How To Install and Use ClickHouse on Ubuntu 20.04

Published on September 22, 2020

- Big Data
- Ubuntu 20.04
- Databases

bsder and Kathryn Hancox

Not using Ubuntu 20.04?Choose a different version or distribution.

Ubuntu 20.04

The author selected the Free and Open Source Fund to receive a donation as part of the Write for DOnations program.

Introduction

ClickHouse is an open source, column-oriented analytics database created by Yandex for OLAP and big data use cases. ClickHouse's support for real-time query processing makes it suitable for applications that require sub-second analytical results. ClickHouse's query language is a dialect of SQL that enables powerful declarative querying capabilities while offering familiarity and a smaller learning curve for the end user.

Column-oriented databases store records in blocks grouped by columns instead of rows. By not loading data for columns absent in the query, column-oriented databases spend less time reading data while completing queries. As a result, these databases can compute and return results much faster than traditional row-based systems for certain workloads, such as OLAP.

Online Analytics Processing (OLAP) systems allow for organizing large amounts of data and performing complex queries. They are capable of managing petabytes of data and returning query results quickly. In this way, OLAP is useful for work in areas like data science and business analytics.

In this tutorial, you'll install the ClickHouse database server and client on your machine. You'll use the DBMS for typical tasks and optionally enable remote access from another server so that you'll be able to connect to the database from another machine. Then you'll test ClickHouse by modeling and querying example website-visit data.

Prerequisites

- One Ubuntu 20.04 server with a `sudo` enabled non-root user and firewall setup. The server should have at least 2GB of RAM. You can follow the Initial Server Setup tutorial to create the user and set up the firewall.
- (Optional) A secondary Ubuntu 20.04 server with a `sudo` enabled non-root user and firewall setup.

Step 1 — Installing ClickHouse

In this section, you will install the ClickHouse server and client programs using `apt`.

First, SSH into your server by running:

```
ssh sammy@your_server_ip
```

Copy

Yandex maintains an APT repository that has the latest version of ClickHouse. Add the repository's GPG key so that you'll be able to securely download validated ClickHouse packages:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E0C56BD4
```

Copy

You will see output similar to the following:

```
OutputExecuting: /tmp/apt-key-gpghome.JkkcKnBAFY/gpg.1.sh --keyserver
keyserver.ubuntu.com --recv E0C56BD4

gpg: key C8F1E19FE0C56BD4: public key "ClickHouse Repository Key
<milovidov@yandex-team.ru>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

The output confirms it has successfully verified and added the key.

Add the repository to your APT repositories list by executing:

```
echo "deb http://repo.yandex.ru/clickhouse/deb/stable/ main/" | sudo tee
/etc/apt/sources.list.d/clickhouse.list
```

Copy

Here you've piped the output of `echo` to `sudo tee` so that this output can print to a root-owned file.

Now, update your packages:

```
sudo apt update
```

Copy

The `clickhouse-server` and `clickhouse-client` packages will now be available for installation. Install them with:

```
sudo apt install clickhouse-server clickhouse-client
```

Copy

During the installation, you will be asked to set a password for the default ClickHouse user.

You've installed the ClickHouse server and client successfully. You're now ready to start the database service and ensure that it's running correctly.

Step 2 — Starting the Service

The `clickhouse-server` package that you installed in the previous section creates a `systemd` service, which performs actions such as starting, stopping, and restarting the database server. `systemd` is an init system for Linux to initialize and manage services. In this section you'll start the service and verify that it is running successfully.

Start the `clickhouse-server` service by running:

```
sudo service clickhouse-server start
```

Copy

The previous command will not display any output. To verify that the service is running successfully, execute:

```
sudo service clickhouse-server status
```

Copy

You'll see output similar to the following:

```
Output● clickhouse-server.service - ClickHouse Server (analytic DBMS for big
data)
     Loaded: loaded (/etc/systemd/system/clickhouse-server.service; enabled;
vendor preset: enabled)
     Active: active (running) since Wed 2020-09-16 05:18:54 UTC; 5s ago
   Main PID: 2697 (clickhouse-serv)
      Tasks: 46 (limit: 1137)
     Memory: 459.7M
     CGroup: /system.slice/clickhouse-server.service
             └─2697 /usr/bin/clickhouse-server --config=/etc/clickhouse-
server/config.xml --pid-file=/run/clickhouse-server/clickhouse-server.pid
```

The output notes that the server is running.

You have successfully started the ClickHouse server and will now be able to use the `clickhouse-client` CLI program to connect to the server.

Step 3 — Creating Databases and Tables

In ClickHouse, you can create and delete databases by executing SQL statements directly in the interactive database prompt. Statements consist of commands following a particular syntax that tell the database server to perform a requested operation along with any data required. You create databases by using the `CREATE DATABASE table_name` syntax. To create a database, first start a client session by running the following command:

```
clickhouse-client --password
```

Copy

You will be asked to enter the password you had set during the installation—enter it to successfully to start the client session.

The previous command will log you in to the client prompt where you can run ClickHouse SQL statements to perform actions such as:

- Creating, updating, and deleting databases, tables, indexes, partitions, and views.
- Executing queries to retrieve data that is optionally filtered and grouped using various conditions.

In this step, with the ClickHouse client ready for inserting data, you're going to create a database and table. For the purposes of this tutorial, you'll create a database named `test`, and inside that you'll create a table named `visits` that tracks website-visit durations.

Now that you're inside the ClickHouse command prompt, create your `test` database by executing:

```
CREATE DATABASE test;
```

Copy

You'll see the following output that shows that you have created the database:

```
OutputCREATE DATABASE test

Ok.

0 rows in set. Elapsed: 0.003 sec.
```

A ClickHouse table is similar to tables in other relational databases; it holds a collection of related data in a structured format. You can specify columns along with their types, add rows of data, and execute different kinds of queries on tables.

The syntax for creating tables in ClickHouse follows this example structure:

```
CREATE TABLE table_name
(
    column_name1 column_type [options],
    column_name2 column_type [options],
    ...
) ENGINE = engine
```

The `table_name` and `column_name` values can be any valid ASCII identifiers. ClickHouse supports a wide range of column types; some of the most popular are:

- `UInt64`: used for storing integer values in the range 0 to 18446744073709551615.
- `Float64`: used for storing floating point numbers such as 2039.23, 10.5, etc.
- `String`: used for storing variable length characters. It does not require a max-length attribute since it can store arbitrary lengths.
- `Date`: used for storing dates that follow the `YYYY-MM-DD` format.
- `DateTime`: used for storing dates coupled with time and follows the `YYYY-MM-DD HH:MM:SS` format.

After the column definitions, you specify the engine used for the table. In ClickHouse, *Engines* determine the physical structure of the underlying data, the table's querying capabilities, its concurrent access modes, and support for indexes. Different engine types are suitable for different application requirements. The most commonly used and widely applicable engine type is `MergeTree`.

Now that you have an overview of table creation, you'll create a table. Start by confirming the database you'll be modifying:

```
USE test;
```

Copy

You will see the following output showing that you have switched to the `test` database from the `default` database:

```
OutputUSE test

Ok.

0 rows in set. Elapsed: 0.001 sec.
```

The remainder of this guide will assume that you are executing statements within this database's context.

Create your `visits` table by running this command:

```
CREATE TABLE visits (
 id UInt64,
 duration Float64,
 url String,
 created DateTime
) ENGINE = MergeTree()
PRIMARY KEY id
ORDER BY id;
```

Copy

Here's a breakdown of what the command does. You create a table named `visits` that has four columns:

- `id`: The primary key column. Similarly to other RDBMS systems, a primary key column in ClickHouse uniquely identifies a row; each row should have a unique value for this column.
- `duration`: A float column used to store the duration of each visit in seconds. `float` columns can store decimal values such as 12.50.
- `url`: A string column that stores the URL visited, such as `http://example.com`.
- `created`: A date and time column that tracks when the visit occurred.

After the column definitions, you specify `MergeTree` as the storage engine for the table. The MergeTree family of engines is recommended for production databases due to its optimized support for large real-time inserts, overall robustness, and query support. Additionally, MergeTree engines support sorting of rows by primary key, partitioning of rows, and replicating and sampling data.

If you intend to use ClickHouse for archiving data that is not queried often or for storing temporary data, you can use the Log family of engines to optimize for that use-case.

After the column definitions, you'll define other table-level options. The `PRIMARY KEY` clause sets `id` as the primary key column and the `ORDER BY` clause will store values sorted by the `id` column. A primary key uniquely identifies a row and is used for efficiently accessing a single row and efficient colocation of rows.

On executing the create statement, you will see the following output:

```
OutputCREATE TABLE visits
(
    id UInt64,
    duration Float64,
    url String,
    created DateTime
)
ENGINE = MergeTree()
PRIMARY KEY id
ORDER BY id

Ok.
```

```
0 rows in set. Elapsed: 0.010 sec.
```

In this section, you've created a database and a table to track website-visit data. In the next step, you'll insert data into the table, update existing data, and delete that data.

Step 4 — Inserting, Updating, and Deleting Data and Columns

In this step, you'll use your `visits` table to insert, update, and delete data. The following command is an example of the syntax for inserting rows into a ClickHouse table:

```
INSERT INTO table_name VALUES (column_1_value, column_2_value, ....);
```

Now, insert a few rows of example website-visit data into your `visits` table by running each of the following statements:

```
INSERT INTO visits VALUES (1, 10.5, 'http://example.com', '2019-01-01 00:01:01');
```

Copy

```
INSERT INTO visits VALUES (2, 40.2, 'http://example1.com', '2019-01-03
10:01:01');
```

Copy

```
INSERT INTO visits VALUES (3, 13, 'http://example2.com', '2019-01-03 12:01:01');
```

Copy

```
INSERT INTO visits VALUES (4, 2, 'http://example3.com', '2019-01-04 02:01:01');
```

Copy

You'll see the following output repeated for each insert statement.

```
OutputINSERT INTO visits VALUES

Ok.

1 rows in set. Elapsed: 0.004 sec.
```

The output for each row shows that you've inserted it successfully into the `visits` table.

Now you'll add an additional column to the `visits` table. When adding or deleting columns from existing tables, ClickHouse supports the `ALTER` syntax.

For example, the basic syntax for adding a column to a table is as follows:

```
ALTER TABLE table_name ADD COLUMN column_name column_type;
```

Add a column named `location` that will store the location of the visits to a website by running the following statement:

```
ALTER TABLE visits ADD COLUMN location String;
```

Copy

You'll see output similar to the following:

```
OutputALTER TABLE visits
    ADD COLUMN
    location String


Ok.

0 rows in set. Elapsed: 0.014 sec.
```

The output shows that you have added the `location` column successfully.

As of version 19.13.3, ClickHouse doesn't support updating and deleting individual rows of data due to implementation constraints. ClickHouse has support for bulk updates and deletes, however, and has a distinct SQL syntax for these operations to highlight their non-standard usage.

The following syntax is an example for bulk updating rows:

```
ALTER TABLE table_name UPDATE  column_1 = value_1, column_2 = value_2 ...  WHERE
filter_conditions;
```

You'll run the following statement to update the `url` column of all rows that have a `duration` of less than 15. Enter it into the database prompt to execute:

```
ALTER TABLE visits UPDATE url = 'http://example2.com' WHERE duration < 15;
```

Copy

The output of the bulk update statement will be as follows:

```
OutputALTER TABLE visits
    UPDATE url = 'http://example2.com' WHERE duration < 15


Ok.

0 rows in set. Elapsed: 0.003 sec.
```

The output shows that your update query completed successfully. The `0 rows in set` in the output denotes that the query did not return any rows; this will be the case for any update and delete queries.

The example syntax for bulk deleting rows is similar to updating rows and has the following structure:

```
ALTER TABLE table_name DELETE WHERE filter_conditions;
```

To test deleting data, run the following statement to remove all rows that have a `duration` of less than `5`:

```
ALTER TABLE visits DELETE WHERE duration < 5;
```

Copy

The output of the bulk delete statement will be similar to:

```
OutputALTER TABLE visits
    DELETE WHERE duration < 5


Ok.

0 rows in set. Elapsed: 0.003 sec.
```

The output confirms that you have deleted the rows with a duration of less than five seconds.

To delete columns from your table, the syntax would follow this example structure:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Delete the `location` column you added previously by running the following:

```
ALTER TABLE visits DROP COLUMN location;
```

Copy

The `DROP COLUMN` output confirming that you have deleted the column will be as follows:

```
OutputALTER TABLE visits
    DROP COLUMN
    location String


Ok.

0 rows in set. Elapsed: 0.010 sec.
```

Now that you've successfully inserted, updated, and deleted rows and columns in your `visits` table, you'll move on to query data in the next step.

Step 5 — Querying Data

ClickHouse's query language is a custom dialect of SQL with extensions and functions suited for analytics workloads. In this step, you'll run selection and aggregation queries to retrieve data and results from your `visits` table.

Selection queries allow you to retrieve rows and columns of data filtered by conditions that you specify, along with options such as the number of rows to return. You can select rows and columns of data using the `SELECT` syntax. The basic syntax for `SELECT` queries is:

```
SELECT func_1(column_1), func_2(column_2) FROM table_name WHERE filter_conditions
row_options;
```

Execute the following statement to retrieve `url` and `duration` values for rows where the `url` is `http://example.com`.

```
SELECT url, duration FROM visits WHERE url = 'http://example2.com' LIMIT 2;
```

Copy

You will see the following output:

```
OutputSELECT
    url,
    duration
FROM visits
WHERE url = 'http://example2.com'
LIMIT 2

┌─url─────────────────┬─duration─┐
| http://example2.com |     10.5 |
└─────────────────────┴──────────┘
┌─url─────────────────┬─duration─┐
| http://example2.com |       13 |
└─────────────────────┴──────────┘

2 rows in set. Elapsed: 0.013 sec.
```

The output has returned two rows that match the conditions you specified. Now that you've selected values, you can move on to executing *aggregation queries*.

Aggregation queries are queries that operate on a set of values and return single output values. In analytics databases, these queries are run frequently and are well optimized by the database. Some aggregate functions supported by ClickHouse are:

- `count`: returns the count of rows matching the conditions specified.
- `sum`: returns the sum of selected column values.
- `avg`: returns the average of selected column values.

Some ClickHouse-specific aggregate functions include:

- `uniq`: returns an approximate number of distinct rows matched.
- `topK`: returns an array of the most frequent values of a specific column using an approximation algorithm.

To demonstrate the execution of aggregation queries, you'll calculate the total duration of visits by running the `sum` query:

```
SELECT SUM(duration) FROM visits;
```

Copy

You will see output similar to the following:

```
OutputSELECT SUM(duration)
FROM visits

┌─SUM(duration)─┐
│          63.7 │
└───────────────┘

1 rows in set. Elapsed: 0.010 sec.
```

Now, calculate the top two URLs by executing:

```
SELECT topK(2)(url) FROM visits;
```

Copy

You will see output similar to the following:

```
OutputSELECT topK(2)(url)
FROM visits

┌─topK(2)(url)──────────────────────────────────────┐
│ ['http://example2.com','http://example1.com'] │
└───────────────────────────────────────────────────┘

1 rows in set. Elapsed: 0.010 sec.
```

Now that you have successfully queried your `visits` table, you'll delete tables and databases in the next step.

Step 6 — Deleting Tables and Databases

In this section, you'll delete your `visits` table and `test` database.

The syntax for deleting tables follows this example:

```
DROP TABLE table_name;
```

To delete the `visits` table, run the following statement:

```
DROP TABLE visits;
```

Copy

You will see the following output declaring that you've deleted the table successfully:

```
outputDROP TABLE visits

Ok.

0 rows in set. Elapsed: 0.005 sec.
```

You can delete databases using the `DROP database table_name` syntax. To delete the `test` database, execute the following statement:

```
DROP DATABASE test;
```

Copy

The resulting output shows that you've deleted the database successfully.

```
OutputDROP DATABASE test

Ok.

0 rows in set. Elapsed: 0.003 sec.
```

You've deleted tables and databases in this step. Now that you've created, updated, and deleted databases, tables, and data in your ClickHouse instance, you'll enable remote access to your database server in the next section.

Step 7 — Setting Up Firewall Rules (Optional)

If you intend to only use ClickHouse locally with applications running on the same server, or do not have a firewall enabled on your server, you don't need to complete this section. If instead, you'll be connecting to the ClickHouse database server remotely, you should follow this step.

Currently your server has a firewall enabled that disables your public IP address accessing all ports. You'll complete the following two steps to allow remote access:

- Modify ClickHouse's configuration and allow it to listen on all interfaces.
- Add a firewall rule allowing incoming connections to port `8123`, which is the HTTP port that the ClickHouse server runs.

If you are inside the database prompt, exit it by typing `CTRL+D`.

Edit the configuration file by executing:

```
sudo nano /etc/clickhouse-server/config.xml
```

Copy

Then uncomment the line containing `<!-- <listen_host>0.0.0.0</listen_host> -->`, like the following file:

/etc/clickhouse-server/config.xml

```
...
 <interserver_http_host>example.yandex.ru</interserver_http_host>
```

```
    -->

    <!-- Listen specified host. use :: (wildcard IPv6 address), if you want to
accept connections both with IPv4 and IPv6 from everywhere. -->
    <!-- <listen_host>::</listen_host> -->
    <!-- Same for hosts with disabled ipv6: -->
    <listen_host>0.0.0.0</listen_host>

    <!-- Default values - try listen localhost on ipv4 and ipv6: -->
    <!--
    <listen_host>::1</listen_host>
    <listen_host>127.0.0.1</listen_host>
    -->
...
```

Save the file and exit. For the new configuration to apply restart the service by running:

```
sudo service clickhouse-server restart
```

Copy

You won't see any output from this command. ClickHouse's server listens on port `8123` for HTTP connections and port `9000` for connections from `clickhouse-client`. Allow access to both ports for your second server's IP address with the following command:

```
sudo ufw allow from second_server_ip/32 to any port 8123
```

Copy

```
sudo ufw allow from second_server_ip/32 to any port 9000
```

Copy

You will see the following output for both commands that shows that you've enabled access to both ports:

```
OutputRule added
```

ClickHouse will now be accessible from the IP that you added. Feel free to add additional IPs such as your local machine's address if required.

To verify that you can connect to the ClickHouse server from the remote machine, first follow the steps in Step 1 of this tutorial on the second server and ensure that you have the `clickhouse-client` installed on it.

Now that you have logged in to the second server, start a client session by executing:

```
clickhouse-client --host your_server_ip --password
```

Copy

You will see the following output that shows that you have connected successfully to the server:

```
OutputClickHouse client version 19.13.3.26 (official build).
Password for user (default):
Connecting to your_server_ip:9000 as user default.
Connected to ClickHouse server version 19.13.3 revision 54425.

hostname :)
```

In this step, you've enabled remote access to your ClickHouse database server by adjusting your firewall rules.

Conclusion

You have successfully set up a ClickHouse database instance on your server and created a database and table, added data, performed queries, and deleted the database. Within ClickHouse's documentation you can read about their benchmarks against other open-source and commercial analytics databases and general reference documents.

Further features ClickHouse offers include distributed query processing across multiple servers to improve performance and protect against data loss by storing data over different shards.