergetree VALUES
('cc','c', 1, '2020-01-01'),
('cc','c', 2, '2020-01-02'),
('cc','c', 1, '2020-01-03');
SELECT * FROM mergetree_testing.aggregatingmergetree_mat_view ;
```

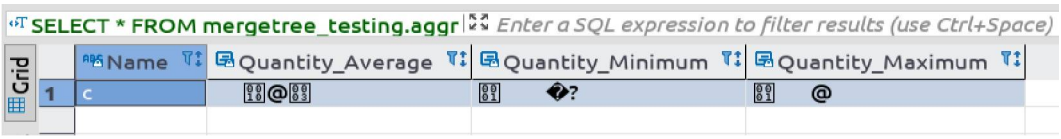


Figure 7.15: Materialized view based on AggregatingMergeTree table engine after inserting a new set of data in the source table

Note that the materialized view will only hold the new set of data that was inserted after the materialized view was created. The old data that is already available in the source table will not be available. To retrieve the aggregated data, the following select query can be used:

```
SELECT Name, avgMerge(Quantity_Average) as average, minMerge(Quantity_
Minimum) as minumum, maxMerge(Quantity_Maximum) as maximum FROM
mergetree_testing.aggregatingmergetree_mat_view GROUP BY Name;
```

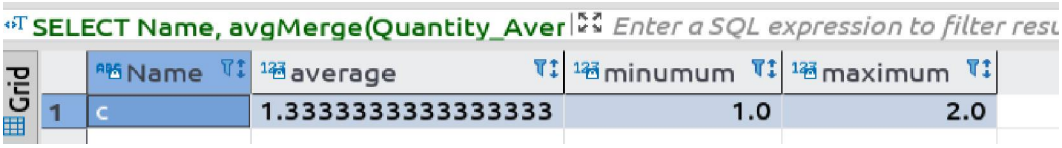


Figure 7.16: Aggregated results from the materialized view based on AggregatingMergeTree table engine

CollapsingMergeTree

This engine collapses (deletes) the rows with the same sorting key based on a sign column. This engine can be useful when we have data that changes continuously and

have to update the rows frequently. Since updates are not efficient in ClickHouse, this engine can come in handy whenever there is a need for frequent updates.

The **sign** column can have either **1** or **-1** as a value. The rows with **1** as the value in the sign column are called state rows and the rows with **-1** are called cancel rows. The state rows hold the most recent data and the cancel rows hold the older records that need to be eliminated. Let us proceed with an example to illustrate the working of the **CollapsingMergeTree** engine. We shall start by creating a table with **CollapsingMergeTree** and insert data to see how the collapsing operations work. The clauses for creating the table are the same as for the MergeTree engine. The following query can be used to create a table:

```
CREATE TABLE IF NOT EXISTS mergetree_testing.collapsingmergetree(
       ID String,
       Count UInt64,
       Sign Int8)
ENGINE = CollapsingMergeTree(Sign)
ORDER BY (ID);
```

The table has three columns (**ID**, **Count**, and **Sign**). The Sign column should be of **UInt8** data type. The table is ordered by **ID** column. Let us begin by inserting the sample data.

```
INSERT INTO mergetree_testing.collapsingmergetree VALUES
('a', 32, 1);
SELECT * FROM mergetree_testing.collapsingmergetree ;
```

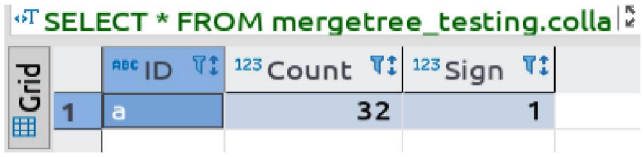


***Figure 7.17**: Test data in CollapsingMergeTree table engine*

The table now has a single row for ID **a**. Let us assume that we need to update the value for the ID **a**. We can perform an insert as given in the following:

```
INSERT INTO mergetree_testing.collapsingmergetree VALUES
('a', 32, -1), ('a', 64, 1);


SELECT * FROM mergetree_testing.collapsingmergetree ;
```

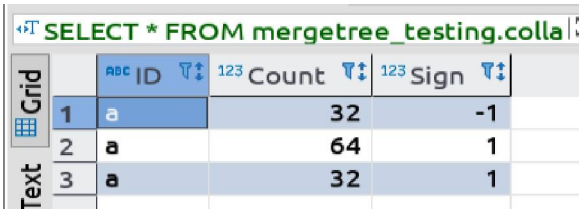


***Figure 7.18**: Updated test data in CollapsingMergeTree table engine*

The new set of rows are inserted. Note that the row with old values is inserted again with sign `-1`, which denotes that the values for the rows with the same sorting key are now in canceled state. So, when a merge occurs, those rows will be removed. Let us force the merge operation in case it has not happened in the background already.

```
OPTIMIZE TABLE mergetree_testing.collapsingmergetree FINAL;
```

```
SELECT * FROM mergetree_testing.collapsingmergetree ;
```

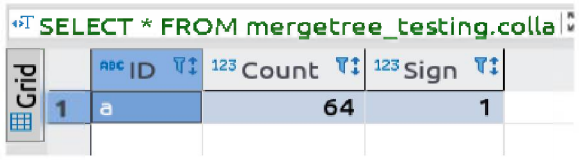


***Figure 7.19**: Updated test data in CollapsingMergeTree table engine, after the merge operation*

We can see that the old rows with the `-1` sign and the same primary/sorting key were removed and only the new rows are available in the table. The following are the data retention rules in the **`CollapsingMergeTree`** table engine.

For rows with the same primary/sorting key:

- If the number of states and cancel rows are equal, and the last row is a state row, then the first cancel and the last state rows are retained.
- If the number of cancel rows are more than state rows, then the first cancel row is retained in the table.
- If the number of state rows are more than cancel rows, then the last state row is retained in the table.
- No rows are retained for all other cases.

Let us test the data collapsing rules in the **`CollapsingMergeTree`** table using the following set of statements:

```
INSERT INTO mergetree_testing.collapsingmergetree VALUES
('b', 32, -1), ('b', 64, -1), ('b', 32, 1), ('b', 64, 1);
```

```
SELECT * FROM mergetree_testing.collapsingmergetree ;
```

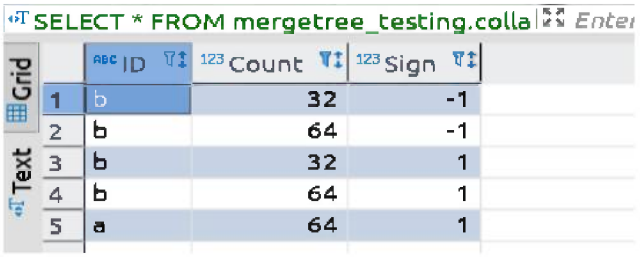


***Figure 7.20**: Test data in CollapsingMergeTree table engine, before the merge operation*

```
OPTIMIZE TABLE mergetree_testing.collapsingmergetree FINAL;
```

```
SELECT * FROM mergetree_testing.collapsingmergetree ;
```

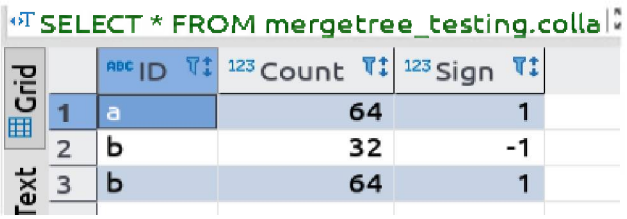


***Figure 7.21**: Test data in CollapsingMergeTree table engine, after the merge operation*

In this example, since the state and cancel rows are equal, the last state row and the first cancel row are retained.

```
INSERT INTO mergetree_testing.collapsingmergetree VALUES
('c', 32, -1), ('c', 64, -1), ('c', 64, 1);
```

```
SELECT * FROM mergetree_testing.collapsingmergetree ;
```

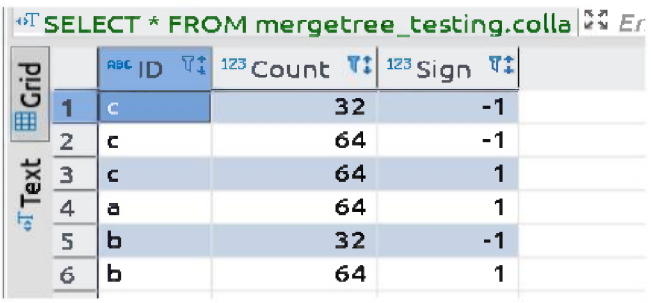


***Figure 7.22**: Test data in CollapsingMergeTree table engine, before the merge operation*

```
OPTIMIZE TABLE mergetree_testing.collapsingmergetree FINAL;

SELECT * FROM mergetree_testing.collapsingmergetree ;
```

SELECT * FROM mergetree_testing.colla

Grid | Text

	ID	Count	Sign
1	a	64	1
2	b	32	-1
3	b	64	1
4	c	32	-1

***Figure 7.23**: Test data in CollapsingMergeTree table engine, after the merge operation*

In this example, since cancel rows are more than the state rows, the first cancel row is retained.

```
INSERT INTO mergetree_testing.collapsingmergetree VALUES
('d', 32, -1), ('d', 64, 1), ('d',128, 1);

SELECT * FROM mergetree_testing.collapsingmergetree ;
```

SELECT * FROM mergetree_testing.colla Enter a SQ

Grid | Text

	ID	Count	Sign
1	d	32	-1
2	d	64	1
3	d	128	1
4	a	64	1
5	b	32	-1
6	b	64	1
7	c	32	-1

***Figure 7.24**: Test data in CollapsingMergeTree table engine, before the merge operation*

```
OPTIMIZE TABLE mergetree_testing.collapsingmergetree FINAL;

SELECT * FROM mergetree_testing.collapsingmergetree ;
```

SELECT * FROM mergetree_testing.colla En

Grid | Text

	ID	Count	Sign
1	a	64	1
2	b	32	-1
3	b	64	1
4	c	32	-1
5	d	128	1

***Figure 7.25**: Test data in CollapsingMergeTree table engine, after the merge operation*

In this example, since state rows are more than the cancel rows, the last state row is retained.

VersionedCollapsingMergeTree

This ClickHouse table engine is quite similar to the **CollapsingMergeTree** engine, but the algorithm used to delete/collapse the data is slightly different.

In addition to the sign column, a version column is required for this table engine. This version column is used to remove the rows properly along with the sign column. In **CollapsingMergeTree**, the ordering of the sign column in the inserted rows is important for collapsing the rows, whereas in **VersionedCollapsingMergeTree**, the presence of the version column removes this requirement.

At the time of inserting the data, if the version column is not declared as primary/sorting key, ClickHouse automatically adds the column for sorting the data before inserting it in the table. At the time of merge operation, the row pairs with same primary/sorting key, same version, and different signs are collapsed/deleted.

To illustrate this, let us begin by creating a table based on the **VersionedCollapsingMergeTree** engine:

```
CREATE TABLE IF NOT EXISTS mergetree_testing.
versionedcollapsingmergetree(
       ID String,
       Count UInt64,
       Sign Int8,
       Version UInt32)
ENGINE = VersionedCollapsingMergeTree(Sign, Version)
ORDER BY (ID);
```

The table has four columns (**ID**, **Count**, **Sign**, and **Version**) and is sorted by ID column. Since this table is created with the table engine as **VersionedCollapsingMergeTree**, the Version column is added automatically to the primary/sorting key. Now, let us insert some test data.

```
INSERT INTO mergetree_testing.versionedcollapsingmergetree VALUES
('a', 32, 1, 1),
('a', 32, 1, 1),
('a', 64, -1, 1),
('a', 256, 1, 2);
SELECT * FROM mergetree_testing.versionedcollapsingmergetree;
```

```
Now, let us force the data merge to see the final rows after the merge
operation.

OPTIMIZE TABLE mergetree_testing.versionedcollapsingmergetree FINAL;

SELECT * FROM mergetree_testing.versionedcollapsingmergetree;
```

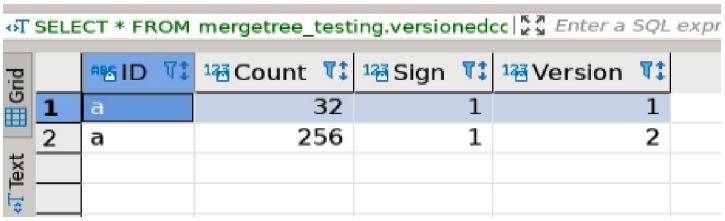


***Figure 7.26**: Test data in VersionedCollapsingMergeTree table engine, after the merge operation*

The engine replaces the rows from older version and keeps the rows from the latest version with state sign. So the additional version column makes it easier compared to **`CollapsingMergeTree`**.

Data replication

ClickHouse can also be set up in cluster mode. Each of the remote servers in the cluster should have:

- ClickHouse server installed in it (recommended to have the same version).
- Configure the remote servers in config file.
- Have the same set of ClickHouse configurations across the remote servers.
- Create the databases and tables manually that uses distributed engine or replicated engines.

ClickHouse supports data distribution (sharding) and data replication across the servers in a cluster. The following is a sample configuration in **`config.xml`** for a **`clickhouse`** cluster with three remote servers in the cluster configured for two shards:

```
<remote_servers>
    <simple_shard_replicas>
        <shard>
            <replica>
                <host>hostname_1</host>
                <port>9000</port>
            </replica>
```

```
                    <replica>
                <host>hostname_2</host>
                <port>9000</port>
            </replica>
        </shard>
        <shard>
            <replica>
                <host>hostname_2</host>
                <port>9000</port>
            </replica>
           <replica>
                <host>hostname_3</host>
                <port>9000</port>
            </replica>
        </shard>
    </simple_shard_replicas>
</remote_servers>
```

The first shard is configured to reside on the servers **hostname_1** and **hostname_2** and the second shard on **hostname_2** and **hostname_3** servers. More details on this will be discussed in *Chapter 10*.

ClickHouse supports data replication across its cluster via Zookeeper. ClickHouse uses Zookeeper to store metainformation of the replicas. So it is mandatory to have a Zookeeper cluster up and running to create replicated tables in ClickHouse.

It is required to execute the **CREATE**, **DROP**, **RENAME**, and **ATTACH** statements separately in each of the ClickHouse servers for the replicated tables. The **INSERT** query and the **ALTER** query can be run on any node of the ClickHouse server and it will be updated in the other nodes of the server. The data replication is asynchronous and the data inserted in one of the nodes may not be instantly available in other nodes.

Data replication is supported only in the MergeTree family of table engines. The following replicated table engines are available in ClickHouse:

- **ReplicatedMergeTree**
- **ReplicatedSummingMergeTree**
- **ReplicatedReplacingMergeTree**
- **ReplicatedAggregatingMergeTree**

- **ReplicatedCollapsingMergeTree**
- **ReplicatedVersionedCollapsingMergeTree**
- **ReplicatedGraphiteMergeTree**

Setting up the Zookeeper cluster is beyond the scope of this section and we shall not cover it here. For now, let us assume that a 3-node Zookeeper cluster is up and running. The first step is to update the configuration file of ClickHouse so that ClickHouse is aware of the Zookeeper cluster's existence and can use it for data replication.

The configuration file is available in **`/etc/clickhouse-server/config.xml`**. The file might require root user privilege to edit. The following section in the configuration file needs to be updated with Zookeeper cluster information:

```
<zookeeper>

    <node index="1">
        <host>zookeeper_host_1</host>
        <port>2181</port>
    </node>

    <node index="2">
        <host> zookeeper_host_2</host>
        <port>2181</port>
    </node>

    <node index="3">
        <host> zookeeper_host_3</host>
        <port>2181</port>
    </node>

</zookeeper>
```

Data is inserted in blocks; and at the time of writing, the duplicate data blocks are ignored and a block is written only once. At the time of data replication, the source data is transferred to the nodes and the merge operations are carried out in the individual nodes, which store the data.

Creating a replicated table

The replicated table engines for the MergeTree family have two additional parameters along with the regular table engine parameters. The first one is the Zookeeper path and the next one is the replica name.

Since the MergeTree table engine has no parameters, providing these two parameters are sufficient in case of the **ReplicatedMergeTree** table engine. If there are any parameters required for the non-replicated variant of the table engine (for example, **CollapsingMergeTree**), then for the replicated version of the table engine, the first two parameters will be the Zookeeper path and replica name and these parameters will be followed by the parameters required for the non-replicated engine variant.

The following **CREATE** statement can be used to create a replicated table based on the MergeTree engine:

```
CREATE TABLE student_info
(
    Name String,
    ID Int32,
    Weight Float32,
    Height Float32,
    DateOfBrith Date)

ENGINE = ReplicatedMergeTree('/clickhouse/tables/student_info',
'replica_name')
PARTITION BY toYYYYMM(DateOfBrith)
ORDER BY (Name, ID)
```

This statement will create a table called student info with five columns (**Name**, **ID**, **Weight**, **Height**, and **DateOfBirth**). The first parameter to the **ReplicatedMergeTree** is the Zookeeper path (where the replication metadata will be stored) and the second one is the replica name (which is used to identify and differentiate between the replica information from the nodes). The Zookeeper path should be different for different replicated tables and the replica name should be different for different ClickHouse server nodes. The rest of the clauses are similar to the MergeTree engine clauses.

This query has to be repeated in all the nodes of the ClickHouse cluster. But while inserting the data, it is sufficient to insert in any of the nodes and the data will be synced across the ClickHouse cluster.

Conclusion

So far, we have seen the different table engines from the MergeTree family. MergeTree family engines are the most robust and feature-rich engines in ClickHouse. The basic engine of this family is the MergeTree engine. The other engines are `ReplacingMergeTree`, `SummingMergeTree`, `AggregatingMergeTree`, `CollapsingMergeTree`, and `VersionedCollapsingMergeTree`. In order to have data replication, the replicated version of these engines is also available.

The next family of table engines in ClickHouse is the Log family of engines. Although they are not as powerful and feature rich as MergeTree, they are still useful in cases when many smaller tables are required. We shall look at the Log family engine and its variants in the next chapter.

Points to remember

- MergeTree family engines are the most powerful and feature-rich engines in ClickHouse.
- `ReplacingMergeTree` engine is useful when data deduplication based on the primary / sorting key is required.
- `SummingMergeTree` engine is used to sum the numeric columns with the same primary / sorting key.
- `AggregatingMergeTree` is used to store the aggregating states of the columns, which can be later used for getting aggregated values.
- `CollapsingMergeTree` is useful when their records are continuously changing. The data deletion is based on the sign column.
- `VersionedCollapsingMergeTree` is similar to `CollapsingMergeTree`, but an additional version column is used for deleting the old records.
- Data replication can be achieved with replicated versions of the MergeTree table engines. ClickHouse uses Zookeeper for data replication.

Multiple choice questions

1. **_____ family engine is recommended for low-load tasks.**
   a) MergeTree
   b) Log
   c) Both
   d) None

2. **At the time of writing the data to disk, the MergeTree engine sorts the data based on the primary/sorting key in _____ order.**

   a) Ascending

   b) Descending

3. **While storing the data in wide format, the MergeTree family stores different files in the disk for different columns.**

   a) True

   b) False

4. **Index is maintained in `<column_name>.mrk` file.**

   a) True

   b) False

5. **Primary key is mandatory for the MergeTree engine.**

   a) True

   b) False

6. **Column that can be specified as parameter to `ReplacingMergeTree()` should be of _____, `Date`, or `DateTime` data type.**

   a) Int*

   b) UInt*

   c) String

   d) FixedString(N)

7. **The `SummingMergeTree` accepts an optional parameter, which is a tuple of columns and the specified columns must not be in primary key columns.**

   a) True

   b) False

8. **`AggregatingMergeTree` table engine to store _____ of columns.**

   a) Aggregated values

   b) Aggregation functions

   c) Aggregating states

9. **While retrieving the aggregated values for a column defined as `avgState(toFloat64(Quantity)) as avgQuantity` from table based on `AggregatingMergeTree` engine, _____ is used.**

   a) avgFinal(avgQuantity)

   b) avgFinalState(avgQuantity)

   c) Merge(avgQuantity)

   d) avgMerge(avgQuantity)

10. **`CollapsingMergeTree` engine collapses (deletes) the rows with same sorting key based on _____ column.**

    a) Primary key

    b) Sorting key

    c) DateTime

    d) Sign

11. **In a table with the `CollapsingMergeTree` engine, the rows with '1' as a value in the sign columns are called state rows, and the rows with `-1` are called cancel rows.**

    a) True

    b) False

12. **In a table with `CollapsingMergeTree` engine, if the number of states and cancel rows are equal for the same sorting key, and the last row is a state row, the first state and the last cancel rows are retained.**

    a) True

    b) False

13. **In a table with `CollapsingMergeTree` engine, if the number of state rows are more than cancel rows or the same sorting key, then the last state row is retained in the table.**

    a) True

    b) False

14. **The ordering of the sign column in the inserted rows is important for collapsing the rows in the `VersionedCollapsingMergeTree` engine.**

    a) True

    b) False

Answers

1. **b**
2. **a**
3. **a**
4. **b**
5. **b**
6. **b**
7. **a**
8. **c**
9. **d**
10. **d**
11. **a**
12. **b**
13. **a**
14. **b**

CHAPTER 8

Table Engines – Log Family

Although the heavyweight tasks are accomplished using the MergeTree table engine family, using them for light-weight tasks may not be an optimal design choice. For such scenarios, the Log engine family is recommended to be used in ClickHouse instead of the MergeTree engine family.

In this chapter, we shall look at the light-weight table engine family in ClickHouse, which is Log. This engine is recommended for smaller tables (less than 1 million rows in them) from where the data can be read in the memory (RAM) as a whole.

Structure

In this chapter, we will discuss the following topics:

- StripeLog engine
- Log engine
- TinyLog engine

Objectives

The main objectives of this chapter are to:

- Learn about the Log engine and the common properties of the Log family engines.
- Understand the different engines belonging to the Log family, their properties, and usage.

Introduction

Log engines are light-weight table engines, which are commonly used for writing many smaller tables. They do not have many advanced features like MergeTree engines and are suitable for light-weight tasks. The common properties of Log table engines are:

- The table is locked while the **`INSERT`** queries are executed.
- When data is inserted into the table, the inserted data is appended towards the end of the table. Mutations are not supported (**`DELETE`** and **`UPDATE`** via **`ALTER`** statement).
- Data is neither sorted, like in MergeTree, nor indexed.
- Data is not written atomically, and this could result in data loss or data corruption if there are any errors while the data is being written.

The Log engine does not support replication and the data becomes irrecoverable if the table is corrupted.

TinyLog

TinyLog is the simplest table engine in the Log family. Being the one with minimal features, it can be used in scenarios where the data in the table is written once and read multiple times.

Parallel data reading is not supported in TinyLog and each column is written as a separate file in the disk. As a result, a file can have multiple descriptors. When compared with the Log and StripeLog, this is the slowest engine in the Log family when it comes to reading the data.

Let us begin by creating a sample table with the TinyLog engine, insert some data, and retrieve the data using the **`SELECT`** statement. The following is the syntax for creating a table with the TinyLog engine:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER
cluster]
(
    column1 [type1] [DEFAULT|MATERIALIZED|ALIAS expression1],
```

```
    column2 [type2] [DEFAULT|MATERIALIZED|ALIAS expression2],
    column3 [type3] [DEFAULT|MATERIALIZED|ALIAS expression3],
    ...
)
ENGINE = TinyLog
```

The **CREATE** syntax is quite similar to the other table engines, but the clauses supported are very minimal. **PRIMARY_KEY**, **ORDER_BY**, **PARTITION_BY**, or **SAMPLE_BY** clauses are not supported. Let us create a sample database and a table using the following SQL statement:

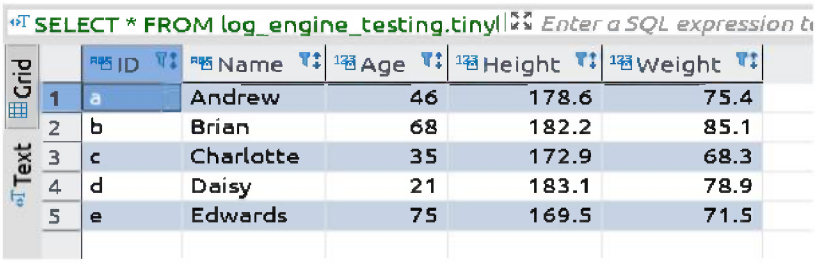
```
CREATE DATABASE log_engine_testing;


CREATE TABLE log_engine_testing.tinylog (
    ID String,
    Name String,
    Age UInt16,
    Height Float32,
    Weight Float32)
ENGINE = TinyLog;
```

The preceding statement will create a table with five columns (**ID**, **Name**, **Age**, **Height**, and **Weight**). Let us now insert some sample data into the table using the following **INSERT** statement:

```
INSERT INTO log_engine_testing.tinylog VALUES
('a','Andrew', 46, 178.6, 75.4),
('b','Brian', 68, 182.2, 85.1),
('c','Charlotte', 35, 172.9, 68.3),
('d','Daisy', 21, 183.1, 78.9),
('e','Edwards', 75, 169.5, 71.5);
```

Once the data is inserted successfully, we shall test the inserted data using the following **SELECT** statement:

```
SELECT * FROM log_engine_testing.tinylog;
```


SELECT * FROM log_engine_testing.tinyl | Enter a SQL expression t

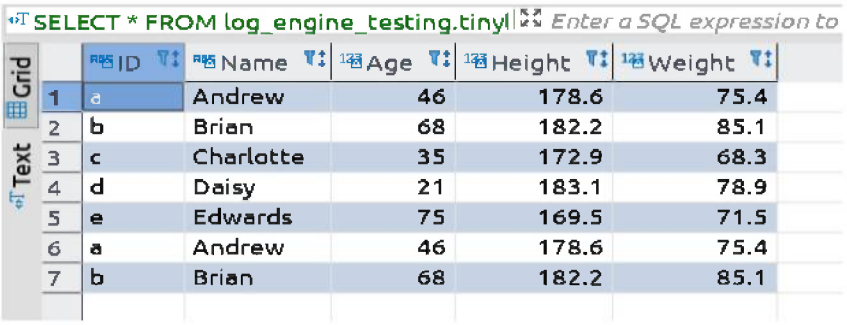
Grid | Text

	ID	Name	Age	Height	Weight
1	a	Andrew	46	178.6	75.4
2	b	Brian	68	182.2	85.1
3	c	Charlotte	35	172.9	68.3
4	d	Daisy	21	183.1	78.9
5	e	Edwards	75	169.5	71.5

***Figure 8.1**: Test data stored in TinyLog table engine*

Now, let us insert a duplicate data to see how the table reacts. To insert duplicate records, let us execute the following statement:

```
INSERT INTO log_engine_testing.tinylog VALUES
('a','Andrew', 46, 178.6, 75.4),
('b','Brian', 68, 182.2, 85.1);
SELECT * FROM log_engine_testing.tinylog;
```


SELECT * FROM log_engine_testing.tinyl | Enter a SQL expression to

Grid | Text

	ID	Name	Age	Height	Weight
1	a	Andrew	46	178.6	75.4
2	b	Brian	68	182.2	85.1
3	c	Charlotte	35	172.9	68.3
4	d	Daisy	21	183.1	78.9
5	e	Edwards	75	169.5	71.5
6	a	Andrew	46	178.6	75.4
7	b	Brian	68	182.2	85.1

***Figure 8.2**: Test data stored in TinyLog table engine*

We can see that the newly inserted rows are appended and not deduplicated. The **OPTIMIZE … FINAL** statement is not supported for the Log family engines. So, whatever data is inserted, it is appended to the end of the table/file. The TinyLog stores the data (columns) as compressed files.

Log engine

Log engine is another table engine from the Log family . The key difference from the TinyLog engine is that Log engine stores mark the information in column files. This mark information has the file offsets using which the ClickHouse will be able to skip the specified number of rows at the time of reading.

This enables the application to read the data from multiple threads, simultaneously. So, the Log engine is comparatively faster than the TinyLog engine when it comes to read operations. For write operations, the table is locked and no other operations are permitted until the lock is released. The Log engine can be used in scenarios similar to the TinyLog engine but if simultaneous read operations are required.

Let us begin by creating a sample table with the Log engine. The following syntax can be used to create a table with a Log engine:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER
cluster]
(
    column1 [type1] [DEFAULT|MATERIALIZED|ALIAS expression1],
    column2 [type2] [DEFAULT|MATERIALIZED|ALIAS expression2],
    column3 [type3] [DEFAULT|MATERIALIZED|ALIAS expression3],
    ...
)
ENGINE = Log
```

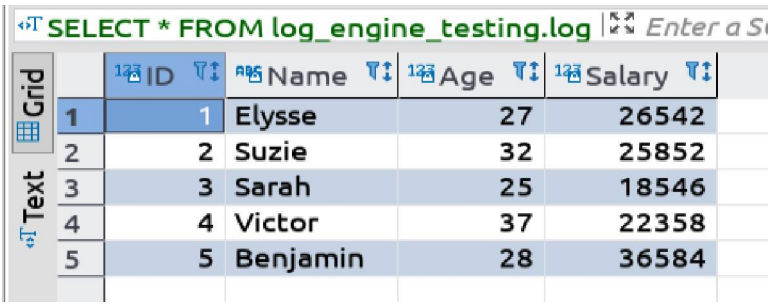
The syntax is very similar to the TinyLog engine's syntax for creating the table. Similar to TinyLog, the **PRIMARY_KEY**, **ORDER_BY**, **PARTITION_BY**, or **SAMPLE_BY** clauses are not supported. Now let us create a new table using the following statement:

```
CREATE TABLE log_engine_testing.log (
    ID UInt16,
    Name String,
    Age UInt16,
    Salary UInt64)
ENGINE = Log;
```

A new table is created with Log engine and has four columns (**ID**, **Name**, **Age**, and **Salary**). Let us test this by inserting few test data using the following statement:

```
INSERT INTO log_engine_testing.log VALUES
(1, 'Elysse', 27, 26542),
(2, 'Suzie', 32, 25852),
(3, 'Sarah', 25, 18546),
(4, 'Victor', 37, 22358),
(5, 'Benjamin', 28, 36584);
```

```
SELECT * FROM log_engine_testing.log;
```



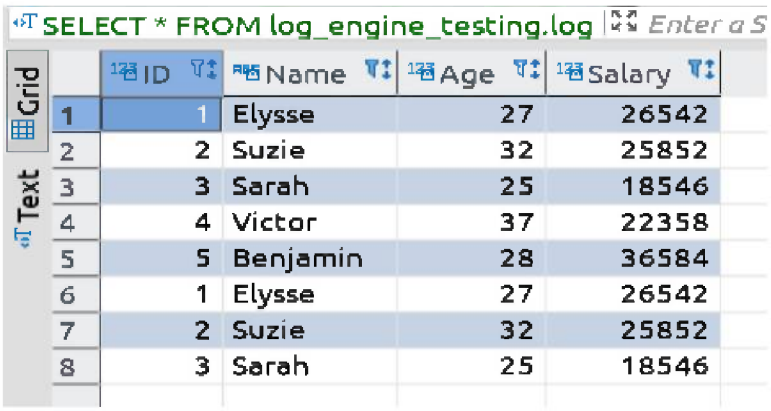
	ID	Name	Age	Salary
1	1	Elysse	27	26542
2	2	Suzie	32	25852
3	3	Sarah	25	18546
4	4	Victor	37	22358
5	5	Benjamin	28	36584

***Figure 8.3**: Test data stored in Log table engine*

The data is inserted successfully and we can query the data from the table. Now, let us try inserting few duplicate rows into the table:

```
INSERT INTO log_engine_testing.log VALUES
(1, 'Elysse', 27, 26542),
(2, 'Suzie', 32, 25852),
(3, 'Sarah', 25, 18546);
SELECT * FROM log_engine_testing.log;
```

The expected behavior is similar to that of TinyLog as the table does not support the primary key or ordering by column values, and the ordering is not guaranteed due to the multi-threaded (parallel) data retrieval.



	ID	Name	Age	Salary
1	1	Elysse	27	26542
2	2	Suzie	32	25852
3	3	Sarah	25	18546
4	4	Victor	37	22358
5	5	Benjamin	28	36584
6	1	Elysse	27	26542
7	2	Suzie	32	25852
8	3	Sarah	25	18546

***Figure 8.4**: Test data stored in Log table engine*

The `OPTIMIZE … FINAL` statement is not supported for the Log engine table and the Log engine stores the data (columns) as compressed files.

StripeLog engine

StripeLog is the third and final engine type in the Log family. The primary difference between StripeLog and the rest of the table engines in the log family is that StripeLog engine writes the data to a single file instead of separate files for columns like in Log and TinyLog engines.

This engine maintains two files per table. The first one is the data file, which holds the data in the table, and the next one is the file containing the mark information, which contains the offset values for each column in the table. The file containing the mark information is updated for each block of data that is inserted. It is used for the parallel reading of data. The following is the syntax for creating a table using the StripeLog engine:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER
cluster]
(
    column1 [type1] [DEFAULT|MATERIALIZED|ALIAS expression1],
    column2 [type2] [DEFAULT|MATERIALIZED|ALIAS expression2],
    column3 [type3] [DEFAULT|MATERIALIZED|ALIAS expression3],
    ...
)
ENGINE = StripeLog
```

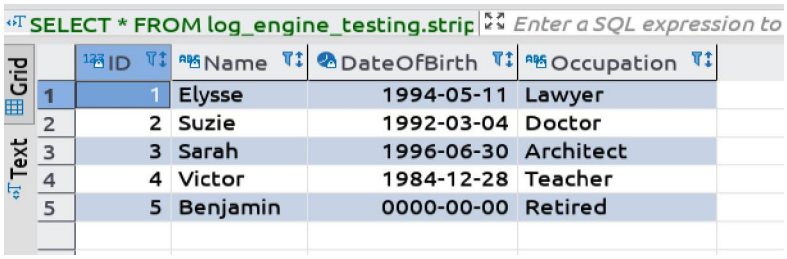
The syntax to create the table is quite similar to the other table engines of the Log family. Similar to the other table engines belonging to the Log family, the **PRIMARY_KEY**, **ORDER_BY**, **PARTITION_BY**, or **SAMPLE_BY** clauses are not supported in the StripeLog engine too. Now let us create a new table using the following statement:

```
CREATE TABLE log_engine_testing.stripelog (
ID UInt16,
Name String,
DateOfBirth Date,
Occupation String)
ENGINE = StripeLog;
```

When the preceding statement is executed, the table is created with four columns (**ID**, **Name**, **DateOfBirth**, and **Occupation**). The next step is to test the table by inserting some test data and by querying the table to view the inserted data.

```
INSERT INTO log_engine_testing.stripelog VALUES
(1, 'Elysse', '1994-05-11', 'Lawyer'),
```

```
(2, 'Suzie', '1992-03-04', 'Doctor'),
(3, 'Sarah', '1996-06-30', 'Architect'),
(4, 'Victor', '1984-12-28', 'Teacher'),
(5, 'Benjamin', '1956-11-01', 'Retired');

SELECT * FROM log_engine_testing.stripelog;
```

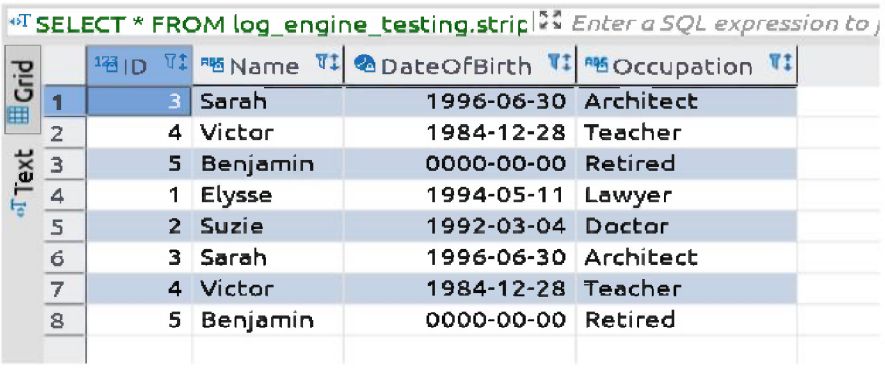


	ID	Name	DateOfBirth	Occupation
1	1	Elysse	1994-05-11	Lawyer
2	2	Suzie	1992-03-04	Doctor
3	3	Sarah	1996-06-30	Architect
4	4	Victor	1984-12-28	Teacher
5	5	Benjamin	0000-00-00	Retired

Figure 8.5: Test data stored in StripeLog table engine

Once the data is inserted successfully, we will be able to query and retrieve the data from the table. Now, let us try inserting the duplicate rows to the table and query the table:

```
INSERT INTO log_engine_testing.stripelog VALUES
(3, 'Sarah', '1996-06-30', 'Architect'),
(4, 'Victor', '1984-12-28', 'Teacher'),
(5, 'Benjamin', '1956-11-01', 'Retired');

SELECT * FROM log_engine_testing.stripelog;
```



	ID	Name	DateOfBirth	Occupation
1	3	Sarah	1996-06-30	Architect
2	4	Victor	1984-12-28	Teacher
3	5	Benjamin	0000-00-00	Retired
4	1	Elysse	1994-05-11	Lawyer
5	2	Suzie	1992-03-04	Doctor
6	3	Sarah	1996-06-30	Architect
7	4	Victor	1984-12-28	Teacher
8	5	Benjamin	0000-00-00	Retired

Figure 8.6: Test data stored in StripeLog table engine

The expected behavior is similar to that of other table engines from the Log family. Since the table engine does not support primary key or ordering by column values,

the ordering is not guaranteed due to multi-threaded (parallel) data retrieval and has to be done manually via the `ORDER BY` clause.

Conclusion

The Log engine, which is a light-weight family of an engine with minimal features, is explored in this chapter. Log engines are rarely used, but provide a better alternative to MergeTree family engines when it comes to maintaining multiple small tables.

We have seen both table engine families in ClickHouse. Based on the usage scenario, it is advisable to choose the appropriate engines. In the forthcoming chapter, we shall see how to integrate ClickHouse with different data sources.

Points to remember

- Log family engine is a light-weight engine with fewer features compared with MergeTree and is used to hold a small amount of data and when multiple small tables are required.
- The table is locked during `INSERT` operation.
- Primary keys and indexing are not supported.
- TinyLog is the simplest engine and doesn't support parallel read operations.
- Log engine is similar to TinyLog, but supports parallel read operations.
- StripeLog supports parallel reads and data is stored in a single data file.

Multiple-choice questions

1. **In the Log family engine, the tables are locked during _____ operation.**
   a) SELECT
   b) DELETE
   c) UPDATE
   d) INSERT

2. **Log family engine supports _____.**
   a) Dense indexing
   b) Sparse indexing
   c) None of the indexing types
   d) Both a and b

3. **The simplest engine of the Log family is _____**
   a) TinyLog
   b) Log
   c) StripeLog

4. **Parallel data reading is supported in _____ and _____ engines.**
   a) TinyLog and Log
   b) Log and StripeLog
   c) StripeLog and TinyLog

5. **_____ engine stores the data in a single file.**
   a) TinyLog
   b) Log
   c) StripeLog

6. **Log family engines support data replication.**
   a) True
   b) False

7. **Primary keys are supported in the Log family engines.**
   a) True
   b) False

Answers

1. **d**
2. **c**
3. **a**
4. **b**
5. **c**
6. **b**
7. **b**

CHAPTER 9
External Data Sources

More often in real-world applications, the data is ingested into ClickHouse from external data sources. ClickHouse supports multiple options to integrate with external data sources.

ClickHouse supports data ingestion from various sources like Kafka, MySQL, PostgreSQL, and from many other data sources/stores via JDBC and ODBC connectivity options. This makes it easier to fetch the data from different sources into ClickHouse or work with multiple data sources via ClickHouse.

Structure

In this chapter, we will discuss how to integrate ClickHouse with:

- Apache Kafka
- MySQL database
- PostgreSQL
- JDBC
- HDFS
- S3

Objectives

After going through this chapter, you will learn to ingest data from multiple data sources and get hands-on about how to integrate ClickHouse from multiple external data sources.

Introduction

ClickHouse has separate engines and supports multiple input and output formats for handling the data from multiple data sources (more information on supported input and output formats are available in the official documentation).

The integration with external data sources is achieved via SQL statements (**CREATE**/**ALTER** table). The configured external data source may appear as a normal ClickHouse table and can be queried from ClickHouse. In most of the integration engines, the queries are executed in the system in which the integrated data source exists.

Kafka

Apache Kafka is an open-source, distributed even streaming platform, which offers benefits like scalability, high throughput, and high availability.

Kafka is installed in a group of machines and that group is called as Kafka cluster. Individual machines in the cluster responsible for maintaining the data is known as a broker. Stream of messages from the same category are called topics and are published by producers. Consumers subscribe to topics produced by the producers.

ClickHouse Kafka engine is capable of subscribing to any particular topic from a Kafka instance and read the data from the topic. The message delivered to the engine is automatically tracked, which means the table reads the message from the topic only once. The settings and parameters are provided in the **CREATE** statement while creating the table.

The following syntax can be used to create a table with Kafka engine:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER
cluster_name]
(
    column1 [data_type1] [DEFAULT|MATERIALIZED|ALIAS expression_1],
    column2 [data_type2] [DEFAULT|MATERIALIZED|ALIAS expression_2],
    column3 [data_type3] [DEFAULT|MATERIALIZED|ALIAS expression_3],
    ...
)
```

```
ENGINE = Kafka()
SETTINGS
kafka_broker_list = 'broker_host:port',
kafka_topic_list = 'topic1,topic2,...',
kafka_group_name = 'kafka_group_name',
kafka_format = 'data_format',
[kafka_row_delimiter = 'delimiter_symbol',]
[kafka_schema = '',]
[kafka_num_consumers = N,]
[kafka_max_block_size = 0,]
[kafka_skip_broken_messages = N,]
[kafka_commit_every_batch = 0]
```

The **CREATE** statement is similar to the other table engines. The following are the parameters to create and configure the table with the Kafka engine:

- **kafka_broker_list**: This is the list of Kafka brokers machine address with port from which the message will be consumed (for example – **kafka.broker1:9092**)
- **kafka_topic_list**: This is the list of Kafka topics from which the messages are to be consumed.
- **kafka_group_name**: This is the Kafka consumer group name. Messages are available once for each consumer group.
- **kafka_format**: This is the data format of the incoming messages, which is used to parse and store in the table (for example – **JsonEachRow**)
- **kafka_row_delimiter**: This is the delimiter character for rows in messages.
- **kafka_schema**: This is the schema if the messages require schema for parsing.
- **kafka_num_consumers**: This is the number of consumers per group, to increase throughput. The number of consumers should not be greater than the number of partitions in the topic.
- **kafka_max_block_size**: This is the maximum batch size for polling the topic.
- **kafka_skip_broken_messages**: This is the number of schema incompatible messages that can be skipped per block.

- **kafka_commit_every_batch**: This means commit every batch instead of the whole block.

To store the data populated in the Kafka engine table, it is a common practice to create a materialized view that reads the data from the Kafka engine table. The materialized view is set to insert the data from the Kafka engine table to a final table with the desired structure. So, the steps involved are:

1. Create a ClickHouse table with Kafka engine. This table will act as a data stream.
2. Create a final table with desired columns.
3. Create a materialized view that reads from the Kafka engine.
4. The materialized view will automatically insert the data that was read from Kafka into the final table.

The following example illustrates setting up ClickHouse to read the data from Kafka and store it in the ClickHouse table. The messages are in attribute–value pair. Attributes correspond to columns and values correspond to the values that correspond to the actual value for that particular attribute.

```
CREATE TABLE external_data_sources.kafka_testing (
    Name String,
    Age UInt64,
    Height UInt64,
    Weight UInt64)
  ENGINE = Kafka
  SETTINGS kafka_broker_list = 'localhost:9092',
           kafka_topic_list = 'clickhouse_kafka_topic',
           kafka_group_name = 'clickhouse_kafka_group_1',
           kafka_format = 'JSONEachRow',
           kafka_num_consumers = 1;
```

The preceding statement creates a table **kafka_testing** under database **external_data_sources**. The table contains four columns (**Name**, **Age**, **Height**, and **Weight**) and listens to the topic **clickhouse_kafka_topic** via **clickhouse_kafka_group_1** consumer group. Note that we have to create the topic in our virtual machine's Kafka setup and publish messages under the topic manually. Let us now create a final table with MergeTree engine:

Note: Steps for setting up Kafka and publishing sample messages are available in the *Appendix* section.

```
CREATE TABLE external_data_sources.kafka_testing_final (
      Name String,
      Age UInt64,
      Height UInt64,
      Weight UInt64)
      ENGINE = MergeTree()
      ORDER BY (Name, Age, Height, Weight)
      index_granularity=2;
```

The final step is to create the materialized view that can read the data from the Kafka table and insert it into the final table.

```
CREATE MATERIALIZED VIEW external_data_sources.kafka_materialized_view
TO external_data_sources.kafka_testing_final
AS SELECT Name,
toUInt64OrZero(Age) AS Age,
toUInt64OrZero(Height) AS Height,
toUInt64OrZero(Weight) AS Weight
FROM external_data_sources.kafka_testing;
```

The materialized view fetches the data from Kafka, converts the data format, and inserts the data to the final table. To test this, we have to publish messages on the specified topic. Steps to publish messages in Kafka are available in the *Appendix* section.

MySQL

ClickHouse has a robust engine for MySQL database, which makes it possible to connect to MySQL database and execute SELECT queries. This makes it easier to read the data from the MySQL table and store it in ClickHouse. The SQL queries are to be written in the ClickHouse dialect (not MySQL's SQL dialect) and ClickHouse will automatically try to cast the data types of the columns.

The syntax for creating a table with MySQL engine is as follows:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER
cluster_name]
(
    column1 [data_type1] [DEFAULT|MATERIALIZED|ALIAS expression1],
    column2 [data_type2] [DEFAULT|MATERIALIZED|ALIAS expression2],
```

```
    column3 [data_type3] [DEFAULT|MATERIALIZED|ALIAS expression3],
    ...
)
ENGINE = MySQL('mysql_host_name:port_number', 'mysql_database_name',
'mysql_table_name', 'mysql_user_name', 'mysql_password');
```

The **CREATE** statement is similar to the ones seen before. The parameters required to create a MySQL engine in ClickHouse are as follows:

- **mysql_hostname**: This is the hostname where MySQL is installed.
- **port_number**: This is the port used to communicate with the MySQL server.
- **mysql_database_name**: This is the database name in the MySQL server.
- **mysql_table_name**: This is the table in MySQL, which would be read by ClickHouse.
- **mysql_user_name**: – This is the user name to login to the MySQL server.
- **mysql_password**: – This is the password for the user.

For illustrating the working of this engine, we shall make use of the data from the **employees** table in the 'employees' database in the MySQL server that was set up earlier. We shall create a table in ClickHouse with MySQL engine, read the data from the **employees** table in MySQL and finally populate the data in a separate table in ClickHouse with MergeTree engine. To create the table, the following syntax can be used in ClickHouse:

```
CREATE DATABASE employees;


CREATE TABLE employees.employees (
      emp_no UInt64,
      first_name Nullable(String),
      last_name Nullable(String),
      hire_date Nullable(Date)
)
ENGINE = MySQL('127.0.0.1:3306', 'employees', 'employees', 'mysql_user',
'clickhouse');
```

Note: Steps for setting up MySQL and loading the employees database are available in the *Appendix* section.

This will create a table called **employees** in **employees** database with MySQL engine as backend. The table will have four columns (**emp_no**, **first_name**, **last_name**, and **birth_date**). Let us execute a simple **SELECT** statement on the newly created table.

Sample output from the query is displayed in the following *Figure 9.1*:

```
SELECT * FROM employees.employees WHERE first_name='Georgi' LIMIT 5;
```

***Figure 9.1**: SELECT query results on the table using MySQL engine*

Let us now create a new table with the MergeTree engine and insert the rows from the MySQL table. The new table will have a similar structure to the MySQL engine table. The following statement can be used to create the MergeTree engine table:

```
CREATE TABLE employees.employees_mergetree (
      emp_no UInt64,
      first_name Nullable(String),
      last_name Nullable(String),
      hire_date Nullable(Date)
)
ENGINE = MergeTree()
PRIMARY KEY emp_no
ORDER BY (emp_no);
```

Once the table is created, let us insert the data from the MySQL database. The following **INSERT** statement can be used to read the data from the MySQL table and insert it into the MergeTree table:

```
INSERT INTO employees.employees_mergetree SELECT * FROM employees.
employees;
```

Let us read some sample data from the table with the following **SELECT** statement:

```
SELECT * FROM employees.employees_mergetree ORDER BY hire_date LIMIT 5 ;
```

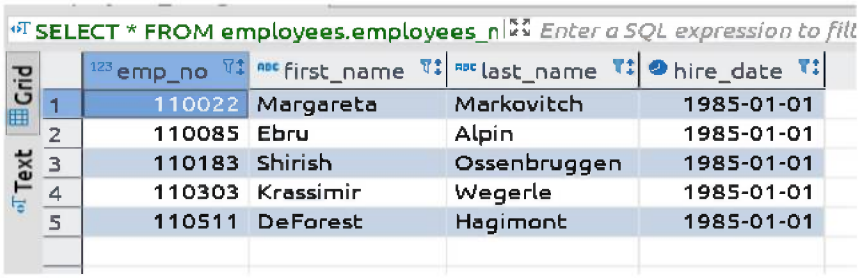


***Figure 9.2**: SELECT query results data populated in the new MergeTree table*

So, we have successfully created a table with MySQL table engine, read the data from MySQL database, and populated it in ClickHouse native table engine.

PostgreSQL

Similar to the MySQL engine, ClickHouse has the PostgreSQL engine, which can be used to connect to the PostgreSQL database server, read and write data in it. Syntax to create a table is as follows:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER
cluster_name]


(   Column1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1] [TTL expr1],
    column2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2] [TTL expr2],
    ...)


ENGINE = PostgreSQL('postgres_host:port', 'postgres_database_name',
'postgres_table_name', 'postgres_user_name', 'password_for_user_name')
```

The parameters for creating table with PostgreSQL engine are:

- **`postgres_hostname`**: This is the hostname where PostgreSQL is installed.
- **`port`**: – **`port`**: This is used to communicate with PostgreSQL server.
- **`postgres_database_name`**: This is the database name in PostgreSQL server.

Note: Steps for setting up PostgreSQL and loading the dvdrental database are available in the Appendix section.

- **postgres_table_name**: This is the table in PostgreSQL, which would be read by ClickHouse.
- **postgres_user_name**: This is the user name to login to PostgresSQL server.
- **postgres_password**: This is the password for the user.

Let us create a table from the **dvdrental** database that is available in PostgreSQL in ClickHouse to test the working of the PostgreSQL.

```
CREATE DATABASE dvdrental;


CREATE TABLE dvdrental.actor (
    actor_id UInt64,
    first_name Nullable(String),
    last_name Nullable(String),
    last_update Nullable(Date)
)
ENGINE = PostgreSQL('127.0.0.1:5432', 'dvdrental', 'actor', 'postgres',
'postgres123');
```

The preceding SQL statements will create a database named **dvdrental** and a table named **actor** in the **dvdrental** database. The table has four columns named **actor_id**, **first_name**, **last_name**, and **last_update**. Let us read the data from the newly created table:

```
SELECT * FROM dvdrental.actor WHERE first_name = 'Russell';
```

SELECT * FROM dvdrental.actor WHERE first_na | Enter a SQL expression to filter results (us

Grid | Text

	actor_id	first_name	last_name	last_update
1	112	Russell	Bacall	2013-05-26
2	149	Russell	Temple	2013-05-26
3	183	Russell	Close	2013-05-26

***Figure 9.3**: SELECT query results from postgres engine table*

JDBC

ClickHouse can be interfaced with other external database management systems via **JDBC** (**Java Database Connectivity**). We shall use this engine to connect with the MySQL database (although we have seen a dedicated MySQL engine, this is an alternate way to connect). This engine requires another application called the

clickhouse-jdbc-bridge. Details on the installation and running of the application are available in the *Appendix* section.

The syntax to create ClickHouse table with JDBC engine is as follows:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER
cluster_name]
(
    column1 [data_type1] [DEFAULT|MATERIALIZED|ALIAS expression1],
    column2 [data_type2] [DEFAULT|MATERIALIZED|ALIAS expression2],
    column3 [data_type3] [DEFAULT|MATERIALIZED|ALIAS expression3],
    ...
)
ENGINE = JDBC(database_uri, database_name, table_name)
```

The table is created with the familiar CREATE statement. The following are the parameters to the JDBC engine in ClickHouse:

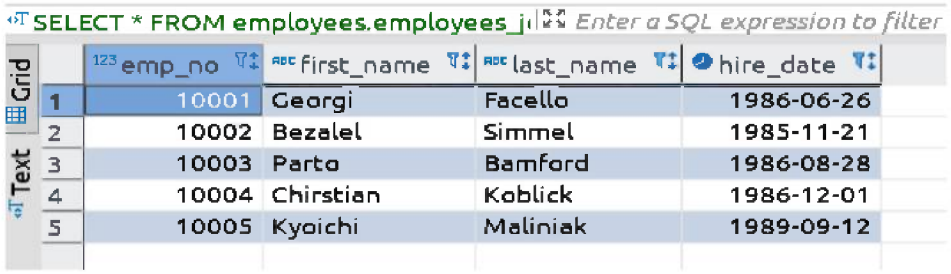
- **database_uri**: This is the URI of the DBMS to be connected with ClickHouse. For MySQL, the example URI to be used is **jdbc:mysql://localhost:3306/?user=mysql_user&password=clickhouse**.
- **database_name**: This is the name of the database to be connected.
- **table_name**: This is the name of the table to be connected.

Let us now create a ClickHouse table with the JDBC engine (connecting to MySQL used in the previous example). Once the table is created, we shall test by querying the newly created table. The following **CREATE** statement can be used to create the table with the JDBC engine:

```
CREATE TABLE employees.employees_jdbc (
      emp_no UInt64,
      first_name Nullable(String),
      last_name Nullable(String),
      hire_date Nullable(Date)
)
ENGINE = JDBC('jdbc:mysql://localhost:3306/?user=mysql_
user&password=clickhouse', 'employees', 'employees')
```

Once the table is created in ClickHouse, let us now read sample data from the table:

```
SELECT * FROM employees.employees_jdbc LIMIT 5;
```

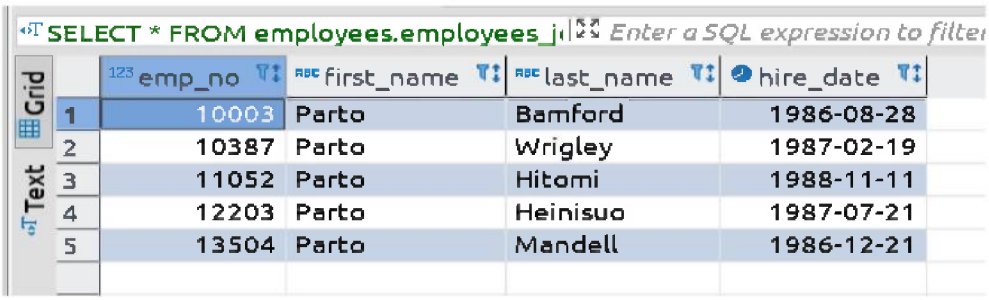


	emp_no	first_name	last_name	hire_date
1	10001	Georgi	Facello	1986-06-26
2	10002	Bezalel	Simmel	1985-11-21
3	10003	Parto	Bamford	1986-08-28
4	10004	Chirstian	Koblick	1986-12-01
5	10005	Kyoichi	Maliniak	1989-09-12

***Figure 9.4**: SELECT query results on the table using JDBC engine*

Similar to the earlier example, we shall create a table with the MergeTree engine and **INSERT** data from this table using the JDBC engine.

```
CREATE TABLE employees.employees_jdbc_mergetree (
       emp_no UInt64,
       first_name Nullable(String),
       last_name Nullable(String),
       hire_date Nullable(Date)
)
ENGINE = MergeTree()
PRIMARY KEY emp_no
ORDER BY (emp_no);
INSERT INTO employees.employees_jdbc_mergetree SELECT * FROM employees.
employees_jdbc;
SELECT * FROM employees.employees_jdbc_mergetree WHERE first_name='Parto'
LIMIT 5;
```



	emp_no	first_name	last_name	hire_date
1	10003	Parto	Bamford	1986-08-28
2	10387	Parto	Wrigley	1987-02-19
3	11052	Parto	Hitomi	1988-11-11
4	12203	Parto	Heinisuo	1987-07-21
5	13504	Parto	Mandell	1986-12-21

***Figure 9.5**: SELECT query results on MergeTree table.*

HDFS

Hadoop Distributed File System (**HDFS**) is a highly fault-tolerant system used to store huge amounts of data in commodity hardware. The HDFS engine in ClickHouse can read structured data from the HDFS file location and perform operations on them.

Although read and write operations are supported, some regular features like **ALTER** statements, indexing, replication, and sampling are not supported in this engine. To create a table with the **HDFS** engine, the following syntax can be used:

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name
(
    column1 [data_type1] [DEFAULT|MATERIALIZED|ALIAS expression1],
    column2 [data_type2] [DEFAULT|MATERIALIZED|ALIAS expression2],
    column3 [data_type3] [DEFAULT|MATERIALIZED|ALIAS expression3],
    ... )
ENGINE = HDFS(URI, format)
```

The parameters required for the engine are:

- **URI**: – The file URI in HDFS
- **Format**: – File format (for example, CSV and TSV)

To create a sample table, the following statement can be used:

```
CREATE TABLE hdfs_engine.sample_hdfs_table (
      emp_no UInt64,
      first_name String,
      last_name String,
      hire_date Date )
ENGINE=HDFS('hdfs://hdfs1:9000/sample_data', 'CSV')
```

The preceding statement will create a table named `sample_hdfs_table` with four columns (`emp_no`, `first_name`, `last_name`, and `hire_date`). The data format specified here is CSV. Regular `SELECT` and `INSERT` statements can be used to read and write the data into the table.

Amazon S3

Amazon **S3** (**Simple Storage Service**) is a scalable cloud storage solution based on object storage architecture. The data is stored in containers called buckets. Objects

are the entities stored in buckets and they consist of object data and metadata. Objects within buckets are identified by unique identifiers, which are called keys. ClickHouse has integration with S3 so that we can create a ClickHouse table to read and write the data stored in the S3 bucket. The syntax for creating a table with the S3 engine is as follows:

```
CREATE TABLE s3_engine_table
(name String,
 value UInt32)
ENGINE = S3(S3_path, format, structure, [compression])
```

The following are the parameters to the S3 engine:

- **S3_path**: — This is the URL of the S3 bucket where the data resides.
- **format**: This is the file format in S3, which includes CSV, TSV, and so on.
- **structure**: This is the table structure defined inside a string.
- **compression**: — This is the compression used for the file stored in S3.

Before we start using S3 engine, the following endpoint settings must be set in the **config.xml** file:

```
<s3>
<endpoint-name>
<endpoint> {endpoint url} </endpoint>
<access_key_id>ACCESS_KEY_ID</access_key_id>
<secret_access_key>SECRET_ACCESS_KEY</secret_access_key>
</endpoint-name>
</s3>
```

To create a table with the **S3** engine, the following statement can be used:

```
CREATE TABLE s3_engine_table
(name String,
value UInt32)
ENGINE=S3('https://example-storage.net/my-bucket/test-data.tsv.gz',
'TSV', 'name String, value UInt32', zstd)
```

This will create a table with the S3 engine and will contain a 'name' and 'value' column. The data format specified here is TSV. Regular **SELECT** and **INSERT** statements can be used to read and write the data into the table while mutations, indexing, and sampling are not supported.

Conclusion

ClickHouse has a robust set of engines to interface the application with external data sources. This makes it easier for the end-users to fetch and manipulate data from the external sources into ClickHouse. Apart from the engines to interface with external data sources, ClickHouse has special engine types for multiple usage scenarios. The upcoming chapter will cover the special table engines and their usage.

Points to remember

- ClickHouse has multiple engines with which we can interface the ClickHouse to external data sources.
- Kafka engine is capable of fetching and storing streaming data from Kafka.
- There is an in-built MySQL engine that can be used to read the data directly from the MySQL database.
- Similarly, there is an in-built PostgreSQL engine, that can be used to read the data directly from the PostgreSQL database.
- JDBC and ODBC engines are available with which we can interface ClickHouse to multiple different databases.
- HDFS engine makes it possible to work with the Hadoop ecosystem.
- Similar to the HDFS engine, there is an S3 that enables ClickHouse to work with Amazon S3.

Multiple choice questions

1. **In most of the integration engines, the queries are executed in the system in which the integrated data source exists.**

   a) True

   b) False

2. **To store the data populated in the Kafka engine table, it is a common practice to create a _____, which stores the read data in the MergeTree table.**

   a) View

   b) Dictionary

   c) Materialized view

   d) Table based on StripeLog engine

3. **MySQL engine in ClickHouse enables us to connect to MySQL database and execute _____ queries on the MySQL database.**
   a) SELECT, INSERT
   b) SELECT, DELETE, UPDATE
   c) SELECT, UPDATE
   d) SELECT

4. **PostgreSQL engine in ClickHouse enables us to connect to PostgreSQL database and execute _____ queries on the MySQL database.**
   a) SELECT, INSERT
   b) SELECT, DELETE, UPDATE
   c) SELECT, UPDATE
   d) SELECT

5. **ClickHouse can be interfaced with other external database management systems via _____.**
   a) ODBC
   b) JDBC
   c) ODBC and JDBC
   d) None of the options

6. **Which of the following is not supported in the HDFS engine?**
   a) SELECT
   b) INSERT
   c) ALTER
   d) All the options

7. **Which of the following is supported in the S3 engine?**
   a) Sampling
   b) INSERT statement
   c) Mutations
   d) Indexing

Answers

1. **a**
2. **c**
3. **d**
4. **a**
5. **c**
6. **c**
7. **b**

CHAPTER 10

Special Engines

We have so far explored the multiple engine types available in ClickHouse to store the data and to interface with external data sources and databases. Apart from MergeTree and Log family engines, there are other engines available that can be used to store the data in RAM, read from the file on the disk, or directly from a web server, and so on. This helps in saving a lot of time spent on implementing these functionalities from scratch and helps us get started very quickly for such scenarios. In this chapter, we shall take a look at other engines available in ClickHouse that can be helpful in special scenarios.

Structure

In this chapter, we will discuss the following ClickHouse engines:

- Dictionary
- File
- Merge
- Set
- Memory
- Buffer

- URL
- Distributed

Objectives

The main objectives of this chapter are to:

- Learn to create, write, and read data from special table engines
- Learn the unique functionalities of different special engines
- Get hands-on with the special engines

Introduction

ClickHouse comes with multiple special table engines intended to be used in specific scenarios. These engines help solve multiple real-world problems quickly. For example, it is possible to load the data from the csv file into ClickHouse directly, without any third-party tools or applications, store the data in RAM, instead of disk, or periodically flush the data from the memory onto the disk.

Dictionary

ClickHouse dictionaries, as their name suggests, are key–attributes mapping. Dictionaries are widely used as reference lists and they help in speeding up the JOIN operations when compared with regular tables. ClickHouse dictionaries can be created using the **CREATE** statement and can be used like any other table using SQL. Dictionaries are stored in RAM either fully or partially depending on the configuration. This helps in considerable speedup on operations performed on the Dictionary.

Dictionaries can be created from different data sources like a text file, an external database, or even from an HTTP source. The following syntax can be used for creating a dictionary in ClickHouse:

```
CREATE DICTIONARY [database_name.]dict_name
(
    column1 data_type1,
    column2 data_type2,
    column3 data_type3,
       ...
)
```

```
PRIMARY KEY ...
SOURCE(...)
LAYOUT(...)
LIFETIME(...)
```

The **CREATE** statement has a mandatory **DICTIONARY** keyword, and the column/data type definitions are similar to the ones we have seen in earlier chapters. The following parameters are required to configure the dictionary that is created:

PRIMARY KEY

The **PRIMARY KEY** are the columns containing the unique keys of the dictionary. The primary key should be of UInt64 data type if it is a single column or can be of different types in the case of a composite key.

LAYOUT

This parameter specifies the way the dictionary is stored in the memory. The following layouts are supported in ClickHouse dictionaries:

CACHE

The **CACHE** layout stores a fixed number of cells in the memory, which contains frequently used elements from the dictionary. While reading the data, the cached elements are searched first and if the keys are not found, then it is searched from the source and updated in the cache. The cache cell limit and the lifetime of the data in the cache can be set. The syntax for specifying cache layout is given as follows:

```
LAYOUT(CACHE(SIZE_IN_CELLS 1000000000))
```

COMPLEX KEY CACHE

Again, this is similar to cache but used for dictionaries with composite keys (where the key can be a tuple from any type of field). The syntax for specifying cache layout is given as follows:

```
LAYOUT(COMPLEX_KEY_CACHE(SIZE_IN_CELLS 1000000000))
```

FLAT

Dictionary is stored in the form of flat arrays. This way of storing the dictionary is the optimal way among all the available layouts and it is recommended for usage since it supports all the available sources as well. The following is the syntax for specifying a flat layout:

```
LAYOUT(FLAT())
```

HASHED

In this layout, the keys are hashed and the dictionary is stored as a hash table in memory. This type of layout can support a large number of keys and supports all the available sources. The following is the syntax for specifying a hashed layout:

```
LAYOUT(HASHSED())
```

COMPLEX KEY HASHED

This layout is quite similar to the hashed layout but it is meant for dictionaries with composite keys. The following is the syntax for specifying a complex key hashed layout:

```
LAYOUT(COMPLEX_KEY_HASHSED())
```

RANGE HASHED

In this layout, the dictionary is stored as a hash table with an ordered array of ranges and their corresponding values. Along with the layout, the minimum and maximum of the range should also be specified. The following is the syntax for specifying a ranged hashed layout:

```
LAYOUT(RANGE_HASHED())
RANGE(MIN column1 MAX column2)
```

SPARSE HASHED

Sparse hashed is similar to the hashed layout but uses sparse hashing, which in turn reduces the memory usage while increasing the computational requirements. The following is the syntax for specifying a sparse hashed layout:

```
LAYOUT(SPARSE_HASHSED())
```

SOURCE

The SOURCE parameter is used to define the data source used for creating the dictionary. The following types of sources are supported in ClickHouse:

FILE

This is used to load a dictionary from a local file in the disk. The syntax for this is:

```
SOURCE(FILE(path '/home/data/dict_data.csv' format 'CSV'))
```

The **`path`** parameter requires the absolute path of the file and the format requires the data format, which will be used for parsing the file.

HTTP/HTTPS

To create dictionary from http/https resource, this can be used. The syntax for loading dictionary from http/https file is given as follows:

```
SOURCE(HTTP(
    url 'http://127.0.0.1/sample/dict_data.csv'
    format 'CSV'
    credentials(user 'user_name' password 'password')
    headers(header(name 'API-Name' value 'key'))
))
```

The following parameters are required for creating a dictionary from http/https data source:

- **`url`**: This is the web URL for the data source.
- **`format`**: This is the file format to be parsed (for example, CSV and TabSeparated).
- **`credentials`**: These are the authentication details (user name and password to authenticate).
- **`headers`**: These are the optional HTTP headers in which the header name and values can be configured.

MySQL

ClickHouse allows loading dictionaries from MySQL tables. The syntax for loading dictionary from MySQL setup that we have been using in this book file is given as follows:

```
SOURCE(MYSQL(
    host 'localhost'
    port 3306
    user 'mysql_user'
    password 'clickhouse'
    db 'database_name'
    table 'table_name'
    where 'column_name=column_value'
))
```

PostgreSQL

ClickHouse can also load dictionaries from PostgreSQL. The syntax for loading dictionary from PostgreSQL is given as follows:

```
SOURCE(POSTGRESQL(
    port 5432
    host 'postgresql-hostname'
    user 'postgres_user'
    password 'postgres_password'
    db 'database_name'
    table 'table_name'
    where 'column_name= column_value''
    ))
```

ClickHouse

Similar to MySQL, it is possible to load a dictionary from another ClickHouse table. The syntax for loading a dictionary from a ClickHouse is as follows:

```
SOURCE(CLICKHOUSE(
    host 'localhost'
    port 9000
    user 'default'
    password ''
    db 'database_name'
    table 'table_name'
    where 'column1=7'))
```

MongoDB

Although MongoDB is a no-SQL database, it is still possible to load dictionaries from it. The syntax for loading a dictionary from a MongoDB is as follows:

```
SOURCE(MONGO(
    host 'localhost'
    port 27017
    user 'user_name'
    password 'password'
    db 'database_name'
    collection 'collection_dictionary'))
```

Redis

Redis is an in-memory key–value store database. ClickHouse allows us to create dictionaries from Redis. The syntax for loading a dictionary from a Redis is as follows:

```
SOURCE(REDIS(
    host 'localhost'
    port 6379
    storage_type 'simple'
    db_index 0))
```

LIFETIME

ClickHouse dictionaries can periodically update from the source based on the LIFETIME configuration provided in the CREATE statement. Based on the dictionary source, ClickHouse applies different methods of finding the updated source. For a text file, for example, the last modified time is used to find whether the source was updated. The following is the syntax for setting the lifetime:

```
LIFETIME(MIN 300 MAX 360)
```

Example 1 – MySQL

Let us now create a dictionary based on the employee table in MySQL. The following **`CREATE`** statements can be used to create a new database called **`dictionary_testing`** and a dictionary called **`mysql_dict`** in it:

```
CREATE DATABASE dictionary_testing;


CREATE DICTIONARY dict_testing.mysql_dict
(
    emp_no Int64,
    last_name String
)
PRIMARY KEY emp_no
SOURCE(MYSQL(
    replica(host '127.0.0.1' priority 1)
    port 3306
    user 'mysql_user'
```

```
    password 'clickhouse'
    db 'employees'
    table 'employees'
))
LAYOUT(FLAT())
LIFETIME(MIN 3600 MAX 72000);
```

Once the dictionary is successfully created, let us read the data from the created dictionary:

```
SELECT * FROM dict_testing.mysql_dict LIMIT 5;
```

SELECT * FROM dict_testing.mysql_dict LIMIT 5

Grid | Text

	emp_no	last_name
1	10001	Facello
2	10002	Simmel
3	10003	Bamford
4	10004	Koblick
5	10005	Maliniak

***Figure 10.1**: Sample data from the dictionary created from MySQL*

Example 2 – ClickHouse

Let us now create a dictionary based on the employee table in ClickHouse that was created in the earlier chapter. The following **CREATE** statements can be used to create a dictionary called employees in it:

```
CREATE DICTIONARY dictionary_testing.employees(
  emp_no UInt64,
  first_name String,
  last_name String,
  hire_date Date
)
PRIMARY KEY emp_no
SOURCE(CLICKHOUSE(HOST 'localhost' PORT 9000 USER 'default' TABLE
'employees_jdbc_mergetree' PASSWORD '' DB 'employees'))
LIFETIME(MIN 100 MAX 10000)
LAYOUT(HASHED());
```

The primary key (unique key) is the **emp_no** column and has three attributes (**first_name**, **last_name**, and **hire_date**). To retrieve attribute value for a given key, the **dictGet** function can be used:

```
SELECT dictGet('dictionary_testing.employees', 'first_name',
toUInt64(33003)) as first_name;
```

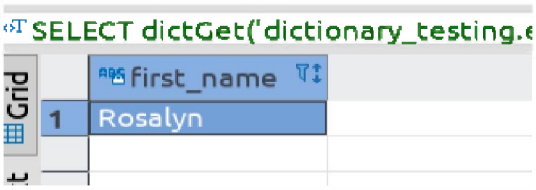


***Figure 10.2**: Retrieving the attributes using the dictGet function.*

Example 3 – PostgreSQL

Let us now create a dictionary based on the actor table (in the **dvdrental** database) in PostgreSQL. The following **CREATE** statements can be used to create a dictionary called **postgresql_dict**:

```
CREATE DICTIONARY dict_testing.postgresql_dict
(
    actor_id Int64,
    first_name String
)
PRIMARY KEY actor_id
SOURCE(POSTGRESQL(
    host '127.0.0.1'
    port 5432
    user 'postgres'
    password 'postgres123'
    db 'dvdrental'
    table 'actor'
))
LAYOUT(FLAT())
LIFETIME(MIN 3600 MAX 72000);
```

Once the dictionary is successfully created, let us read the data from the created dictionary:

```
SELECT * FROM dict_testing.postgresql_dict LIMIT 5;
```

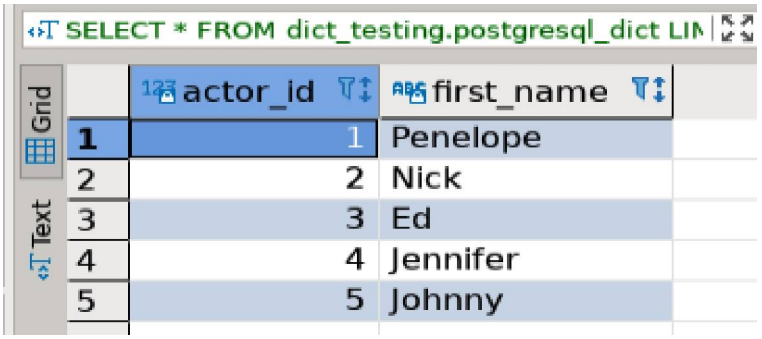


***Figure 10.3**: Sample data from the dict created from PostgreSQL.*

File

This engine allows ClickHouse to create and store the data in any of the supported file formats (CSV, TabSeparated, and so on) instead of regular columnar format. The syntax for creating a table in ClickHouse with file engine is as follows:

```
CREATE TABLE [database_name.]table_name
(
    column1 data_type1,
    column2 data_type2,
    column3 data_type3,
       ...
)
ENGINE=File(Format)
```

We shall create a table with three columns (**`Name`**, **`Age`**, and **`Weight`**) as an example to illustrate the File engine. The following **`CREATE`** statement can be used to create a database for testing out special engines and the table for File engines:

```
CREATE DATABASE special_engines;

CREATE TABLE special_engines.file_engine
(
    Name String,
    Age Int16,
```

```
    Weight Float32
)
ENGINE=File(CSV);
```

Once the table is created successfully, let us insert some test data in it using the following **INSERT** statement:

```
INSERT INTO special_engines.file_engine VALUES
('Jack', 31, 65.4),
('John', 41, 75.9),
('Jim', 54, 85.3);
```

After the data is inserted, we shall check the inserted data using the **SELECT** statement:

```
SELECT * FROM special_engines.file_engine;
```

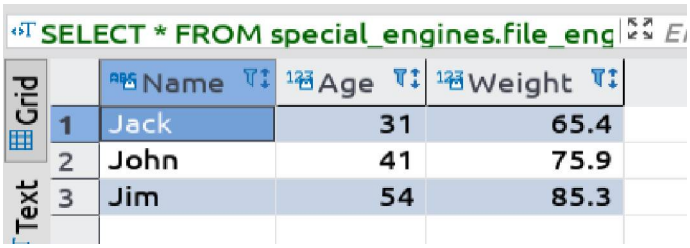


***Figure 10.4**: Data inserted in the table with File engine*

To view the file directly in the disk, the following Linux commands shall be executed sequentially:

```
$ sudo -s
$ nano /var/lib/clickhouse/data/special_engines/file_engine/data.csv
```

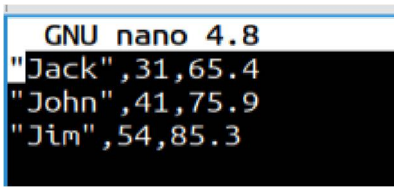


***Figure 10.5**: Data in the table with File engine, available as CSV file and viewed using nano editor*

Merge

Merge engine allows us to read simultaneously from multiple tables in ClickHouse. While the table engine enables parallel reading, storing or writing the data to the tables via the engine is not allowed. The Merge engine function accepts two

parameters. The first one is the database name and the second one is the regular expression, which will be used to merge the tables in the database.

For testing out the working of the Merge engine, we shall create two tables with identical columns and merge them. We shall begin by creating the first table meant for storing information about girls and insert some sample data in it as shown in the following syntax:

```
CREATE TABLE special_engines.merge_girls(
      ID Int32,
      Name String,
      Age Int16
)
ENGINE=MergeTree()
ORDER BY (ID, Name, Age);

INSERT INTO special_engines.merge_girls VALUES (1, 'Hannah', 24),
(2, 'Daphne', 22),
(3, 'Masha', 28);
```

We shall create another identical table meant for storing information about boys and insert some test data in it:

```
CREATE TABLE special_engines.merge_boys(
      ID Int32,
      Name String,
      Age Int16
)
ENGINE=MergeTree()
ORDER BY (ID, Name, Age);

INSERT INTO special_engines.merge_boys VALUES (4, 'Dexter', 27),
(5, 'Freddy', 21),
(6, 'Joe', 29);
```

Now, we shall proceed to create the Merge engine table, which will allow us to query the data from both tables simultaneously:

```
CREATE TABLE special_engines.merge_both as special_engines.merge_girls
```

```
ENGINE=Merge('special_engines', '^merge');
SELECT * FROM special_engines.merge_both;
```

SELECT * FROM special_engines.merge_

	ID	Name	Age
1	4	Dexter	27
2	5	Freddy	21
3	6	Joe	29
4	1	Hannah	24
5	2	Daphne	22
6	3	Masha	28

***Figure 10.6**: Data retrieved from the table with Merge engine*

The retrieved data is read from the tables that are merged and data is not stored in the Merge engine table. This engine can be useful for reading multiple small tables created over a period of time or to merge data from different tables with identical columns but with a different/modified configuration.

SET

Set table engine is a special type of engine that holds data in RAM and unlike other regular engines, it is not possible to retrieve the data using **SELECT** statements, but the data can be inserted in the table. The data can be read and retrieved from the table by using the table data on the right side of the **IN** operator.

The data in the set engine is stored in a disk along with RAM and loaded into the memory (RAM), every time the server is properly restarted. Let us test the working of the Set engine by creating a table with the Set engine, insert some test data, and read from the table using **IN** operator. The following statements can be used to create and insert test data to the Set engine:

```
CREATE TABLE special_engines.set_engine (
      ID UInt8)
Engine = Set;


INSERT INTO special_engines.set_engine VALUES
(1), (2), (3), (4), (5);
```

Now, let us create a test table with MergeTree engine and insert some test data.

```
CREATE TABLE special_engines.set_engine_source (
```

```
        ID UInt8,
        Dt Date DEFAULT toDate('2020-01-01')
)
Engine = MergeTree()
ORDER BY (ID, Dt);

INSERT INTO special_engines.set_engine_source (ID) VALUES (1),(2),(7);

SELECT * FROM special_engines.set_engine_source;
```

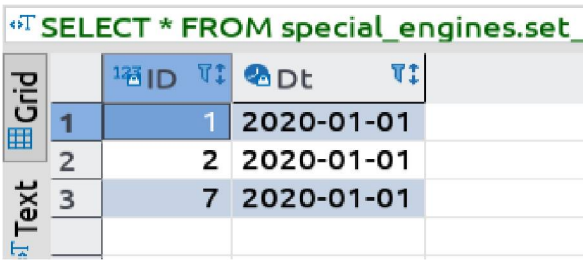


***Figure 10.7**: Data inserted into the table (MergeTree engine)*

Let us now read the table with the Set engine by using **IN** statement:

```
SELECT * FROM special_engines.set_engine_source
WHERE ID IN special_engines.set_engine;
```

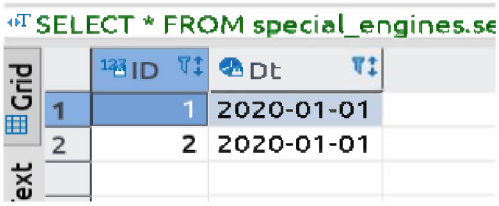


***Figure 10.8**: Query results - data read from Set engine table and used for filtering the data*

Memory

Memory engine, as the name suggests, stores the data in RAM. The main advantage of this engine is its speed since the data resides in RAM and supports parallel reading. Unlike the Set engine, which stores the data in disk along with RAM, the Memory engine does not store the data in the disk. So the contents are lost when the server is restarted. So the table contents are lost while the table would be still available.

Memory engine supports reading and writing the data directly from SQL statements. The following example demonstrates how to create a table using the Memory engine, write and read data to it:

```
CREATE TABLE special_engines.memory_engine (
      ID UInt8,
      Name String)
Engine = Memory;

INSERT INTO special_engines.memory_engine VALUES
(1,'Smith'),
(2, 'Heidi'),
(3, 'Victoria'),
(4, 'Andrew');

SELECT * FROM special_engines.memory_engine;
```

***Figure 10.9**: Data in Memory engine table*

Buffer

Buffer engine is used to write data into the RAM and the engine periodically flushes the data to another permanent table after the configured time period or row count is reached on the table. The syntax to create a table with a Buffer engine is given as follows:

```
CREATE TABLE [database_name.]buffer_table_name AS [database_name.]final_
table_name

Buffer(database, table_name, num_layers, min_time, max_time, min_rows,
max_rows, min_bytes, max_bytes)
```

The following are the parameters for the Buffer engine in ClickHouse:

- **db_name**: This is the name of the database in which the table (to which the data will be flushed) resides.
- **table_name**: This is the table to which the data will be flushed.
- **num_layers**: This is the number of independent buffers (determines the parallelism).
- **min_time** and **max_time**: This is the time for which the data resides in a buffer.
- **min_rows** and **max_rows**: This is the limit for the number of rows in a buffer.
- **min_bytes** and **max_bytes**: This is the limit for the size of data in buffer.

To test the Buffer engine, let us first create a MergeTree table to which the data will be written by the Buffer engine:

```
CREATE TABLE special_engines.buffer_engine_mergetree (
      ID UInt8,
      Name String)
Engine = MergeTree
ORDER BY (ID,Name);
```

Then, we shall create the Buffer engine table and insert some sample data into it. Once the data is inserted, we should be able to read the data from the Buffer engine table but not from the MergeTree table engine for the time period specified in the configuration. After that, we should be able to read from both tables. Consider the following example:

```
CREATE TABLE special_engines.buffer_engine AS special_engines.buffer_
engine_mergetree
ENGINE = Buffer(special_engines, buffer_engine_mergetree, 16, 10, 100,
2, 4, 10000000, 100000000);


INSERT INTO special_engines.buffer_engine VALUES
(1,'Smith'),
(2, 'Heidi'),
(3, 'Victoria'),
(4, 'Andrew');
```

Based on the configured time limit, the data will be flushed to the MergeTree table. Both the queries should return the same result since querying the Buffer engine table will result in data being read from both the buffer and the table to which the data is inserted:

```
SELECT * FROM special_engines.buffer_engine_mergetree;
SELECT * FROM special_engines.buffer_engine;
```

***Figure 10.10**: Data from the Buffer engine table*

URL

The URL engine can be used to read/write data from a remote http/https server. The engine is similar to the File engine, but the data resides in a remote server.

The **`SELECT`** and **`INSERT`** statements are translated to **`GET`** and **`POST`** requests to the server, respectively. The engine requires two parameters. The first one is the URL and the next one is the data format in the URL. The syntax for creating the table with URL engine is given as follows:

```
CREATE TABLE [database_name].table_name
(
    column1 data_type1,
    column2 data_type2,
    column3 data_type3,
       ...
)
ENGINE=URL(url, DataFormat)
```

Example

Let us create a sample table from a remote URL. We shall use the car evaluation dataset available in the UCI Machine Learning repository. The dataset has seven attributes pertaining to car and is available as an open dataset. The SQL statement for creating the table using the URL engine is given as follows:

```
CREATE TABLE special_engines.url_table
(Buying Nullable(String),
Maintenance Nullable(String),
Doors Nullable(UInt64),
Persons Nullable(UInt64),
Luggage_Boot Nullable(String),
safety Nullable(String),
Class_Value Nullable(String))
ENGINE=URL('https://archive.ics.uci.edu/ml/machine-learning-databases/
car/car.data', CSV);
```

Once the table is created, let us run the following query and see the data in the table:

```
SELECT * FROM special_engines.url_table LIMIT 10;
```

SELECT * FROM special_engines.url_table LIMIT | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

	Buying	Maintenance	Doors	Persons	Luggage_Boot	safety	Class_Value
1	vhigh	vhigh	2	2	small	low	unacc
2	vhigh	vhigh	2	2	small	med	unacc
3	vhigh	vhigh	2	2	small	high	unacc
4	vhigh	vhigh	2	2	med	low	unacc
5	vhigh	vhigh	2	2	med	med	unacc
6	vhigh	vhigh	2	2	med	high	unacc
7	vhigh	vhigh	2	2	big	low	unacc
8	vhigh	vhigh	2	2	big	med	unacc
9	vhigh	vhigh	2	2	big	high	unacc
10	vhigh	vhigh	2	4	small	low	unacc

***Figure 10.11**: Data from the URL engine table*

Distributed

ClickHouse supports splitting the tables and storing the shards in multiple servers in the cluster. **Distribution** refers to splitting the data into multiple parts, storing those parts in different servers; and **replication** is storing the copies of data in more than one server.

The sample configuration for data distribution is already covered in *Chapter 7*. The steps to create a distributed table are:

1. Create the table on each of the servers where the shards are configured to be stored.
2. Create the distributed table.

As an example, let us create the following table on each of the servers:

```
CREATE TABLE distribute_test
(
 name String,
 age Int16,
 weight Float32
)
Engine=MergeTree()
ORDER BY name;
```

Once the tables are created in each of the servers in the ClickHouse cluster, the distributed table needs to be created in the servers on which the data will be queried as shown in the following statements:

```
CREATE TABLE distribute_table_test
(
 name String,
 age Int16,
 weight Float32
)
ENGINE = Distributed(simple_shard_replicas, default, distribute_test);
```

The first parameter is the cluster name configured in config.xml, the second one is the schema name, and the third is the name of the table that is distributed. The distributed table is just like a view of the **`distribute_test`** table. When we query the **`distribute_table_test`**, the data is processed on the servers where the shards reside and the final result is consolidated in the server from which the query was executed.

Conclusion

We have seen all the commonly used table engines in ClickHouse including MergeTree family, Log family, and Special family of engines. We have also explored engines used to integrate ClickHouse with external data sources. These engines are useful in multiple scenarios and require to be chosen based on the problem at hand.

In the upcoming chapters, we shall cover the advanced/administrative operations such as configuring ClickHouse, creating and managing users and user roles, and on system tables.

Points to remember

- ClickHouse has multiple engines for special usage scenarios.
- Dictionary engine is a key–attribute mapping and is commonly used for JOIN operations.
- File engine stores the data in specified file format instead of regular columnar format.
- Merge engine can be used to merge multiple tables of similar structure and data can be read parallel from them.
- Set engine stores data in RAM and can be read using IN operators
- The Memory engine stores the data in RAM and is volatile.
- Buffer engine stores the data temporarily in RAM and flushes to another permanent table.
- URL engine is similar to File engine, but the data is managed and stores in a remote server.

Multiple choice questions

1. **Dictionaries in ClickHouse are always stored completely in RAM.**
   a) True
   b) False

2. **The most optimal layout for storing dictionaries is _____.**
   a) Cache
   b) Hashed
   c) Flat
   d) None of the options

3. **Storing or writing the data to the tables via the Merge engine is not allowed.**
   a) True
   b) False

4. **In the SET engine, the data can be inserted in the table and read using SELECT statements.**
   a) True
   b) False

5. **The SET engine stores the data in _____.**
   a) Disk
   b) SSD and RAM
   c) RAM
   d) Disk and RAM

6. **The MEMORY engine stores the data in _____.**
   a) Disk
   b) SSD and RAM
   c) RAM
   d) Disk and RAM

7. **Querying a Buffer engine table will result in data being read from both the Buffer engine table and the table to which the data is inserted.**
   a) True
   b) False

8. **The URL engine can be used to read/write data from a remote http/https server.**
   a) True
   b) False

Answers

1. **b**
2. **c**
3. **a**
4. **b**
5. **d**
6. **c**
7. **a**
8. **a**

CHAPTER 11

Configuring the ClickHouse Setup – Part 1

In the preceding chapters, we have learned how to perform basic actions in ClickHouse like creating tables, querying the data from the tables, multiple engines available, and populating data from other data sources. In this section, we shall cover advanced operations like configuring the ClickHouse setup, creating users and user profiles, and also learn about important system tables. Note that in the production environment, configuring or changing the default configurations are for advanced users/database admins and wrong/invalid configurations could result in unusable/instability in the setup.

Most of the contents in the next two chapters are based on official documentation and the source code. These chapters can be used as offline/alternate quick references for ClickHouse settings.

Structure

In this chapter, we will discuss the various server settings available for the end users to configure:

- Network-related settings
- SSL settings
- Internal settings

- Table engine settings

Objectives

The main objectives of this chapter are to:

- Know the different configuration settings in ClickHouse.
- Understand the various available settings to optimize/tune ClickHouse, based on user requirements.

Introduction

All of the ClickHouse server and user configurations are available in XML file format. The default server configuration file is located in the path **`/etc/clickhouse-server/config.xml`** and the user configurations in **`/etc/clickhouse-server/users.xml`**. The individual settings are represented as XML elements and the setting values are configured as the tag content. An example for logger configuration is given as follows:

```
<logger>
<level>trace</level>
<log>/var/log/clickhouse-server/clickhouse-server.log</log>
<errorlog>/var/log/clickhouse-server/clickhouse-server.err.log</
errorlog>
<size>1000M</size>
<count>10</count>
</logger>
```

The logger settings are defined in the XML section, which includes logging level, the path of the log file/error log file, size of the file, and so on. The contents of these tags can be modified in case the end user wishes to change the default configuration. The server configuration settings (configs in config.xml) cannot be changed at session level or via using SET statements in queries and will be covered in this chapter.

Network settings

ClickHouse allows users to configure settings related to network, ports, and timeout settings. The firewall settings are still managed at the server level.

HTTP/HTTPS port

This setting defines the http/https ports through which the ClickHouse server can be connected via a third-party application. SSL settings must be configured if https

port is configured. If http port is configured, the SSL settings are ignored. By default, only http port is configured and the default value is 8123 (we had earlier specified this while setting up DBeaver to connect to ClickHouse).

```
<http_port>8123</http_port>
```

```
<https_port>8443</https_port>
```

HTTP server's default response

By default, a web page with "Ok" text is shown when ClickHouse's http server is accessed. This can be tested by visiting http://127.0.0.1:8123/ via any browser from the server:

```
<http_server_default_response>
```

```
<![CDATA[<html><head><title>ClickHouse HTTP</title></head><body><p>My
Custom Message.</p></body></html>]]>
```

```
</http_server_default_response>
```

Inter-server host and port

Hostname and port can be specified for the current server using which other servers can access ClickHouse in the current server via the specified port.

```
<interserver_http_host>custom_hostname</interserver_http_host>
```

```
<interserver_http_port>9009</interserver_http_port>
```

Inter-server credentials

Credentials are used to authenticate the server during the data replication (used by replicated table engines). By default, ClickHouse replication is authenticated without any password, and enabling the password can act as an extra layer of security while ClickHouse performs data replication.

The credentials must be same across the ClickHouse servers used for replication. These credentials are not to be confused with ClickHouse user credentials, which are different:

```
<interserver_http_credentials>
```

```
<user>user_name</user>
```

```
<password>password_for_user_name</password>
```

```
</interserver_http_credentials>
```

TCP/TCP secure port

This is the port used for communicating with ClickHouse server via TCP protocol. If secure TCP port is specified, SSL should also be configured.

```
<tcp_port>9000</tcp_port>
<tcp_port_secure>9440</tcp_port_secure>
```

Request timeout

For the incoming requests, ClickHouse waits for up to 3 seconds by default after which the connection is closed automatically.

```
<keep_alive_timeout>3</keep_alive_timeout>
```

Allowing inbound requests

Users can specify the list of hosts from which the requests are allowed. Setting the value to `::1` will allow ClickHouse to respond to the requests from any server.

```
<listen_host>::1</listen_host>
```

or

```
<listen_host>127.0.0.1</listen_host>
```

Maximum inbound connections

This setting can be used to limit the maximum number of inbound connections to the ClickHouse server. The default value is `4096`.

```
<max_connections>4096</max_connections>
```

SSL settings

SSL enables secure communication between ClickHouse servers. To enable encryption, the `tcp_port_secure` and `https_port` need to be enabled in the configuration. The following is a sample for SSL settings:

```
<openSSL>

<server>
<certificateFile>/etc/clickhouse-server/server.crt</certificateFile>
<privateKeyFile>/etc/clickhouse-server/server.key</privateKeyFile>
<dhParamsFile>/etc/clickhouse-server/dhparam.pem</dhParamsFile>
<verificationMode>none</verificationMode>
<loadDefaultCAFile>true</loadDefaultCAFile>
<cacheSessions>true</cacheSessions>
<disableProtocols>sslv2,sslv3</disableProtocols>
<preferServerCiphers>true</preferServerCiphers>
</server>
```

```
<client>
<loadDefaultCAFile>true</loadDefaultCAFile>
<cacheSessions>true</cacheSessions>
<disableProtocols>sslv2,sslv3</disableProtocols>
<preferServerCiphers>true</preferServerCiphers>
<invalidCertificateHandler>
<name>RejectCertificateHandler</name>
</invalidCertificateHandler>
</client>

</openSSL>
```

SSL server configurations

To enable secure communication via SSL, the following files are required:

- **`server.crt`** (server X509 certificate) and **`server.key`** (private key)
- **`dhparam.pem`** (Diffie–Hellman parameters)

We can create our own certificate authority and use the root certificate to sign our certificates or use a self-signed certificate. It is also possible to use a certificate signed by a recognized certificate authority, but this can be slightly expensive and time-consuming. The Diffie–Hellman parameters can also be generated manually, and it is an optional parameter. The following are the supported settings for SSL:

- **`caConfig`**: This is the file path or directory where the trusted root certificates are stored.
- **`cacheSessions`**: This enables or disables caching sessions.
- **`certificateFile`**: This is the file path to the client/server certificate in PEM format.
- **`cipherList`**: This supports OpenSSL encryptions.
- **`disableProtocols`**: This is used to disable SSL protocols that are not allowed to use (for example, sslv2 and sslv3).
- **`extendedVerification`**: This means after the session ends, it automatically extends the verification of SSL certificates.
- **`fips`**: This enables OpenSSL **FIPS (Federal Information Processing Standard)** compliant mode.

- **invalidCertificateHandler**: This class is for verifying invalid certificates.

```
<invalidCertificateHandler> <name>ConsoleCertificateHandler</name>
</invalidCertificateHandler>
```

- **loadDefaultCAFile**: When this is enabled, the built-in CA certificates for OpenSSL will be used. Allowed values could be either true or false.
- **preferServerCiphers**: This parameter is for the preferred server ciphers on the client.
- **privateKeyFile**: This is the file path to the secret key of the PEM certificate.
- **privateKeyPassphraseHandler**: This is the class that requests the passphrase for accessing the private key.

```
<privateKeyPassphraseHandler>
<name>KeyFileHandler</name>
<options><password>test</password></options>
</privateKeyPassphraseHandler>
```

- **requireTLSv1**: Set whether a TLSv1 connection is required. It can be either true or false.
- **requireTLSv1_1**: Set whether a TLSv1.1 connection is required. It can be either true or false.
- **requireTLSv1**: Set whether a TLSv1.2 connection is required. It can be either true or false.
- **sessionCacheSize**: This parameter is for the maximum number of sessions that the server caches. The default value is **1024*20**.
- **sessionIdContext**: This is a unique set of random characters that the server appends to each generated identifier. The length of the string must not exceed the value set in the **SSL_MAX_SSL_SESSION_ID_LENGTH**.
- **sessionTimeout**: This is the time limit for caching a session on the server.
- **verificationDepth**: This is the upper limit for the length of the verification chain. If the certificate chain length exceeds the limit, the verification will not succeed.
- **verificationMode**: This is the verification method for checking the node's certificates. Allowed values are none, once, relaxed, or strict.

Internal server settings

This section contains the relevant internal configuration settings to tweak the ClickHouse application. For example, to configure the macros, logging, and default profiles.

Maximum concurrent queries

This setting will allow the user to set the maximum number of queries that can be run in the server in parallel. The default value is **100**.

```
<max_concurrent_queries>100</max_concurrent_queries>
```

```
< max_concurrent_queries_for_all_users> 80 </max_concurrent_queries_for_
all_users>
```

Maximum table and partition size to drop

This is the maximum size of a table or partition (in bytes) from MergeTree engine that can be dropped using **DROP** SQL statement. The default value is 50 GB and setting it to 0 would allow dropping tables/partitions of any size:

```
<max_table_size_to_drop>1024000000</max_table_size_to_drop>
```

```
<max_partition_size_to_drop>1024000000</max_partition_size_to_drop>
```

Default database

The default database is chosen when the database is not specified in the SQL queries.

```
<default_database>default</default_database>
```

Default profile

Default user settings profile that is read from the config loaded from **users_profile** configuration.

```
<default_profile>default</default_profile>
```

Format schema path

This setting allows the user to specify the path containing the schema of the input formats supported by ClickHouse.

```
<format_schema_path>/var/lib/clickhouse/format_schemas/</format_schema_
path>
```

Logger

Log files are stored in the path **/var/logs/clickhouse-server**. ClickHouse supports multiple logging level settings. The following settings and values are allowed:

- `level`: This is the logging level. It can take any of the values in trace, debug, information, warning, or error.
- `log`: This is the log file path where the logging information is stored.
- `errorlog`: This is the file path of the error log.
- `size`: This is the maximum allowed size of the log file. When this limit is reached, ClickHouse automatically archives and renames the old log file and creates a new log file.
- `count`: This is the maximum number of archived log files allowed.

```
<logger>
<level>trace</level>
<log>/var/log/clickhouse-server/clickhouse-server.log</log>
<errorlog>/var/log/clickhouse-server/clickhouse-server.err.log</
errorlog>
<size>1000M</size>
<count>10</count>
</logger>
```

ClickHouse also supports logging in syslog. The following settings template can be used to enable logging to syslog:

```
<logger>
<use_syslog>1</use_syslog>
<syslog>
<address>syslog_address:10514</address>
<hostname>myhost.local</hostname>
<facility>LOG_LOCAL6</facility>
<format>syslog</format>
</syslog>
</logger>
```

Macros

Macros are commonly used in `ReplicatedMergeTree` family engines. Parameter substitutions are configured that are used in creating replicated tables.

```
<macros>
<layer>01</layer>
<shard>02</shard>
```

```
<replica>replica1</replica>
</macros>
```

Path

This is the path in the server where ClickHouse stores all the data. The user can provide different paths if the default path doesn't have sufficient space or need not be used.

```
<path>/var/lib/clickhouse/</path>
```

Mark cache size

This is the size of the cache (in bytes) of marks used by MergeTree family engines. The minimum recommended value is **5368709120**.

```
<mark_cache_size>5368709120</mark_cache_size>
```

Logging MergeTree events

This setting must be enabled to log all the information on any events in MergeTree tables (creation, deletion, merges, downloads).

```
<part_log>
    <database>system</database>
    <table>part_log</table>
    <partition_by>toMonday(event_date)</partition_by>
    <flush_interval_milliseconds>7500</flush_interval_milliseconds>
</part_log>
```

Logging queries in system table

This setting enables ClickHouse to store the queries executed in a separate table. By default, the queries are logged in the **system.query_log** table. The following setting will log the queries in the **query_log** table in the system database. The queries are stored for the specified number of days. The **log_query** setting must be set via query or configured in the profile of the users for this to work:

```
<query_log>
<database>system</database>
<table>query_log</table>
<engine>Engine = MergeTree PARTITION BY event_date ORDER BY event_time
TTL event_date + INTERVAL 30 day</engine>
<flush_interval_milliseconds>7500</flush_interval_milliseconds>
</query_log>
```

Logging the threads of received queries

This enables logging of query thread. By default, the queries are logged in the **system.query_thread_log** table. The **log_query_threads** setting must be set via query or configured in the profile of the users for this to work:

```
<query_thread_log>
    <database>system</database>
    <table>query_thread_log</table>
    <partition_by>toMonday(event_date)</partition_by>
    <flush_interval_milliseconds>7500</flush_interval_milliseconds>
</query_thread_log>
```

Text log

This setting is used if the logs are required to be written in a table (**system.text_log**), instead of separate log files:

```
<text_log>
      <database>system</database>
      <table>text_log</table>
      <flush_interval_milliseconds>7500</flush_interval_milliseconds>
      <level></level>
</text_log>
```

Query profiler trace logs

If query profiler is enabled, and if we wish to store the stack traces in a table (**system.trace_log**), **trace_log** setting can be used:

```
<trace_log>
      <database>system</database>
      <table>trace_log</table>
      <partition_by>toYYYYMM(event_date)</partition_by>
      <flush_interval_milliseconds>7500</flush_interval_milliseconds>
</trace_log>
```

Data compression

Data compression settings are available for the MergeTree table engine. Multiple cases can be set and the cases are selected by ClickHouse if the data parts match the case condition:

```
<compression>
```

```
    <case>
      <min_part_size>102400000</min_part_size>
      <min_part_size_ratio>0.01</min_part_size_ratio>
      <method>lz4</method>
    </case>
</compression>
```

Masking the queries before logging

This setting is used to mask sensitive information (using Regex) that may be present in the query, before logging the query in a file or the system table. The patterns matching the Regex will be replaced by the string specified by the user in the configuration:

```
<query_masking_rules>
    <rule>
        <name>hide SSN</name>
        <regexp>(^|\D)\d{3}-\d{2}-\d{4}($|\D)</regexp>
        <replace>000-00-0000</replace>
    </rule>
</query_masking_rules>
```

TimeZone

This can be used to manually set the time zone of the ClickHouse server.

```
<timezone>Asia/Kolkata</timezone>
```

Temporary path

This setting sets the temporary folder path used for processing larger queries:

```
<tmp_path>/var/lib/clickhouse/tmp/</tmp_path>
```

Users configuration

This is the file path for the user configuration file:

```
<users_config>users.xml</users_config>
```

MySQL port

A specified port number will be used to connect to MySQL server:

```
<mysql_port>9004</mysql_port>
```

Zookeeper settings

Zookeeper is used in replicated table engines. An example for zookeeper settings is given as follows:

```
<zookeeper>
    <node>
        <host>hostname1</host>
        <port>2181</port>
    </node>
    <node>
        <host>hostname2</host>
        <port>2181</port>
    </node>
    <session_timeout_ms>30000</session_timeout_ms>
    <root>/path/to/zookeeper/node</root>
    <identity>user_name:password</identity>
</zookeeper>
```

- `node`: This is the Zookeeper host name.
- `port`: This is the Zookeeper port.
- `session_timeout`: This is the timeout for the client session (milliseconds).
- `root`: This is the root node in zookeeper.
- `identity`: This is the username and password to connect.

Table engine settings

These settings are used to change the default settings of table engines available in ClickHouse. We have separate settings for Dictionary as well as the MergeTree table engines.

Dictionary settings

The following settings are available for changing the configuration of dictionaries:

Built-in dictionaries reload interval

This determines the time interval between successive refreshes of the in-built dictionaries in ClickHouse. The default value is `3600` seconds.

```
<builtin_dictionaries_reload_interval>3600</builtin_dictionaries_reload_
interval>
```

External dictionary configuration

This is used to set the file path for the configuration of external dictionaries.

```
<dictionaries_config>*_dictionary.xml</dictionaries_config>
```

Lazy load

Lazy load determines whether the application loads dictionary on first use (True) or while loading the server (False). The default setting is True.

```
<dictionaries_lazy_load>true</dictionaries_lazy_load>
```

MergeTree settings

MergeTree engine, being the most robust, is feature rich and the most widely used table engine. It has the most number of user-configurable settings. The following is an of the setting:

```
<merge_tree>
    <max_suspicious_broken_parts>5</max_suspicious_broken_parts>
</merge_tree>
```

The following settings listed in the table are available for the MergeTree engine. More settings are available in **`/src/Storages/MergeTree/MergeTreeSettings.h`** file in the source code:

Data settings		
Setting	**Default value**	**Explanation**
`index_granularity`	`8192`	Number of rows that correspond to marks of an index
`min_bytes_for_wide_part`	`10485760`	Minimum size in bytes to create the part in wide format.
`min_rows_for_wide_part`	`0`	The minimum number of rows to create the part in wide format.

***Table 11.1**: Data settings for the MergeTree table engine*

The following configuration settings are available for insert operations in the MergeTree engine table:

Insert settings		
Setting	**Default value**	**Explanation**
`parts_to_delay_insert`	`150`	Insert operation is slowed down by the application if there are more than or equal to the specified number of active parts in a single partition
`parts_to_throw_insert`	`300`	Insert operation throws an exception if there are more than or equal to the specified number of active parts in a single partition
`max_delay_to_insert`	`1`	Insert operations are delayed if there are a lot of unmerged parts in a single partition
`max_parts_in_total`	`100000`	Insert operation throws an exception if there are more than or equal to the specified number of active parts in all the partitions

Table 11.2: Insert settings for the MergeTree table engine

The following configuration settings are available for the merge operations in a MergeTree table engine:

Merge settings		
Setting	**Default value**	**Explanation**
`merge_max_block_size`	`DEFAULT_MERGE_BLOCK_SIZE`	Number of rows in blocks required to be formed for merge operations
`max_bytes_to_merge_at_max_space_in_pool`	`150ULL * 1024 * 1024 * 1024`	Maximum size of parts to merge when there are maximum free threads in background pool
`max_bytes_to_merge_at_min_space_in_pool`	`1024 * 1024`	Maximum size of parts to merge when there are minimum free threads in background pool
`max_replicated_merges_in_queue`	`16`	The upper limit for the number of merging and mutating parts in the ReplicatedMergeTree queue

`max_replicated_mutations_in_queue`	`8`	The upper limit for the number of simultaneous mutating parts in the ReplicatedMergeTree queue
`number_of_free_entries_in_pool_to_lower_max_size_of_merge`	`8`	When there are less than the specified number of free entries in the pool/replicated queue, start to lower the maximum size of the merge to process. This enables small merges to process thereby not filling the pool with long-running merges
`number_of_free_entries_in_pool_to_execute_mutation`	`10`	When there are less than the specified number of free entries in a pool, part mutations are not executed. This is to leave free threads for regular merges
`old_parts_lifetime`	`8 * 60`	The lifetime of obsolete parts in seconds
`temporary_directories_lifetime`	`86400`	The lifetime of temporary directories in seconds
`disable_background_merges`	`false`	Disallow background merges

***Table 11.3**: Merge settings for the MergeTree table engine*

The following configuration settings are available in ClickHouse for replication settings in the `ReplicatedMergeTree` family engine tables:

Replication settings		
Setting	**Default value**	**Explanation**
`replicated_deduplication_window`	`100`	Recent blocks of hashes are be kept in ZooKeeper.
`replicated_deduplication_window_seconds`	`7 * 24 * 60 * 60 (1 Week)`	The lifetime of recent blocks of hashes to be kept in ZooKeeper.
`max_replicated_logs_to_keep`	`10000`	The maximum number of log entries about inactive replicas.
`min_replicated_logs_to_keep`	`10`	The maximum number of log entries about inactive replicas, even if obsolete.

prefer_fetch_merged_part_time_threshold	3600	If the time elapsed after replication log entry creation exceeds this threshold and the total size of parts is greater than prefer_fetch_merged_part_size_threshold, prefer fetching merged part from replica instead of performing merge locally.
prefer_fetch_merged_part_size_threshold	10ULL * 1024 * 1024 * 1024	If the size of parts exceeds this threshold and the total size of parts is greater than prefer_fetch_merged_part_time_threshold, prefer fetching merged part from replica instead of performing merge locally.
max_suspicious_broken_parts	10	Maximum suspicious broken parts allowed, beyond which automatic deletion is not possible.
max_files_to_modify_in_alter_columns	75	The upper limit for the number of files that can be modified while performing ALTER operations.
max_files_to_remove_in_alter_columns	50	The upper limit for the number of files that can be deleted while performing ALTER operations.
replicated_max_ratio_of_wrong_parts	0.5	Maximum allowed ratio of the number of broken parts to the total number of parts.
replicated_max_parallel_fetches	0	The upper limit for parallel data fetches.
replicated_max_parallel_fetches_for_table	0	The upper limit for parallel data fetches per table.
replicated_max_parallel_fetches_for_host	DEFAULT_COUNT_OF_HTTP_CONNECTIONS_PER_ENDPOINT	Limit parallel fetches from an endpoint.
replicated_max_parallel_sends	0	The upper limit for parallel sends.
replicated_max_parallel_sends_for_table	0	The upper limit for parallel sends per table.
replicated_can_become_leader	True	Replicas of replicated tables can become the leader.

`zookeeper_session_expiration_check_period`	`60`	Time period to check Zookeeper session expiration.

Table 11.4: Replication settings for the MergeTree table engine

The following configuration settings are available for checking the replication delay in a `ReplicatedMergeTree` family table:

Check delay of replicas		
Setting	**Default value**	**Explanation**
`min_relative_delay_to_measure`	`120`	Calculate relative replica delay only if the absolute delay is not less than that of this value.
`cleanup_delay_period`	`30`	Time period for performing the clean-up activity on old queue logs, blocks hashes, and parts.
`cleanup_delay_period_random_add`	`10`	Random time to add with `cleanup_delay_period` to avoid thundering herd effect and subsequent DoS of ZooKeeper.
`min_relative_delay_to_close`	`300`	The minimum time delay from other replicas before closing, stop serving requests, and not return OK during the status check.
`min_absolute_delay_to_close`	`0`	The minimum time delay before closing, stop serving requests, and not return OK during the status check.
`enable_vertical_merge_algorithm`	`1`	Enables usage of vertical merge algorithm.
`vertical_merge_algorithm_min_rows_to_activate`	`16 * DEFAULT_MERGE_BLOCK_SIZE`	The minimal number of rows in merging parts to activate the vertical merge algorithm.
`vertical_merge_algorithm_min_columns_to_activate`	`11`	The minimal amount of non-PK columns to activate the vertical merge algorithm.

Table 11.5: Settings for check delays of replicas in the MergeTree table engine

The following configuration settings are available for compatibility in a MergeTree table engine:

Compatibility settings		
Setting	**Default value**	**Explanation**
`compatibility_allow_sampling_expression_not_in_primary_key`	`False`	Flag to allow table creation with sampling expression, not in the primary key.
`use_minimalistic_checksums_in_zookeeper`	`True`	Use the format that results in a smaller size for part checksums in ZooKeeper.
`use_minimalistic_part_header_in_zookeeper`	`True`	Store part header, which contains checksums and columns in a compact format. A single part of znode is created instead of separate znodes.
`finished_mutations_to_keep`	`100`	Record count about mutations that are done.
`min_merge_bytes_to_use_direct_io`	`10ULL * 1024 * 1024 * 1024`	The minimum number of bytes to enable `O_DIRECT` in merge operations.
`index_granularity_bytes`	`10 * 1024 * 1024`	Approximate number of bytes in a granule.
`merge_with_ttl_timeout`	`3600 * 24`	Timeout after which merge with TTL can be repeated.
`ttl_only_drop_parts`	`False`	Drop the expired parts completely and not prune them partially.
`write_final_mark`	`1`	Write the final mark after the end of the column.
`enable_mixed_granularity_parts`	`0`	Enable parts with adaptive and non-adaptive granularity.
`max_part_loading_threads`	`0`	The number of threads to load data parts at start-up.
`max_part_removal_threads`	`0`	The number of threads for concurrent removal of inactive data parts.
`concurrent_part_removal_threshold`	`100`	Activate concurrent part removal after the threshold is crossed.
`storage_policy`	`"default"`	Name of the storage disk policy.
`allow_nullable_key`	`False`	If enabled, Nullable data types are allowed in primary keys.

`allow_s3_zero_copy_replication`	`False`	Enable or disable zero-copy replication in S3 storage.
`remove_empty_parts`	`True`	Enable or disable deleting the empty parts as a result of the TTL setting.
`max_partitions_to_read`	`-1`	The maximum number of partitions that are allowed to be read in a single query. Negative values indicate an unlimited number of partitions that can be read.
`max_concurrent_queries`	`0`	The maximum number of queries that can be executed in a MergeTree table engine.

***Table 11.6**: Compatibility settings in the MergeTree table engine*

Example

Let's say that we have to configure the following for our ClickHouse setting:

```
index_granularity -1024 rows
max_parts_in_total – 8192 parts while inserting the data
old_parts_lifetime – 60 seconds
max_replicated_logs_to_keep – 100 logs
max_suspicious_broken_parts – 1000 parts
zookeeper_session_expiration_check_period – 120 seconds
cleanup_delay_period – 60 seconds
finished_mutations_to_keep – 1000 mutations
max_partitions_to_read – 128 partitions
```

The following section has to be updated in the config.xml file:

```
<merge_tree>
<index_granularity>1024</index_granularity>
<max_parts_in_total>8192</max_parts_in_total>
      <old_parts_lifetime>60</old_parts_lifetime>
      <max_replicated_logs_to_keep>100</max_replicated_logs_to_keep>
      <max_suspicious_broken_parts>1000</max_suspicious_broken_parts>
      <zookeeper_session_expiration_check_period>120</zookeeper_
session_expiration_check_period>
```

```
        <cleanup_delay_period>60</cleanup_delay_period>
        <finished_mutations_to_keep>1000</finished_mutations_to_keep>
        <max_partitions_to_read>128</max_partitions_to_read>
</merge_tree>
```

Conclusion

In this chapter, we have seen the configuration settings that are available and valid in `config.xml`. Other configuration settings that are available can be found in the `users.xml` file, which will be covered in the next chapter along with user creation and role-based access control.

Points to remember

- ClickHouse configurations are available in config.xml and `users.xml` files.
- Network-related settings, SSL settings, application settings, and table engine settings are available in the `config.xml` file.
- Settings related to queries, profiles, and user settings are available in the `users.xml` file.

Multiple choice questions

1. **Changing the default configuration of the ClickHouse server is recommended for novice users.**
   a) True
   b) False

2. **ClickHouse configurations are stored in _____ format.**
   a) YAML
   b) JSON
   c) HTML
   d) XML

3. **The server configuration settings cannot be changed at the session level or via using SET statements in queries.**
   a) True
   b) False

4. **ClickHouse supports logging in syslog.**
   a) True
   b) False
5. **The default data directory can be specified in config.xml.**
   a) True
   b) False

Answers

1. **b**
2. **d**
3. **a**
4. **a**
5. **a**

Chapter 12

Configuring the ClickHouse Setup – Part 2

ClickHouse server settings and settings related to the table engines were covered in the previous chapter. In this chapter, we shall take a look at the settings related to the queries that are executed, settings profile, user settings, and system tables available in the ClickHouse setup.

Structure

In this chapter, we will learn about the various settings available for the end users to configure:

- Query permissions
- Query complexity
- Settings profile
- Quota
- User settings
- Role-based access control
- System tables

Objectives

The main objectives of this chapter are to:

- Know the different configuration settings in ClickHouse
- Configuring the ClickHouse based on the requirement
- Learn role-based access control
- Know about the available system tables and their usage

Introduction

In this chapter, we will take a look at the settings available in the users.xml file. It is also possible to update the settings from the console client (using the SET statement) for each session. The settings in the users.xml will also be covered in detail.

Similar to the server settings, the settings discussed here are stored in the xml format (**`users.xml`**). The query settings are stored under profiles and each user is assigned a profile and quota.

A sample xml structure is as follows:

```
<yandex>
<profiles>
      <profile_name>
            …
      </profile_name >
</profiles>
<quotas>
      <quota_name>
            …
      </quota_name>
</quotas>
<users>
      <user_name>
<password>abc123</password>
            …
<profile>profile_name</profile>
<quota>quota_name</quota>
```

```
</users>
</yandex>
```

Query permissions

Query permission settings are used to restrict/allow users to execute a certain set of SQL queries. Separate settings are available to enable read-only (**SELECT**, **DESCRIBE**, **EXISTS**, and **SHOW**) or allow write queries (**INSERT** and **OPTIMIZE**), or to allow DDL queries (**CREATE**, **ALTER**, **RENAME**, **ATTACH**, **DETACH**, **DROP**, and **TRUNCATE**).

Read-only

The read-only setting, when enabled, will allow the users to execute only the read queries like **SELECT** and **SHOW**:

```
<readonly>1</readonly>
```

The allowed values are **1** (read-only), **0** (allow all queries), and **2** (read-only, but **SET** and **USE** statements are allowed to modify the settings). The default value is **0**.

Allow DDL

Similar to read-only, this setting allows or denies DDL queries:

```
<allow_ddl>1</allow_ddl>
```

Query complexity

Limiting the complexity of the query helps in the safer execution of the query without exhausting the hardware resources.

The settings starting with **max_** accept numeric values and **0** means unlimited or unrestricted for that setting. The settings with overflow mode can accept throw (throws an exception when the limit is exceeded) or break (breaks the operation when the limit is reached and returns the partial result).

ClickHouse checks the query complexity restriction settings on data parts, not on rows. These restrictions predominantly impact read operations (**SELECT** query). For multi-server setup, the settings are required to be updated in each of the servers.

Query complexity settings – memory		
Setting	**Default value**	**Explanation**
`max_memory_usage`	`0`	Maximum memory allowed for single query execution.
`max_memory_usage_for_user`	`0`	Maximum memory allowed for a user in a single server to execute queries.
`max_memory_usage_for_all_queries`	`0`	Maximum memory allowed for all the queries in a single server.

Table 12.1: Memory settings for query complexity

The following restriction settings are available in ClickHouse for reading data from the server:

Query complexity settings – read operations		
Setting	**Default value**	**Explanation**
`max_rows_to_read`	`0`	Maximum number of rows that can be read from a table.
`max_bytes_to_read`	`0`	Maximum amount of data (in bytes) that can be read from the table.
`read_overflow_mode`	`throw`	When the data volume is reached, either throw an exception or break the operation to return partial results.
`max_columns_to_read`	`0`	Maximum number of columns that can be read from a table in a single query.
`max_temporary_columns`	`0`	Maximum number of temporary columns that can be held in memory (including constant columns) while executing a query.
`max_temporary_non_const_columns`	`0`	Maximum number of temporary columns to be held in memory (excluding constant columns) while executing a query.

Table 12.2: Read operation settings for query complexity

The following restriction settings are available in ClickHouse pertaining to execution time and speed of read query:

Query complexity settings – query execution time and speed		
Setting	**Default value**	**Explanation**
`max_execution_time`	`0`	The upper limit for the execution time of a query. Default is unlimited.
`timeout_overflow_mode`	`Throw`	Either throw an exception when the execution timeout limit is reached or break the execution and return partial results.
`min_execution_speed`	`0`	Minimum execution speed in rows (per block) per second.
`max_execution_speed`	`0`	Maximum execution speed in rows (per block) per second.
`min_execution_speed_bytes`	`0`	Minimum execution speed in bytes (per block) per second.
`max_execution_speed_bytes`	`0`	Maximum execution speed in bytes (per block) per second.
`timeout_before_checking_execution_speed`	`10`	Time to wait until checking the execution speed.

***Table 12.3**: Query execution time and speed settings for query complexity*

The following restriction settings are available in ClickHouse pertaining to query clauses in read queries:

Query complexity settings – SQL clauses		
Setting	**Default value**	**Explanation**
`max_rows_to_group_by`	`0`	Maximum number of unique values allowed after performing a group by operation.
`group_by_overflow_mode`	`Throw`	Either throw an exception when the maximum group by row limit is reached or break the execution and return partial results.
`max_bytes_before_external_group_by`	`0`	Perform group by in external memory. The default setting is disabled (`0`).

`max_rows_to_sort`	`0`	Maximum rows from the query result that are allowed while sorting.
`max_bytes_to_sort`	`0`	Maximum data size from the query result that are allowed while sorting.
`sort_overflow_mode`	`Throw`	Either throw an exception when the maximum limit for sort data volume is reached or break the execution and return partial results.
`max_rows_in_distinct`	`0`	Maximum rows of distinct values from the query result.
`max_bytes_in_distinct`	`0`	Maximum size of distinct values from the query result (in bytes).
`distinct_overflow_mode`	`Throw`	Either throw an exception when the maximum limit for distinct data volume is reached or break the execution and return partial results.
`max_rows_in_join`	`0`	The number of rows in the hash table that is used when joining tables.
`max_bytes_in_join`	`0`	Maximum size of the hash table in bytes for the `JOIN` operation.
`join_overflow_mode`	`Throw`	Either throw an exception when the data volume limit for joins is reached or break the execution and return partial results.
`max_partitions_per_insert_block`	`100`	Maximum number of partitions allowed while performing the `INSERT` operation.

***Table 12.4**: Settings for SQL clauses*

The following restriction settings are available in ClickHouse pertaining to subqueries in a query:

Query complexity settings – subqueries		
Setting	**Default value**	**Explanation**
`max_subquery_depth`	`100`	Maximum depth allowed for subqueries.
`max_pipeline_depth`	`1000`	Maximum depth/the number of transformations that each data block goes through during query processing.
`max_ast_depth`	`1000`	Maximum nesting depth of a query syntactic tree.

max_ast_elements	50000	Maximum number of elements in a query syntactic tree.
max_rows_in_set	0	Maximum number of rows allowed for a data set in the IN clause created from a subquery.
max_bytes_in_set	0	Maximum size of data allowed for a data set in the IN clause created from a subquery.
set_overflow_mode	Throw	Either throw an exception when the data volume limit for IN clause is reached or break the execution and return partial results.
max_rows_to_transfer	0	Maximum number of rows that can be transferred to a remote server or stored in a temporary table when using GLOBAL IN.
max_bytes_to_transfer	0	Maximum size of data (in bytes) that can be transferred to a remote server or stored in a temporary table when using GLOBAL IN.
transfer_overflow_mode	Throw	Either throw an exception when the limit for data volume transfer is reached or break the execution and return partial results.

Table 12.5: Subquery settings for query complexity

The following restriction settings are available in ClickHouse pertaining to a result set from a read query:

Query complexity settings – query result		
Setting	**Default value**	**Explanation**
max_result_rows	0	Maximum number of rows allowed in a result set.
max_result_bytes	0	Maximum data size allowed (in bytes) in a result set.
result_overflow_mode	Throw	Either throw an exception when the limit is reached for maximum results data volume or break the execution and return partial results.

Table 12.6: Query result settings for query complexity

Settings profile

Different combination of settings are grouped under settings profile and stored in users.xml file. Each and every user in ClickHouse should be assigned a settings profile. In fresh installation, there will be a profile named 'default' and the database administrator can create new profiles based on the requirements. The following is an example of a custom profile:

```
<profiles>
  <analyst_profile>
    <max_rows_to_read>10000</max_rows_to_read>
    <max_bytes_to_read>100000000</max_bytes_to_read>
    <max_rows_to_sort>1000000</max_rows_to_sort>
    <max_bytes_to_sort>1000000000</max_bytes_to_sort>
    <max_result_rows>100000</max_result_rows>
    <max_result_bytes>100000000</max_result_bytes>
    <result_overflow_mode>break</result_overflow_mode>
    <max_execution_time>300</max_execution_time>
    <min_execution_speed>1000000</min_execution_speed>
    <max_columns_to_read>128</max_columns_to_read>
    <max_subquery_depth>10</max_subquery_depth>
    <max_pipeline_depth>100</max_pipeline_depth>
    <max_ast_depth>20</max_ast_depth>
    <max_ast_elements>100</max_ast_elements>
    <readonly>2</readonly>
  </ analyst_profile>
</profiles>
```

Quotas

In order to track resource usage or limit resource usage over a period of time, quotas can be used. Quota is applicable to a set of queries instead of a single query. In ClickHouse, the default quota tracks hourly resource consumption, without any restrictions on the usage. The resource consumption is calculated for each interval and is written to the server log file. The following quota settings are available for which the limits can be set:

- **`duration`**: Time duration for which the limits apply. Even if the limits are not set, the resource usage information is written to the log file, collected values are cleared, and the calculation starts over from the beginning.
- **`queries`**: The total number of requests allowed within the specified duration.
- **`errors`**: The number of queries that threw an exception within the specified duration
- **`result_rows`**: The total number of rows given as the result within the set duration.
- **`read_rows`**: The total number of source rows read from tables for running the query, including the remote servers within the set duration.
- **`execution_time`**: The total query execution time, in seconds, accumulated within the set duration.

Sample quota definition in users.xml is given as follows:

```
<quotas>
  <my_quota>
    <interval>
      <duration>3600</duration>
      <queries>8192</queries>
      <errors>1000</errors>
      <result_rows>1000000</result_rows>
      <read_rows>1000000000</read_rows>
      <execution_time>60</execution_time>
    </interval>
  </ my_quota >
</quotas>
```

User settings

Users can be added to the ClickHouse server either by configuring the users in **`users.xml`** file or via SQL. The following sample configuration illustrates adding users via XML configuration:

```
<users>
  <clickhouse_user>
    <password>ClickHouse123!</password>
```

```
    <networks incl="networks" replace="replace">
      <ip>127.0.0.1</ip>
      <host>mydomain.com</host>
    </networks>
    <profile>custom_profile</profile>
    <quota>default</quota>
  </clickhouse_user >
</users>
```

The preceding configuration is for a user named **clickhouse_user** and the password for the user is **ClickHouse123!**. The user is assigned a profile named **custom_profile** and assigned a **default** quota. It is also possible to restrict the login for the user based on IP or hostname.

Role-based access control

ClickHouse supports access control management (based on role-based access control approach) via SQL-driven workflow. The restrictions and permissions are granted based on user roles. The following access entities can be configured via the SQL-driven workflow:

- Role
- User account
- Row policy
- Settings profile
- Quota

Role

The role consists of a set of configurations for the access entities. Roles hold the information about privileges, settings, and constraints and a list of roles to be granted for a set of users. The syntax for creating a user role is as follows:

```
CREATE ROLE [IF NOT EXISTS | OR REPLACE] role_name
[SETTINGS settings_name [= value] [MIN [=] min_value] [MAX [=] max_
value] [READONLY|WRITABLE] | PROFILE 'settings_profile_name']
```

Example:

```
CREATE ROLE analyst;
GRANT SELECT ON electricity.* TO analyst;
```

The preceding example creates a role called **analyst**. Since no profile or settings are specified, the settings from the **default** profile are inherited. In the next statement, the users under the role **analyst** are granted permission to execute the **SELECT** statement on all the tables under the **electricity** database.

The roles once created can be modified using the **ALTER** statement. The following is the syntax for altering the role:

```
ALTER ROLE [IF EXISTS] role_name [RENAME TO new_role_name] [SETTINGS
variable [= value] [MIN [=] min_value] [MAX [=] max_value]
[READONLY|WRITABLE] | PROFILE 'settings_profile_name']
```

Example:

```
ALTER ROLE analyst RENAME TO data_analyst;
```

The preceding example alters the **analyst** role and renames the role to **data_analyst**. The other operations permitted to roles are:

- **SET ROLE**: Sets a role to the current user.
- **SET DEFAULT ROLE**: Assigns a default role to the user from one of the previously assigned roles.
- **SHOW ROLES**: Displays the list of roles available.
- **SHOW CREATE ROLE**: Displays the SQL statement to create the role.
- **DELETE ROLE**: Removes the user role.

Example:

```
SET ROLE data_analyst;

SET DEFAULT data_analyst;

SHOW ENABLED ROLES;

SHOW CURRENT ROLES;

SHOW CREATE ROLE data_analyst;

DELETE ROLE data_analyst;
```

Row policy

Row policy is used to allow or restrict the rows from ClickHouse tables that are available to a user or a role. The syntax for creating row policy is as follows:

```
CREATE [ROW] POLICY [IF NOT EXISTS | OR REPLACE] policy_name ON [db.]
table

[AS {PERMISSIVE | RESTRICTIVE}]
```

```
[FOR SELECT]
[USING condition]
[TO {role [,...] | ALL | ALL EXCEPT role [,...]}]
```

Example:

```
CREATE ROW POLICY row_policy_1 ON electricity.consumption
FOR SELECT USING Voltage>250.0 TO 'data_analyst';
```

The preceding example creates a row policy named **`row_policy_1`** and allows **`data_analyst`** to access rows in the table **`electricity.consumption`** with **`Voltage`** column values greater than **`250`**. Other operations allowed for row policy are given as follows:

- **ALTER ROW POLICY**
- **SHOW ROW POLICIES**
- **SHOW CREATE ROW POLICY**
- **DROP ROW POLICY**

Examples:

```
ALTER ROW POLICY row_policy_1
FOR SELECT USING Voltage>220.0 TO data_analyst;
SHOW ROW POLICIES;
SHOW CREATE ROW POLICY row_policy_1;
DROP ROW POLICY row_policy_1;
```

Settings profile

A group of settings can be clubbed together and stored as a settings profile. Settings profile can be applied to users to assign the settings, restrictions, and list of user roles allowed for the user. The syntax to create a settings profile is as follows:

```
CREATE SETTINGS PROFILE [IF NOT EXISTS | OR REPLACE TO]  settings_
profile_name [SETTINGS variable [= value] [MIN [=] min_value] [MAX [=]
max_value] [READONLY|WRITABLE] | INHERIT 'profile_name']
```

Example:

```
CREATE SETTINGS PROFILE rows_sort_profile SETTINGS max_rows_to_sort =
2048 MIN 1024 MAX 2100 TO clickhouse_user;
```

The preceding example has a setting to limit the number of rows in sort operation and is saved to profile named **`rows_sort_profile`** and assigned to the user named

clickhouse_user. Other operations allowed with respect to settings profile are as follows:

- **ALTER SETTINGS PROFILE**
- **DROP SETTINGS PROFILE**
- **SHOW CREATE SETTINGS PROFILE**
- **SHOW PROFILES**

Example:

```
ALTER SETTINGS PROFILE rows_sort_profile SETTINGS max_rows_to_sort = 4096
MIN 1024 MAX 8182 ;


SHOW CREATE SETTINGS PROFILE rows_sort_profile;

SHOW SETTINGS PROFILES;

DROP SETTINGS PROFILE rows_sort_profile;
```

Quotas

Quotas are used to track resource usage or limit resource usage over a period of time. The list of users/roles to which the quotas apply are also defined. The syntax to create quotas is as follows:

```
CREATE QUOTA [IF NOT EXISTS | OR REPLACE] quota_name

[KEYED BY {'none' | 'user name' | 'ip address' | 'forwarded ip address'
| 'client key' | 'client key or user name' | 'client key or ip
address'}]

[FOR [RANDOMIZED] INTERVAL number {SECOND | MINUTE | HOUR | DAY | WEEK |
MONTH | QUARTER | YEAR}

{MAX { {QUERIES | ERRORS | RESULT ROWS | RESULT BYTES | READ ROWS | READ
BYTES | EXECUTION TIME} = number } [,...] |

NO LIMITS | TRACKING ONLY}]

[TO {role [role_name] | ALL | ALL EXCEPT role [list_of_role_names]}]
```

Example:

```
CREATE QUOTA my_quota1 FOR INTERVAL 1 MONTH MAX QUERIES 1024 TO
clickhouse_user;
```

In the preceding example, a quota named **my_quota1** is created and assigned to the user named **clickhouse_user**. Apart from these, other operations allowed with respect to quotas are:

- **ALTER QUOTA**
- **DROP QUOTA**
- **SHOW CREATE QUOTA**
- **SHOW QUOTA**

Example:

```
ALTER QUOTA my_quota1 FOR INTERVAL 1 MONTH MAX QUERIES 2048;

SHOW CREATE QUOTA my_quota1;

SHOW QUOTAS;

DROP QUOTA my_quota1;
```

User account

User accounts help the individual end users or an application to connect and access the data stored in ClickHouse. Every user account has at least a settings profile and quota attached to it. In addition to these, roles are also associated with the user. The syntax to create a user account is as follows:

```
CREATE USER [IF NOT EXISTS | OR REPLACE] user_name

[IDENTIFIED [WITH {NO_PASSWORD | PLAINTEXT_PASSWORD| SHA256_
PASSWORD| SHA256_HASH| DOUBLE_SHA1_PASSWORD| DOUBLE_SHA1_HASH} ] BY
{'password'|'hash'}]

[HOST {LOCAL | NAME 'host_name' | REGEXP 'host_name_regexp' | IP 'host_
ip_address' | LIKE 'pattern'} [,...] | ANY | NONE]

[DEFAULT ROLE [,...]]

[SETTINGS variable [= value] [MIN [=] min_value] [MAX [=] max_value]
[READONLY|WRITABLE] | PROFILE 'user_profile_name']
```

Example:

```
CREATE USER IF NOT EXISTS clickhouse_analyst

IDENTIFIED WITH SHA256_HASH BY
'a1fe8f79a121256842e7aaef2ab1e339a553a74fe05834ca081259cf66ac5fb5'

HOST IP '127.0.0.1'

DEFAULT ROLE 'data_analyst'

PROFILE 'default';
```

The passwords can be specified in any of the following password formats allowed. Hashed passwords are recommended instead of plain text passwords since they are safer:

- **no_password**
- **plaintext_password**
- **sha256_password**
- **sha256_hash**
- **double_sha1_password**
- **double_sha1_hash**

Similarly, allowed hosts can be specified in any of the following formats:

- `HOST IP`: This is the IP address from which the user is allowed to connect to ClickHouse.
- `HOST ANY`: The user can connect from any IP address.
- `HOST LOCAL`: The user can connect locally from the server in which ClickHouse is installed.
- `HOST NAME`: This parameter is for a fully qualified domain name. For example: `mydomain.com`.
- `HOST NAME REGEXP`: This is the host name in regular expression format.
- `HOST LIKE`: This is to identify the allowed hosts via the LIKE operator. For example: `%mydomain.%`.

Roles can be assigned to user accounts via `GRANT` statements. Other operations permitted for a user account are as follows:

- **ALTER USER**
- **SHOW USERS**
- **SHOW CREATE USER**
- **DROP USER**

Example:

```
ALTER USER clickhouse_analyst DEFAULT ROLE 'new_role_name';
SHOW USERS;
SHOW CREATE USER clickhouse_analyst;
DROP USER IF EXISTS clickhouse_analyst;
```

System tables

System tables are tables located in the `system` database, and they cannot be dropped or altered. The tables are created when the ClickHouse server starts and stores the data predominantly in RAM. They store information about ClickHouse's internal process, server states, and information about the environment. The following are the important tables that can be used to provide diagnostic information, details on ClickHouse server operations, and are commonly used by database administrators.

Name	Functionality
`asynchronous_metric_log`	Stores the historical data from the `asynchronous_metrics` table, and the data is stored once per minute.
`asynchronous_metrics`	Metrics pertaining to the hardware resource usage of ClickHouse such as the number of background threads, RAM usage, and so on are stored.
`Clusters`	Holds information on the clusters and the servers that are part of the clusters.
`Columns`	Contains information about the columns available in all the tables.
`data_type_families`	The supported data types and case insensitivity information is available in this table.
`Databases`	Holds the list of databases available for the current user.
`detached_parts`	This table has the information pertaining to the detached parts of MergeTree tables.
`Dictionaries`	Holds the information about the external dictionaries created by the users.
`Disks`	This table has the information about the data path used for storing the data in ClickHouse.
`Events`	This table has the information about the events that have occurred in the ClickHouse.
`Formats`	This table has the information about the input and output data formats supported in ClickHouse.
`Functions`	All the regular functions and aggregate functions supported are available here.
`graphite_retentions`	This table has rollup parameters used in the `GraphiteMergeTree` engine.

Macros	Macros and substitutions that are defined in config.xml are available here.
merge_tree_settings	Settings configured for MergeTree engine tables are stored here.
Merges	Merges and part mutations that are currently in process for the MergeTree tables are available here.
metric_log	Contains the older metrics from `system.metrics` and `system.events` table.
Metrics	Present metrics like the number of queries, number of background merges are stored here.
Mutations	Stores information about mutation commands (`DELETE`, `UPDATE`, and so on) and their status.
Parts	This table has the information about the data parts from the MergeTree table.
parts_columns	This table has the information about the data part columns from the MergeTree table.
Processes	Holds information on the current processes of ClickHouse.
query_log	Contains information about the queries that were executed.
query_thread_log	Contains information about the query execution threads.
quota_usage	Contains usage details and remaining quota for the current user.
Quotas	Contains the list of available quotas.
Replicas	This table has information on the replicated tables available.
replication_queue	Contains information about tasks that are in Zookeeper's replication queues.
row_policies	Contains information about row policies for the tables and users to which those policies are applicable.
Settings	This table has the session settings for the current user.
stack_trace	Contains stack traces of all the server threads.

`storage_policies`	Stores the information on server storage policies and volumes.
`table_engines`	Has the list of table engines supported by the ClickHouse.
`table_functions`	Has the list of table functions supported by the ClickHouse.
`Tables`	This table has the list of all non-detached tables that are available to ClickHouse.
`trace_log`	Stores the stack trace information from the sampling query profiler.

Table 12.7: Important system tables

Conclusion

In this chapter, we have seen the configuration settings for query complexity, settings profile, quotas, and user information that are configured in `users.xml`. These settings are commonly accessed and updated by database administrators who maintain the ClickHouse setup. With this, we have come to the end of the configurations available in ClickHouse.

Points to remember

- Query complexities, settings profile, quotas, and user configurations are available in the `users.xml` file.
- ClickHouse supports SQL-driven workflow for user management and access control.
- User roles, settings profiles, quotas, and row policies can also be set via SQL statements.
- System tables are available, which can provide useful information about the ClickHouse server.

Multiple choice questions

1. **Query permissions and user settings are available in _____.**
   a) settings.xml
   b) users.xml

2. **Different combinations of query-related settings are grouped under the settings profile.**
   a) True
   b) False

3. **Hardware resource-related settings are available in _____ settings.**
   a) profile
   b) quota
   c) user

4. **Settings profile can be set via users.xml and SQL-based workflow.**
   a) True
   b) False

5. **System tables are read-only and are predominantly loaded in RAM.**
   a) True
   b) False

Answers

1. **b**
2. **a**
3. **b**
4. **a**
5. **a**

Appendix A

Installing Lubuntu 20.04 in Oracle Virtualbox 6.1

1. Open Virtualbox and click on **New** (*Ctrl* + *N*):

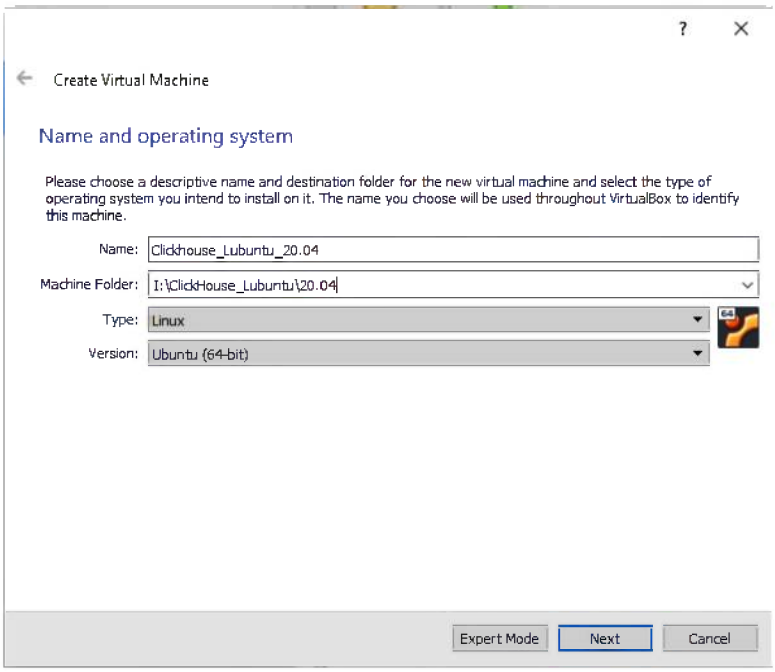


***Figure A.1**: Configuring our sandbox/virtual machine in Oracle Virtualbox*

2. Enter the **Name**, **Folder** to store the Virtual OS, **Type** – **Linux**, and **Version** – **Ubuntu (64-bit)**. Click **Next**.

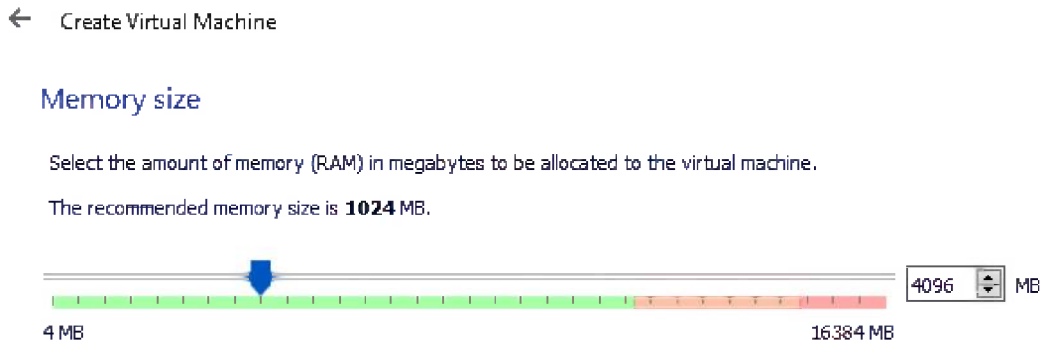


Figure A.2: Configuring the RAM size for the sandbox

3. Select the RAM for the OS. This value will depend on the available memory in the user's computer (higher is better). Select **Next** once the desired value is entered.

4. In the next screen, select **Create a virtual hard disk now**:

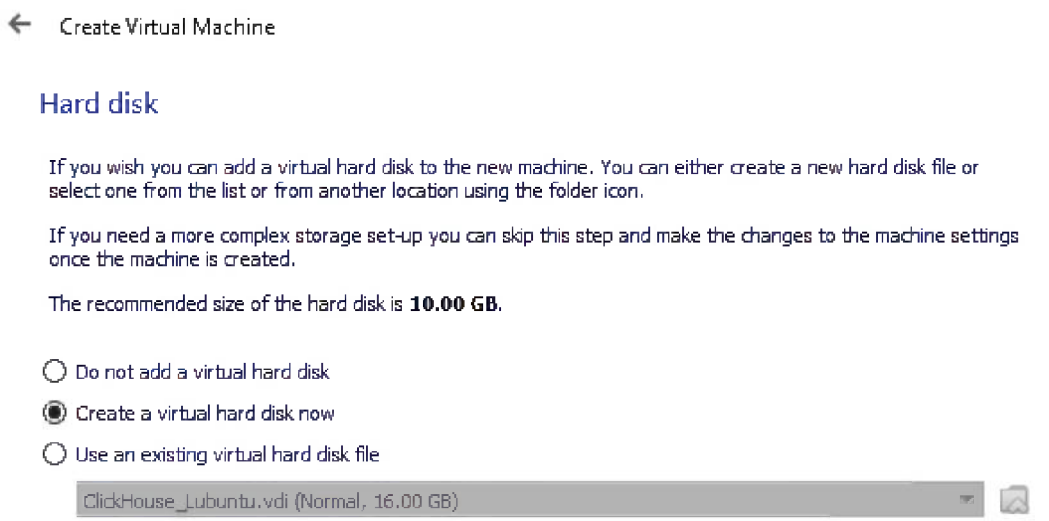


Figure A.3: Creating a virtual hard disk for the new sandbox

5. In the next screen, select **`VDI`** and click on the **`Next`** button:

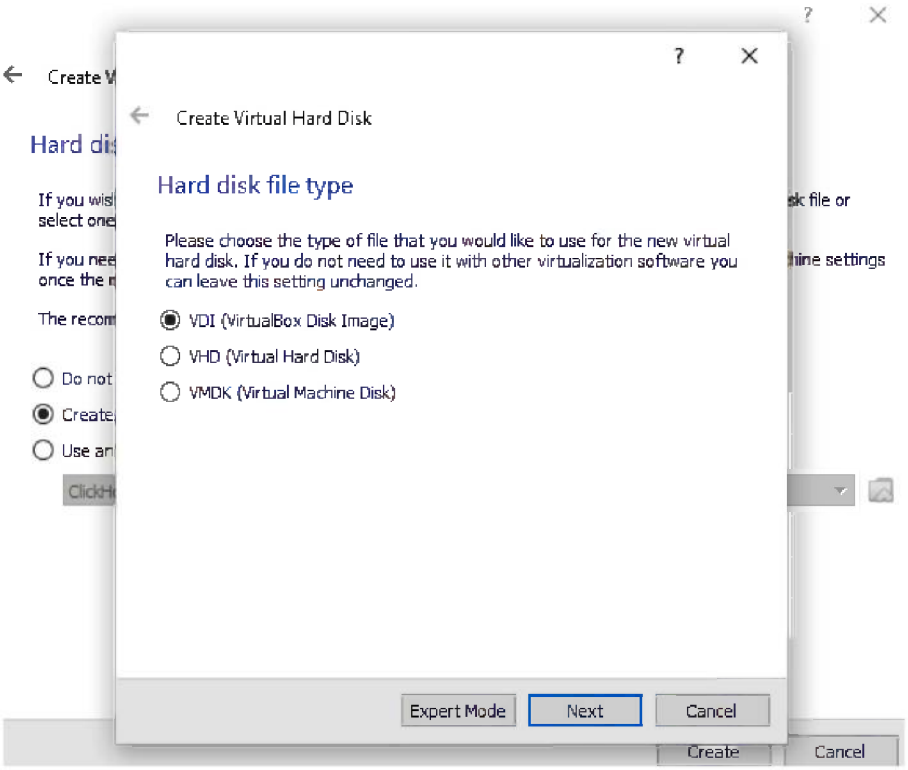


***Figure A.4**: Choosing the virtual hard disk type for the new sandbox*

6. In the next screen, select **`Dynamically allocated`** and click on the **`Next`** Button.

***Figure A.5**: Choosing dynamically allocated hard disk for the new sandbox*

7. In the next screen, select the virtual hard disk location and limit (higher is better). Click on the **`Create`** button to create the virtual hard disk.

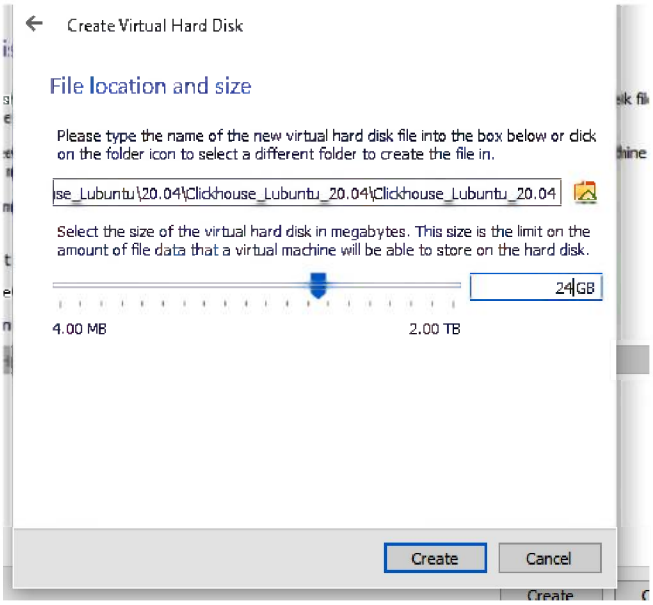


***Figure A.6**: Choosing the file location for the virtual hard disk*

8. Meanwhile, download the Lubuntu image from this location (20.04 series).
   https://lubuntu.me/downloads/

9. The following screen will display the list of the available virtual OS. Select the newly created virtual OS and double click the selection:

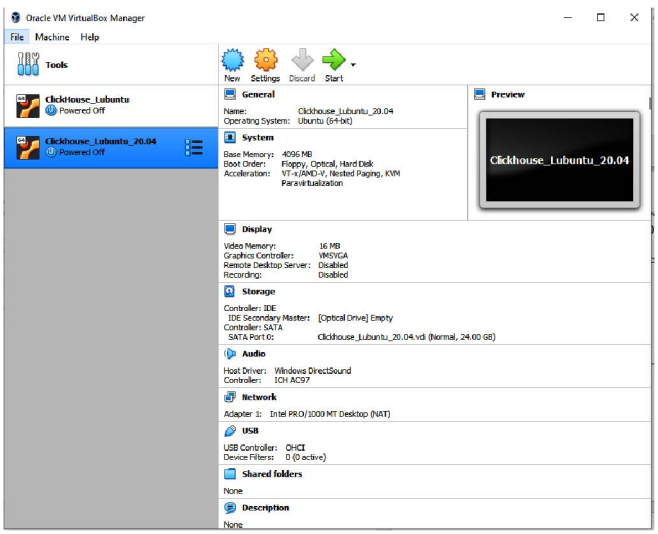


***Figure A.7**: Staring the new sandbox*

10. Choose the downloaded Lubuntu image and click on the **`Start`** button:

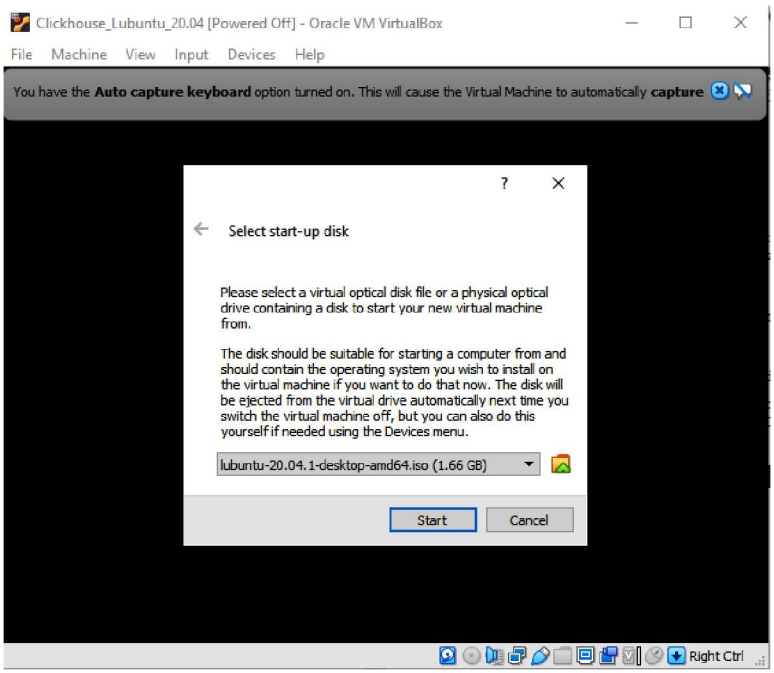


***Figure A.8**: Choosing the bootable OS image*

11. Choose the English language for installation (already selected by default) and proceed by pressing *Enter*. Then select **`Start Lubuntu`** and hit the *Enter* button:

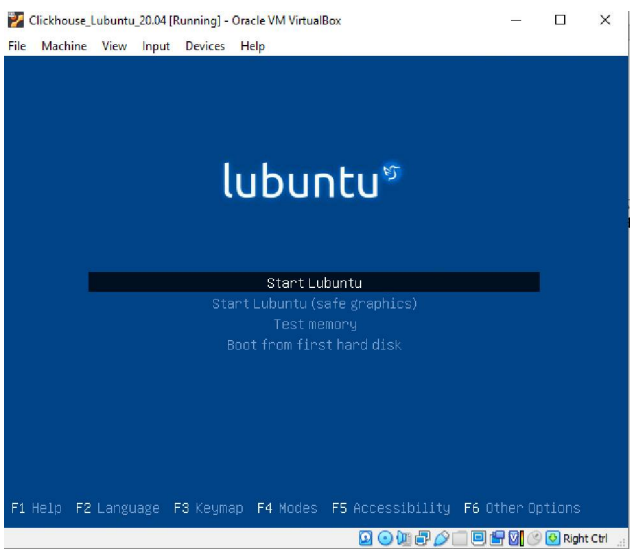


***Figure A.9**: Booting the sandbox OS for installation*

12. Once you are in the booted Live OS, select `Install Lubuntu 20.04 LTS` icon to permanently install the virtual OS:

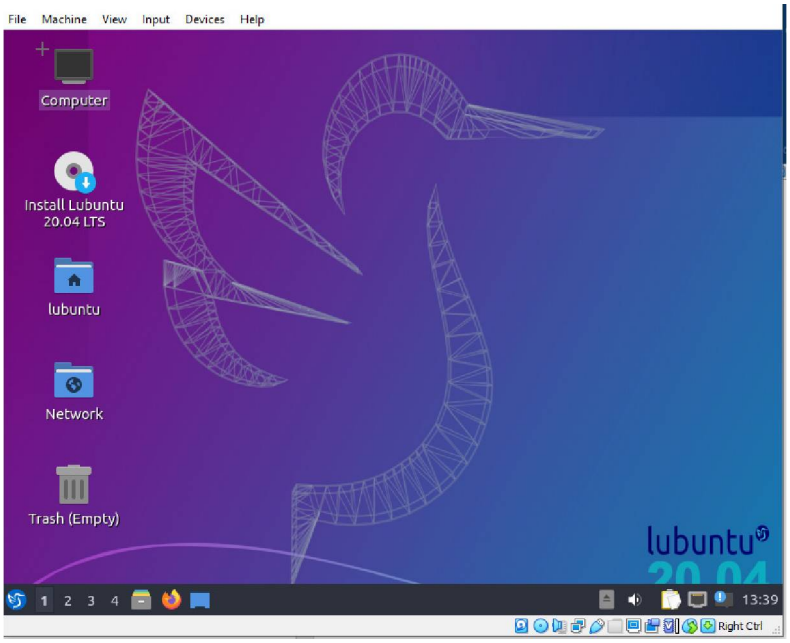


Figure A.10: Live OS for installation

13. In the next installer screen, choose the installer language and click on the `Next` button:

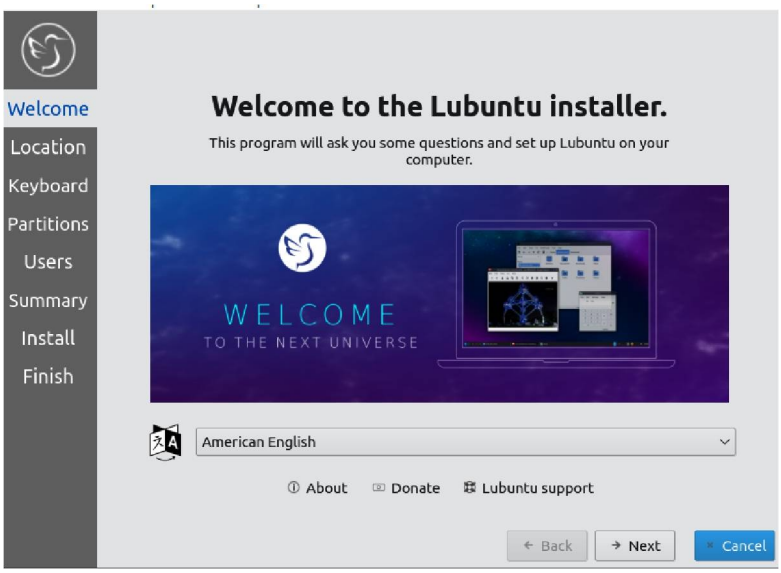


Figure A.11: Beginning the installation

14. Select the time zone and region in the next screen and proceed:

***Figure A.12**: Choosing the timezone*

15. In the next screen, select the keyboard layout and proceed:

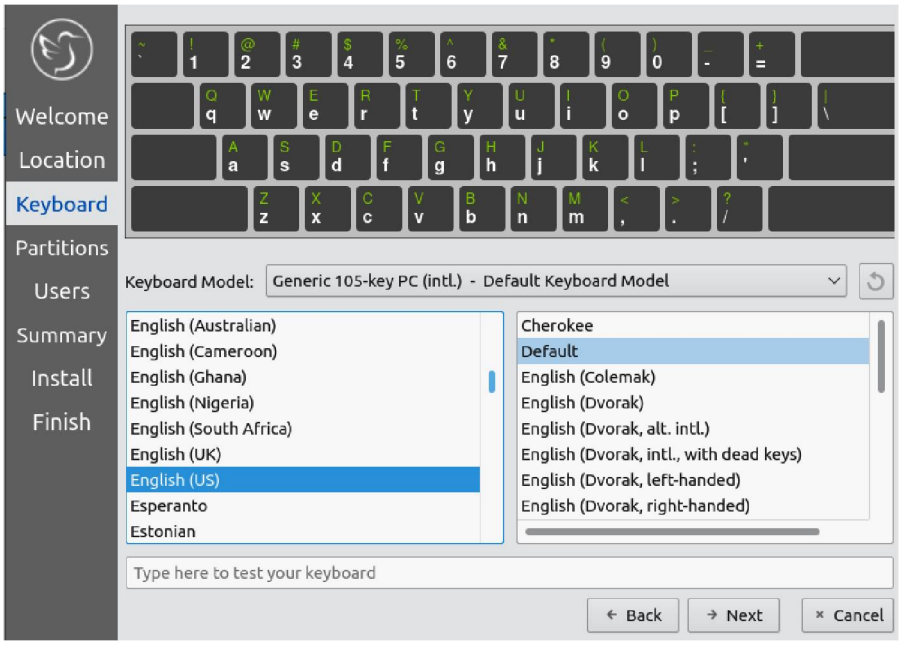


***Figure A.13**: Choosing the keyboard layout*

16. In the next screen, choose the **`Erase disk`** option and proceed further:

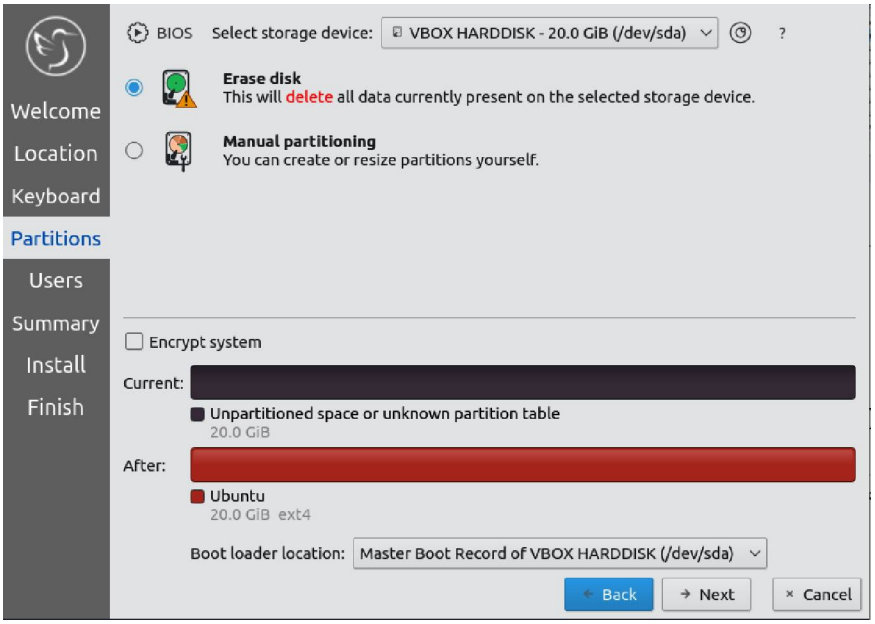


***Figure A.14**: Erasing the virtual disk before installation*

17. In the next screen, enter the user name, login name, computer name, and password to log in. Make sure to remember the password.

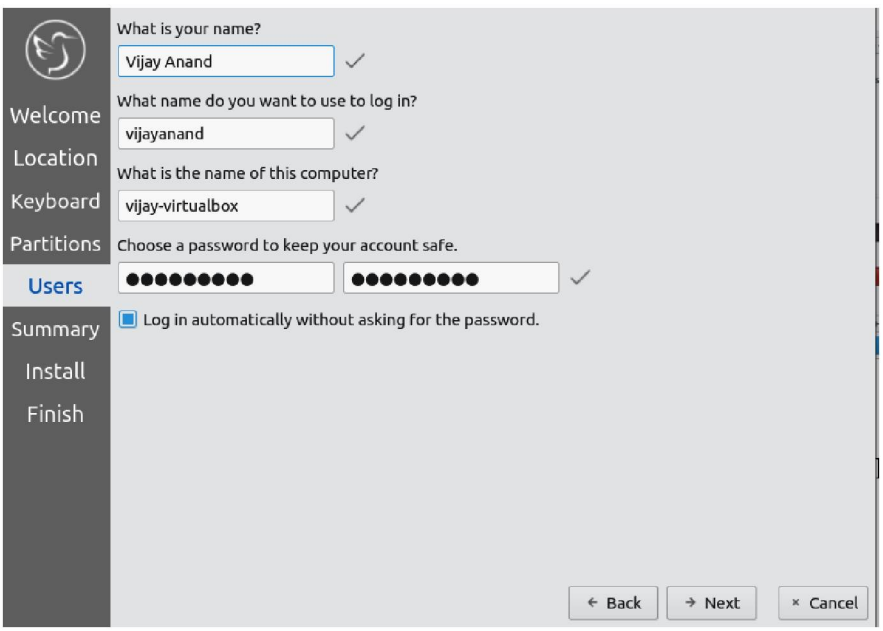


***Figure A.15**: Choosing the username and password*

18. In the next screen, check the summary and if everything is ok, click on the **`Install`** button to install the OS. In the confirmation box, select the **`Install now`** option.

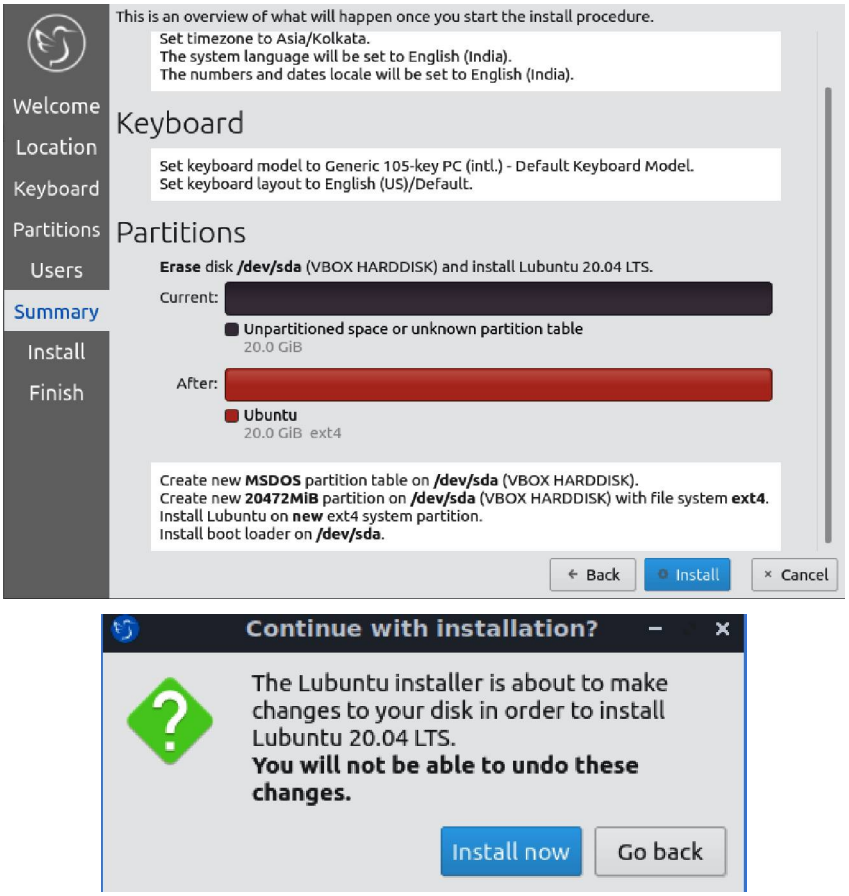


***Figure A.16**: Beginning the installation*

19. Wait until the installation is over.

***Figure A.17**: Installation process*

20. Once the installation is over, select **`Restart now`** and press **`Done`**. Once the OS restarts, we will be able to use it for installing the necessary software.

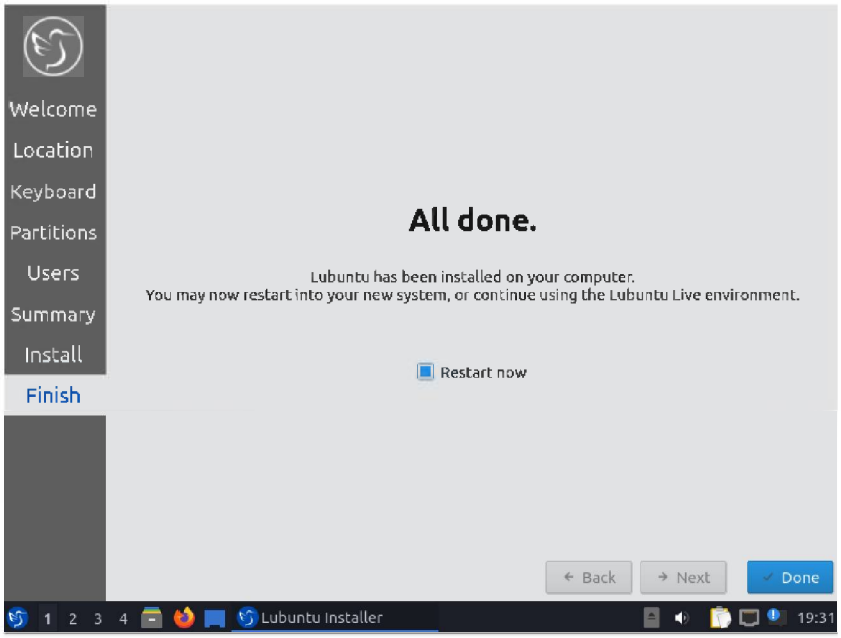


***Figure A.18**: Finishing the installation*

APPENDIX B
Installing External Data Sources

Setting up Kafka for Testing Kafka Integration

The following steps can be used to download and install Apache Kafka in Ubuntu 20.04.

1. Java should be installed before installing Kafka.

```
$ cd ~
$ sudo apt install default-jre default-jdk
$ mkdir kafka
$ cd kafka
$ wget https://mirrors.estointernet.in/apache/kafka/2.6.0/
kafka_2.13-2.6.0.tgz
$ tar -xzf kafka_2.13-2.6.0.tgz
$ cd kafka_2.13-2.6.0
```

2. Run this to start a Zookeeper server.

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

3. Open a new terminal window/tab and start the Kafkaserver, without closing any terminal windows/tabs opened earlier:

```
$ ~/kafka/kafka_2.13-2.6.0/bin/kafka-server-start.sh config/
server.properties
```

4. Open a new terminal session and run this to create a topic:

```
$ ~/kafka/kafka_2.13-2.6.0/bin/kafka-topics.sh --create --topic
first_installation_testing --bootstrap-server localhost:9092
```

5. To publish messages in a new topic:

```
$ ~/kafka/kafka_2.13-2.6.0/bin/kafka-console-producer.sh --topic
first_installation_testing --bootstrap-server localhost:9092
```

6. To consume messages from a topic:

```
$ ~/kafka/kafka_2.13-2.6.0/bin/kafka-console-consumer.sh --topic
first_installation_testing --from-beginning --bootstrap-server
localhost:9092
```

MySQL installation

MySQL is a popular open-source relational DBMS, which was developed by MySQL AB, a Swedish company and it was first released in 1995. Currently, it is now developed and maintained by Oracle Corporation.

The reason for installing MySQL is to demonstrate ClickHouse's functionality of connecting and querying external databases via ODBC and JDBC drivers. ClickHouse has a dedicated MySQL engine that can be used to exchange data between ClickHouse and MySQL.

Note that these installation instructions and configurations are not intended to be followed in any production setup. This is for novice users and targeted at ease of use and less emphasis on security. Let us begin the installation by executing the following command in the terminal:

```
$ sudo apt install mysql-server
```

Then, we should run the security script to make additional configuration changes.

```
$ sudo mysql_secure_installation
```

The following message should be displayed. We can ignore this for now since we are going to use MySQL for testing/learning purpose only:

```
Securing the MySQL server deployment.

Connecting to MySQL using a blank password.
```

```
VALIDATE PASSWORD COMPONENT can be used to test passwords
and improve security. It checks the strength of password
and allows the users to set only those passwords which are
secure enough. Would you like to setup VALIDATE PASSWORD component?

Press y|Y for Yes, any other key for No:
```

Hit the *Enter* key (without typing anything) and we will be prompted to set the root user password. The root user is a superuser and has all the privileges to this database. For the test installation, we shall set the root password to **`mysql_clickhouse`**. Type the password without quotes and hit the Enter key and we will be prompted to re-enter the password for confirmation. Once confirmed, the following message will be displayed:

```
By default, a MySQL installation has an anonymous user,
allowing anyone to log into MySQL without having to have
a user account created for them. This is intended only for
testing, and to make the installation go a bit smoother.
You should remove them before moving into a production
environment.

Remove anonymous users? (Press y|Y for Yes, any other key for No) :
```

We can ignore removing anonymous users, So hit the *Enter* key (without typing anything) to move to the next step. The following message is displayed:

```
Normally, root should only be allowed to connect from
'localhost'. This ensures that someone cannot guess at
the root password from the network.

Disallow root login remotely? (Press y|Y for Yes, any other key for No):
```

This setting can be skipped and hit the Enter key (without typing anything) to move to the next configuration step:

```
By default, MySQL comes with a database named 'test' that
anyone can access. This is also intended only for testing,
and should be removed before moving into a production
environment.
```

```
Remove test database and access to it? (Press y|Y for Yes, any other key
for No) :
```

We shall keep the test database in case we want to test MySQL after installation. So, hit the Enter key to keep them for now. This will be followed by the prompt:

```
Reloading the privilege tables will ensure that all changes
made so far will take effect immediately.


Reload privilege tables now? (Press y|Y for Yes, any other key for No) :
```

Since we need to update our changes, type **y** (without quotes) and hit the Enter key. The following success message should be displayed:

```
Success.


All done!
```

We are all set to use MySQL now. To access MySQL prompt (similar to **clickhouse**-client), type the following in the terminal:

```
$ sudo mysql
```

Once the command is executed, the following message will be displayed:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 19
Server version: 8.0.21-0ubuntu0.20.04.4 (Ubuntu)


Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights
reserved.


Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.


Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.


mysql>
```

Try executing the following commands, similar to what we have done after ClickHouse installation:

```
mysql> SHOW DATABASES;


mysql> USE mysql;


mysql> SHOW TABLES;
```

Installing sample database in MySQL

Employees data set can be added to MySQL based on the scripts in the following repository:

https://github.com/datacharmer/test_db

Installation is straightforward. Either download the whole repository as ZIP file (`master.zip`) via browser and create a folder named employees and move the downloaded ZIP file or open a terminal and execute the following commands:

```
$ mkdir employees
$ cd employees
$ wget https://github.com/datacharmer/test_db/archive/master.zip
```

Once the `master.zip` is available in the location, execute the following commands in the terminal (if downloaded manually via the browser, navigate to the folder containing master.zip file via the terminal):

```
$ unzip master.zip
$ cd test_db-master/
$ sudo mysql -t < employees.sql
```

This should create a database called the `employee` and few tables under it. To test the tables, start the MySQL prompt (using `sudo mysql` command in the terminal) and execute the following SQL statements and verify the output:

```
mysql> SHOW DATABASES;
mysql> USE employees;
mysql> SHOW TABLES;
```

Let us continue creating a user for accessing the database via DBeaver.

```
mysql> CREATE USER 'mysql_user'@'localhost' IDENTIFIED BY 'clickhouse';
mysql> ALTER USER 'mysql_user'@'localhost' IDENTIFIED WITH mysql_native_
```

```
password BY 'clickhouse';
mysql> GRANT ALL PRIVILEGES ON employees.* TO 'mysql_user'@'localhost';
mysql> FLUSH PRIVILEGES;
```

We have successfully installed and created a sample database and tables in the MySQL database. Let us now connect to MySQL via DBeaver. Similar to what we have done for ClickHouse, search for **MySQL 8+** in the new database connection option in DBeaver, provide a user name (**mysql_user**) and password, which we had created recently. Download the necessary drivers and connect to the database.

To start querying after a successful connection, open a new SQL editor window for MySQL (cannot execute SQL statements across multiple database connections from same SQL editor window in DBeaver) and try executing the following SQL statements:

```
SELECT * FROM employees.departments LIMIT 5;
SELECT * FROM employees.dept_emp LIMIT 5;
SELECT * FROM employees.dept_manager LIMIT 5;
SELECT * FROM employees.employees LIMIT 5;
SELECT * FROM employees.salaries LIMIT 5;
SELECT * FROM employees.titles LIMIT 5;
```

That's all. We have successfully installed MySQL and tested it with sample data. We can also connect to MySQL via DBeaver and test the same.

Installing Postgresql

Go to the Linux terminal and start the **postgres** installation by executing the following command:

```
$ sudo apt install postgresql postgresql-contrib
```

When the installation is complete, we can enter the command line client for **postgres** by executing the following command. A user named **postgres** will be already available and we can make use of it.

```
$ sudo -u postgres psql
```

Let us set the password for the user named **postgres**.

```
postgres=# alter user postgres with encrypted password postgres;
```

We shall use the DVD rental database for our tutorials. We shall begin by creating the database via the following SQL statements and make sure **postgres** user has all privileges to it.

```
postgres=#  CREATE DATABASE dvdrental;
postgres=#  GRANT ALL PRIVILEGES ON DATABASE dvdrental TO postgres;
```

Now exit the postgres CLI by pressing *Ctrl + D* keys or by typing **\q** and pressing enter. We have to change the authentication method of the **postgres** user to install the sample database. To open the config file for editing, execute the following command:

```
$ sudo nano /etc/postgresql/12/main/pg_hba.conf
```

Search for the following lines and replace the **peer** authentication method with **md5**, which is a password authentication method.

```
From:
local   all             postgres                                peer
To:
local   all             postgres                                md5
```

The next step is to download the **dvd** rental database. Enter the following commands to install the database in **postgresql** server:

```
$ wget https://sp.postgresqltutorial.com/wp-content/uploads/2019/05/
dvdrental.zip
$ unzip dvdrental.zip
$ pg_restore -U postgres -d dvdrental dvdrental.tar
```

We can connect to the database via DBeaver and test the data in the tables.

Installing ClickHouse JDBC bridge

The JDBC bridge is available in the official ClickHouse repository and the installation instructions are available there for multiple platforms. **https://github.com/ClickHouse/clickhouse-jdbc-bridge**.

Steps to install:

```
$ sudo apt update && apt install -y procps wget
$ wget https://github.com/ClickHouse/clickhouse-jdbc-   bridge/releases/
download/v2.0.1/clickhouse-jdbc-bridge_2.0.1-1_all.deb
$ sudo apt install --no-install-recommends -f ./clickhouse-jdbc-
bridge_2.0.1-1_all.deb
$ clickhouse-jdbc-bridge
```

Index