EDB Postgres AI for ClickHouse

Version 26.3

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

Built at 2026-06-22T16:19:39

1	EDB Postgres AI for ClickHouse	3
2	Release notes	4
2.1	EDB Postgres AI for ClickHouse 26.3.13.31 release notes	5
3	Overview	6
3.1	ClickHouse architecture	8
3.2	EDB Postgres AI for ClickHouse compatibility	11
3.3	Known issues	12
4	Installing and configuring EDB Postgres AI for ClickHouse	13
4.1	Installing EDB Postgres AI for ClickHouse on Rocky Linux	14
4.2	Installing EDB Postgres AI for ClickHouse with Docker	22
4.3	Installing EDB Postgres AI for ClickHouse on Kubernetes	33
5	Getting started with ClickHouse	37
6	Connecting ClickHouse to WarehousePG	38
7	Integrating with Iceberg catalogs	42

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

1 EDB Postgres AI for ClickHouse

ClickHouse is a columnar database engine for real-time analytics. It ingests millions of events per second and makes them queryable immediately, delivering sub-second aggregations across billions of rows. It's the right engine for workloads that demand sub-second answers on live, high-volume data.

EDB Postgres AI for ClickHouse delivers EDB-signed packages, SLAs, CVE response, and break/fix expertise for open-source ClickHouse. Not a fork. Not a managed service. Run your own ClickHouse deployment with enterprise support from EDB.

EDB Postgres AI for ClickHouse is based entirely on open-source ClickHouse. For a full breakdown of what's covered and what requires a separate commercial offering, see the overview page.

What ClickHouse is for

ClickHouse is optimized for workloads where speed beats everything else:

- High-volume analytics where data arrives seconds or minutes ago and must be queryable immediately
- IoT monitoring, ad-tech bidding, fraud detection, and similar time-critical workloads
- User-facing dashboards with thousands of concurrent users on live data
- Sub-second rollups on events, logs, and telemetry
- Postgres observability, storing and querying metrics from Postgres databases at scale

ClickHouse and WarehousePG

ClickHouse and WarehousePG answer different questions and complement each other within the EDB Postgres AI Analytics pillar.

ClickHouse answers "What's happening right now?": sub-second aggregations on high-volume, recent data, optimized for few-column scans across huge single tables.

WarehousePG answers "What happened?": complex multi-table joins, enrichment, data modeling, and batch analytics where accuracy matters more than speed.

Apache Iceberg® acts as the shared source of truth between them: data written by either engine is accessible to both, with no copies or ETL pipelines between them.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

2 Release notes

The EDB Postgres AI for ClickHouse documentation describes the latest version of EDB Postgres AI for ClickHouse, including minor releases and patches. These release notes cover what was new in each release.

Version	Release date
26.3.13.31	24 June 2026

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

2.1 EDB Postgres AI for ClickHouse 26.3.13.31 release notes

Released: 24 June 2026

EDB Postgres AI for ClickHouse 26.3.13.31 is the initial release. New features, enhancements, bug fixes, and other changes include:

Type	Description
Upstream merge	Based on open-source ClickHouse 26.3.13.31 LTS. See the ClickHouse 26.3 changelog for details of upstream changes.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

3 Overview

Understand how EDB packages and distributes ClickHouse, what's covered in this documentation, and what falls outside its scope.

How EDB provides ClickHouse

EDB Postgres AI for ClickHouse isn't a fork. EDB builds and distributes the open-source ClickHouse binary, identical to the upstream release, repackaged as RPMs, container images, and a Kubernetes operator for enterprise Linux distributions, and supported by EDB.

EDB's contribution is:

- Building the official open-source ClickHouse source at a pinned upstream release
- Packaging the binary as RPMs, a container image, and a Kubernetes operator, distributed through the EDB package repository
- Integrating ClickHouse into the EDB Postgres AI Analytics pillar alongside WarehousePG and the EDB Postgres Lakehouse
- Providing EDB support

The ClickHouse query engine, SQL dialect, table engines, and performance characteristics are identical to the upstream open-source release.

Verify you're running an EDB-packaged build:

```
SELECT value FROM system.build_options WHERE name = 'VERSION_OFFICIAL';
```

The output includes `(EDB Build)` for EDB-packaged releases.

What's included

EDB packages and supports the open-source ClickHouse binary. The following capabilities are available in the EDB distribution.

Area	What's covered
Table engines	MergeTree family, integration engines (`PostgreSQL` , `S3` , `Kafka` , `IcebergS3` , `DeltaLake` , `MongoDB` , `Redis` , and more), and special engines (`Distributed` , `MaterializedView` , `Dictionary` , and more)
Database engines	`Atomic` , `Replicated` , `PostgreSQL` , `MySQL` , `SQLite` , `DataLakeCatalog`
SQL and query capabilities	Aggregations, window functions, CTEs, subqueries, materialized views, all major data formats (Parquet, JSON, CSV, Avro, ORC, Arrow), dictionaries, projections, and RBAC
Data lake integration	Read access to Apache Iceberg® tables via `DataLakeCatalog` (REST, AWS Glue, Databricks Unity Catalog, Hive, OneLake) and directly via `IcebergS3` , `IcebergAzureBlobStorage` , and related engines; Delta Lake and Apache Hudi via dedicated table engines
Operations	ClickHouse Keeper, `clickhouse-client` , `clickhouse-local` , and Named Collections

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

What's not covered

The following features are exclusive to ClickHouse® Cloud, a separate commercial service from the upstream ClickHouse project. They're not part of the open-source binary and aren't covered in this documentation.

Feature	Description
ClickPipes	Managed ingestion pipelines for Kafka, S3, Postgres CDC, MySQL, MongoDB, and others
SharedMergeTree	Proprietary table engine for ClickHouse® Cloud's shared-storage architecture
SQL Console	Web-based query editor with dashboards, visualizations, and GenAI SQL assistance
Query Insights	Managed query log analytics built into the Cloud console
Managed Postgres	Fully managed Postgres service collocated with ClickHouse
Compute-compute separation	Architecture for scaling compute independently of storage
Auto-scaling	Automatic vertical and horizontal scaling
Managed backups	Automated, configurable backups with external export
BYOC	Bring Your Own Cloud, a managed service deployed in your AWS account
Compliance certifications	HIPAA, SOC2, PCI DSS, ISO 27001 (managed by the Cloud service)
SSO / CMEK	Single sign-on and customer-managed encryption keys via Cloud console

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

3.1 ClickHouse architecture

EDB Postgres AI for ClickHouse runs as a standalone instance or as a distributed cluster, with support for sharding, replication, or both.

The ClickHouse server is the database engine that stores and processes data. The ClickHouse client is the command-line interface for connecting to a server and running queries.

Deployment topologies

Your topology determines how data is stored, replicated, and served, as well as how much operational complexity you take on.

- **Single-node**: ClickHouse runs as a standalone instance on a single host. No cluster coordination is needed. Suitable for development, testing, and single-tenant analytical workloads.
- **Multi-node cluster**: Data is distributed across shards for horizontal scale and replicated for high availability. A multi-node cluster can use sharding alone, replication alone, or both together. Replication requires ClickHouse Keeper.

Table engines

In ClickHouse, every table has an engine that determines how data is stored, indexed, and replicated. The primary family is `MergeTree`, designed for high-throughput analytical workloads. `ReplicatedMergeTree` extends it with built-in replication across nodes. For distributed queries, the `Distributed` engine acts as a virtual table that routes queries to the underlying shards and merges the results. For a full reference, seeTable engines in the ClickHouse documentation.

Sharding

ClickHouse uses a shared-nothing architecture: each node has its own storage and operates independently. Sharding takes advantage of this by splitting your data horizontally across multiple nodes, each holding a different subset of rows. Queries run in parallel across shards and their results are merged before being returned to the client. This approach scales write throughput and storage beyond what a single node can handle, but doesn't provide high availability. If a shard goes down, its data is unavailable.

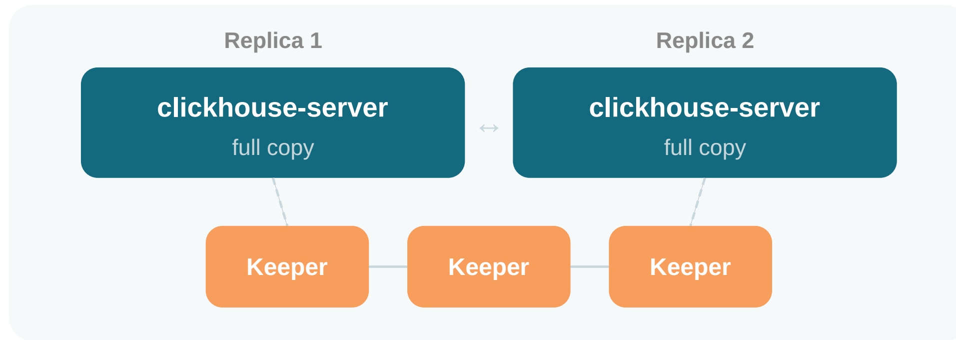
Sharding doesn't require cluster coordination. Query routing is handled by the `Distributed` table engine, which sends queries to all shards in parallel and merges the results transparently. See How ClickHouse routes queries for details.

Replication

Replication keeps identical copies of the same data on multiple nodes. If one node goes down, another replica can serve queries. Replication provides high availability but doesn't increase storage capacity or write throughput on its own.

To enable replication in ClickHouse, choose `ReplicatedMergeTree` as your table engine and deploy ClickHouse Keeper. Keeper coordinates replicas by tracking which data parts exist and keeping copies in sync.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.



ClickHouse Keeper

ClickHouse Keeper is ClickHouse's built-in coordination service, equivalent in role to Apache ZooKeeper and compatible with it at the protocol level. It handles:

- Tracking which data parts exist across replicas
- Coordinating inserts so replicas stay in sync
- Executing distributed DDL (schema changes applied across all nodes)
- Electing leaders for replicated operations

Keeper uses the Raft consensus algorithm and requires a quorum of nodes to function. The minimum is three Keeper nodes, which allows the cluster to tolerate one node failure. With five Keeper nodes it can tolerate two failures.

Keeper can run in two ways. In a co-located deployment, Keeper runs alongside the ClickHouse server on the same nodes. Co-location is simpler to manage and works well for smaller clusters. In a dedicated deployment, Keeper runs on separate nodes with no server workload, which gives better isolation and is recommended for larger or production clusters.

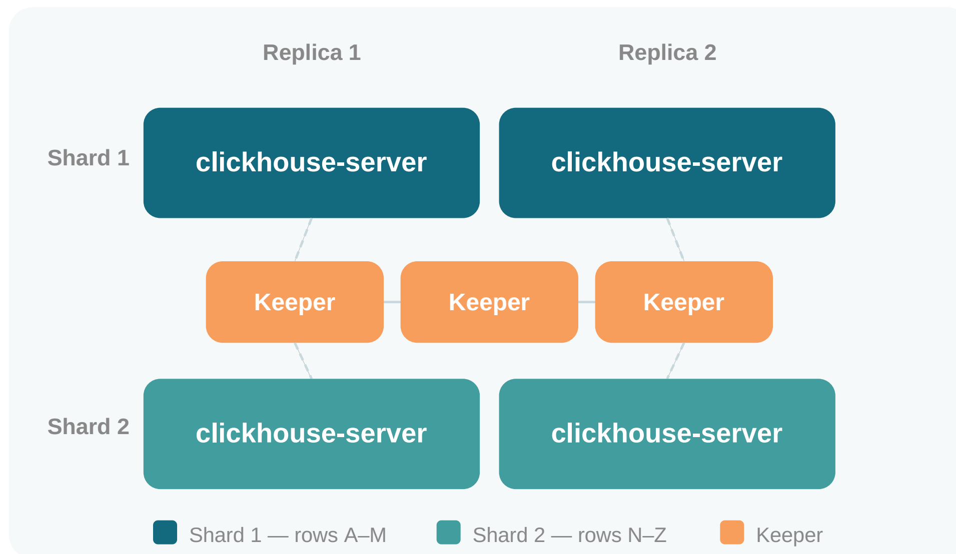
Combining sharding and replication

Combine sharding and replication to achieve both horizontal scale and high availability. A common production topology is two shards with two replicas each, giving four server nodes total:

- Two shards provide horizontal scale across two independent data sets.
- Two replicas per shard provide high availability: each shard has a standby copy.

Keeper runs on at least three nodes, either co-located on server nodes or on separate dedicated nodes. See Deployment options for guidance on the two Keeper deployment approaches.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.



How ClickHouse routes queries

When data is sharded across multiple nodes, queries need to reach all shards and have their results combined. ClickHouse handles this routing with the `Distributed` table engine.

A `Distributed` table is a virtual table that sits on top of the underlying `MergeTree` or `ReplicatedMergeTree` tables on each shard. When you query a `Distributed` table on any node, that node fans the query out to all shards in parallel, collects the results, and returns the merged result.

Your application connects to any server node and queries the `Distributed` table without needing to know the sharding topology. For writes, inserting into a `Distributed` table routes each row to the correct shard using a sharding key you define, typically a hash of a column.

Sharding, replication, and Keeper topology are all defined in the ClickHouse server configuration file. For details, see Configuration files in the ClickHouse documentation.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

3.2 EDB Postgres AI for ClickHouse compatibility

Platform compatibility

Rocky Linux 10 on x86-64.

System requirements

Minimum recommended hardware per node:

	Single-node	Cluster node
CPU	4 cores	4 cores
RAM	16 GB	16 GB
Storage	100 GB (NVMe preferred)	100 GB (NVMe preferred)

For production sizing guidance, see the ClickHouse sizing and hardware recommendations.

WarehousePG

- WarehousePG 6.x or later.

Apache Iceberg®

Iceberg format version	Read	Write
v1	Yes	No
v2 (position deletes)	Yes	No
v2 (equality deletes)	Yes	No
v3 (deletion vectors)	No	No

Write support for Iceberg isn't yet available. See Known issues.

Object storage

Provider	Table engines
Amazon S3	`IcebergS3` , `DeltaLake` , `S3`
Azure Blob Storage	`IcebergAzureBlobStorage` , `AzureBlobStorage`
HDFS	`IcebergHDFS` , `HDFS`
Local filesystem	`IcebergLocal` , `File`

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

3.3 Known issues

This release includes the following known issues and limitations. Where applicable, workarounds are included to help mitigate the impact. These issues are actively tracked and are planned for resolution in a future release.

- The `PostgreSQL` table engine connects only to the WarehousePG coordinator and doesn't engage WarehousePG's massively parallel processing (MPP) architecture. For bulk data movement, use `INSERT INTO clickhouse_table SELECT * FROM warehousepg_linked_table` to load data into a native ClickHouse `MergeTree` table.

- The Iceberg catalog integration is read-only. `INSERT`, `UPDATE`, and `DELETE` operations aren't supported. Use Spark, PyIceberg, or another Iceberg-compatible writer to create and populate Iceberg tables.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

4 Installing and configuring EDB Postgres AI for ClickHouse

Plan your deployment and install EDB Postgres AI for ClickHouse on your infrastructure.

Deployment options

Your topology determines how data is stored, replicated, and served, as well as how much operational complexity you take on. For a full explanation of sharding, replication, and ClickHouse Keeper, see Architecture.

Single-node

ClickHouse runs as a standalone instance on a single host. No cluster coordination is needed. Suitable for development, testing, and single-tenant analytical workloads.

Multi-node cluster

Data is distributed across shards for horizontal scale and replicated for high availability. Replication requires ClickHouse Keeper running on at least three nodes. Keeper can run co-located on server nodes or on separate dedicated nodes.

Choosing an installation method

EDB provides three ways to install ClickHouse:

- Rocky Linux
- Docker
- Kubernetes operator

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

4.1 Installing EDB Postgres AI for ClickHouse on Rocky Linux

EDB distributes ClickHouse as RPM packages for Rocky Linux 10 on x86-64. The following packages are available:

Package	Description	Install on
`edb-clickhouse-server`	ClickHouse server engine, systemd services, and default configuration files. Includes `clickhouse-keeper.service`, so server nodes can run Keeper alongside the server without an additional package.	Server nodes
`edb-clickhouse-client`	`clickhouse-client`, `clickhouse-local`, `clickhouse-benchmark`, and related tools.	Server nodes, remote client hosts
`edb-clickhouse-common-static`	Compiled ClickHouse binary, required by the server and client packages.	All server and client nodes
`edb-clickhouse-keeper`	Standalone Keeper binary for dedicated coordination nodes. Only needed for multi-node deployments that run Keeper on separate nodes.	Dedicated Keeper nodes only

For an explanation of node roles and deployment topologies, see Architecture.

Prerequisites

- Rocky Linux 10
- `sudo` or root access
- An EDB subscription token, available from the EDB customer portal

Deploying a single node

ClickHouse runs as a standalone instance with no cluster coordination required. Suitable for development, testing, and single-tenant analytical workloads.

1. Install the packages:

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=clickhouse
curl -1sSLf
"https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_SUBSCRIPTION_PLAN/setup.rpm.sh" |
sudo -E bash
sudo dnf install -y
\
  edb-clickhouse-common-static \
  edb-clickhouse-server \
  edb-clickhouse-client
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

2. Start the server:

```
sudo systemctl enable clickhouse-
server
sudo systemctl start clickhouse-
server
```

Verify the server is running:

```
sudo systemctl status clickhouse-
server
```

3. Connect using the ClickHouse command-line client:

```
clickhouse-client
```

By default, the `default` user has no password. To set one, run the following after connecting:

```
ALTER USER default IDENTIFIED BY 'yourpassword';
```

Then connect with:

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

4. Verify the EDB build by checking the version string:

```
SELECT value FROM system.build_options WHERE name = 'VERSION_OFFICIAL';
```

The output includes `(EDB Build)` for EDB-packaged releases.

Deploying a cluster

A cluster deployment distributes data across multiple server nodes using sharding, replication, or both. Replication requires ClickHouse Keeper running on at least three nodes. For an explanation of sharding, replication, and Keeper roles, see Architecture.

The following steps walk through a five-node deployment: four server nodes across two shards with two replicas each, with Keeper co-located on node-1 and node-2 and running as a dedicated service on node-5. Adjust the configuration for your own topology.

Node	Role	Shard	Replica
node-1	ClickHouse server + Keeper	1	1
node-2	ClickHouse server + Keeper	1	2
node-3	ClickHouse server	2	1
node-4	ClickHouse server	2	2
node-5	Keeper		

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

Installing packages

1. Set up the EDB repository on every node:

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=clickhouse
curl -1sSLf
"https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_SUBSCRIPTION_PLAN/setup.rpm.sh" |
sudo -E bash
```

2. Install the role-appropriate packages on each node.

   On each server node:

```
sudo dnf install -y
\
  edb-clickhouse-common-static \
  edb-clickhouse-server \
  edb-clickhouse-client
```

   For co-located deployments, `edb-clickhouse-server` already includes `clickhouse-keeper.service`, so no additional package is needed.

   On each dedicated Keeper node:

```
sudo dnf install -y edb-clickhouse-
keeper
```

Configuring Keeper

Configure all Keeper nodes before starting any service. Use a minimum of three Keeper nodes to tolerate one node failure.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

1. The RPM installs a default `/etc/clickhouse-keeper/keeper_config.xml`. Replace its contents on each Keeper node with the following, setting `<server_id>` to a unique integer (`1`, `2`, `3`) per node:

```
<clickhouse>

<listen_host>0.0.0.0</listen_host>

<keeper_server>

<tcp_port>9181</tcp_port>
        <server_id>1</server_id><!-- Set to 1, 2, or 3 depending on the node --
>

<log_storage_path>/var/lib/clickhouse/coordination/log</log_storage_path>

<snapshot_storage_path>/var/lib/clickhouse/coordination/snapshots</snapshot_storage_path>

<coordination_settings>

<operation_timeout_ms>10000</operation_timeout_ms>

<session_timeout_ms>30000</session_timeout_ms>

<raft_logs_level>information</raft_logs_level>
        </coordination_settings>

<raft_configuration>
            <server><id>1</id><hostname><node-1-ip></hostname><port>9234</port>
</server>
            <server><id>2</id><hostname><node-2-ip></hostname><port>9234</port>
</server>
            <server><id>3</id><hostname><node-5-ip></hostname><port>9234</port>
</server>
        </raft_configuration>
    </keeper_server>
</clickhouse>
```

Configuring the servers

Each server node needs a cluster configuration that defines the remote servers, Keeper endpoints, and the node's own shard and replica identity.

1. Create `/etc/clickhouse-server/config.d/cluster.xml` on each server node, adjusting the `<shard>` and `<replica>` values in the `<macros>` section for each node:

Node	`<shard>`	`<replica>`
node-1	01	01
node-2	01	02
node-3	02	01
node-4	02	02

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```
<clickhouse>

<listen_host>0.0.0.0</listen_host>

<remote_servers>

<my_cluster>

<shard>

<internal_replication>true</internal_replication>
                <replica><host><node-1-ip></host><port>9000</port>
</replica>
                <replica><host><node-2-ip></host><port>9000</port>
</replica>
            </shard>

<shard>

<internal_replication>true</internal_replication>
                <replica><host><node-3-ip></host><port>9000</port>
</replica>
                <replica><host><node-4-ip></host><port>9000</port>
</replica>
            </shard>
        </my_cluster>
    </remote_servers>

<zookeeper>
        <node><host><node-1-ip></host><port>9181</port>
</node>
        <node><host><node-2-ip></host><port>9181</port>
</node>
        <node><host><node-5-ip></host><port>9181</port>
</node>
    </zookeeper>

<macros>

<cluster>my_cluster</cluster>
        <shard>01</shard><!-- Adjust per node, see table above --
>
        <replica>01</replica><!-- Adjust per node, see table above --
>
    </macros>

<distributed_ddl>

<path>/clickhouse/task_queue/ddl</path>
    </distributed_ddl>
</clickhouse>
```

Starting services

1. Start Keeper on all Keeper nodes. On each Keeper node:

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```
sudo systemctl enable clickhouse-
keeper
sudo systemctl start clickhouse-
keeper
```

2. Verify each Keeper node is healthy. Run the following on each Keeper node. A healthy node responds with `imok`:

```
clickhouse-keeper-client -h 127.0.0.1 -p 9181 -q "ruok"
```

output

```
imok
```

3. Confirm quorum is formed from any one Keeper node. A value of `2` means two followers are synced to the leader, confirming all three Keeper nodes have formed a quorum:

```
clickhouse-keeper-client -h 127.0.0.1 -p 9181 -q "mntr" | grep
zk_synced_followers
```

output

```
zk_synced_followers 2
```

4. Start the server on all server nodes:

```
sudo systemctl enable clickhouse-
server
sudo systemctl start clickhouse-
server
```

5. Verify the server is running on each server node:

```
sudo systemctl status clickhouse-
server
```

Connecting to the cluster

1. Connect to any server node from a host with `edb-clickhouse-client` installed:

```
clickhouse-client -h <server-node-ip>
```

By default, the `default` user has no password. To set one, run the following after connecting:

```
ALTER USER default IDENTIFIED BY 'yourpassword';
```

Then connect with:

```
clickhouse-client -h <server-node-ip> --
password
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

2. Confirm all server nodes are registered. `system.clusters` lists server nodes only, so the dedicated Keeper node-5 doesn't appear:

```
SELECT cluster, shard_num, replica_num, host_name
FROM system.clusters
WHERE cluster = 'my_cluster';
```

```
output
┌─cluster────┬─shard_num─┬─replica_num─┬─host_name───┐
│ my_cluster │         1 │           1 │ <node-1-ip> │
│ my_cluster │         1 │           2 │ <node-2-ip> │
│ my_cluster │         2 │           1 │ <node-3-ip> │
│ my_cluster │         2 │           2 │ <node-4-ip> │
└────────────┴───────────┴─────────────┴─────────────┘
```

Testing replication and sharding

Create a local `ReplicatedMergeTree` table and a `Distributed` table on top of it. Insert some rows and query both tables to confirm that replication is working within each shard and that the distributed table aggregates data across all shards.

1. Create a local storage table. The `ReplicatedMergeTree` engine stores data physically on each node and replicates it within the shard:

```
CREATE TABLE events ON CLUSTER my_cluster
(
    event_id
UInt32,
    user_id    UInt32,
    event_type String,
    created_at
DateTime
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/events',
'{replica}')
ORDER BY event_id;
```

2. Create a `Distributed` table on top of `events`. Unlike the local table, it stores no data. It routes queries to `events` across all shards and aggregates the results:

```
CREATE TABLE events_dist ON CLUSTER my_cluster
(
    event_id
UInt32,
    user_id    UInt32,
    event_type String,
    created_at
DateTime
) ENGINE = Distributed(my_cluster, default, events,
rand());
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

3. Insert some rows via `events_dist`. The `rand()` sharding key distributes each row across shards:

```
INSERT INTO events_dist
VALUES
    (1, 101, 'login',
now()),
    (2, 102, 'purchase',
now()),
    (3, 103, 'logout',
now()),
    (4, 104, 'login',
now());
```

4. Query the local table on any server node. Each node returns only the rows on its shard. The specific rows vary depending on how `rand()` distributed them:

```
SELECT * FROM events;
```

```
output
┌─event_id─┬─user_id─┬─event_type─┬──────────created_at─┐
│        2 │     102 │ purchase   │ 2026-06-10 14:48:15 │
│        3 │     103 │ logout     │ 2026-06-10 14:48:15 │
└──────────┴─────────┴────────────┴─────────────────────┘
```

5. Query `events_dist` from any node to confirm all rows are returned across shards:

```
SELECT * FROM events_dist ORDER BY
event_id;
```

```
output
┌─event_id─┬─user_id─┬─event_type─┬──────────created_at─┐
│        1 │     101 │ login      │ 2026-06-10 14:48:15 │
│        2 │     102 │ purchase   │ 2026-06-10 14:48:15 │
│        3 │     103 │ logout     │ 2026-06-10 14:48:15 │
│        4 │     104 │ login      │ 2026-06-10 14:48:15 │
└──────────┴─────────┴────────────┴─────────────────────┘
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

4.2 Installing EDB Postgres AI for ClickHouse with Docker

EDB publishes a container image built from the same RPMs as the Linux packages. The image is based on Rocky Linux 10 minimal, is available for x86-64, and includes `clickhouse-server`, `clickhouse-keeper`, and `clickhouse-client`.

Prerequisites

- Docker Engine 20.10 or later with Compose v2 (`docker compose` command)
- An EDB subscription token, available from the EDB customer portal

Pulling the image

Log in to the EDB container registry and pull the image:

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
docker login docker.enterprisedb.com \
  --username <your-edb-email> \
  --password "$EDB_SUBSCRIPTION_TOKEN"
docker pull docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
```

Running a single-node server

For development and testing purposes, run a single ClickHouse Server node.

1. Start the server:

```
docker run -d
\
  --name clickhouse-server
\
  --ulimit nofile=262144:262144 \
  -p 8123:8123 \
  -p 9000:9000 \
  docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
```

Port `8123` exposes the HTTP interface. Port `9000` exposes the native TCP interface used by `clickhouse-client` and most drivers.

To persist data across container restarts, add a volume mount:

```
docker run -d
\
  --name clickhouse-server
\
  --ulimit nofile=262144:262144 \
  -p 8123:8123 \
  -p 9000:9000 \
  -v /local/path/clickhouse-data:/var/lib/clickhouse \
  docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

2. To override default server settings, mount a configuration directory:

```
docker run -d
\
  --name clickhouse-server
\
  --ulimit nofile=262144:262144 \
  -p 8123:8123 \
  -p 9000:9000 \
  -v /local/path/config:/etc/clickhouse-server/config.d \
  docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
```

   ClickHouse merges all `.xml` files in `config.d` into the main configuration at startup.

3. Connect with the ClickHouse client:

```
docker exec -it clickhouse-server clickhouse-
client
```

4. Verify the EDB build:

```
SELECT value FROM system.build_options WHERE name = 'VERSION_OFFICIAL';
```

   The output includes `(EDB Build)` for EDB-packaged releases.

Running a multi-node cluster

Use Docker Compose to deploy a multi-node cluster. The EDB-provided server image includes both `clickhouse-server` and `clickhouse-keeper`, so all nodes in the cluster run from a single image.

The following example sets up a 2-shard, 2-replica cluster with 3 dedicated Keeper nodes (7 containers total). For an explanation of sharding, replication, and Keeper roles, see Architecture.

Creating the configuration files

Create the following directory structure:

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```
cluster/
├── docker-compose.yml
├── keeper-01/
│   └── keeper_config.xml
├── keeper-02/
│   └── keeper_config.xml
├── keeper-03/
│   └── keeper_config.xml
├── server-01/
│   ├── config.xml
│   └── users.xml
├── server-02/
│   ├── config.xml
│   └── users.xml
├── server-03/
│   ├── config.xml
│   └── users.xml
└── server-04/
    ├── config.xml
    └── users.xml
```

Populate each directory with the configuration files for its node role:

Keeper nodes

The three Keeper nodes form a Raft quorum. Each has a unique `server_id`. Create `keeper_config.xml` in each keeper directory, setting `server_id` to `1`, `2`, and `3` respectively.

keeper_config.xml

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```
<clickhouse
replace="true">

<logger>

<level>information</level>

<console>1</console>
    </logger>

<listen_host>0.0.0.0</listen_host>

<keeper_server>

<tcp_port>9181</tcp_port>

<server_id>1</server_id>

<log_storage_path>/var/lib/clickhouse/coordination/log</log_storage_path>

<snapshot_storage_path>/var/lib/clickhouse/coordination/snapshots</snapshot_storage_path>

<coordination_settings>

<operation_timeout_ms>10000</operation_timeout_ms>

<session_timeout_ms>30000</session_timeout_ms>

<raft_logs_level>information</raft_logs_level>
        </coordination_settings>

<raft_configuration>

<server>

<id>1</id>
                <hostname>keeper-
01</hostname>

<port>9234</port>
            </server>

<server>

<id>2</id>
                <hostname>keeper-
02</hostname>

<port>9234</port>
            </server>

<server>

<id>3</id>
                <hostname>keeper-
03</hostname>

<port>9234</port>
            </server>
        </raft_configuration>
    </keeper_server>
</clickhouse>
```

Server nodes

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

Each server node needs its cluster topology, Keeper endpoints, and shard/replica macros. Create `config.xml` in each server directory. The only values that differ between nodes are the `<shard>` and `<replica>` entries in the `<macros>` section:

Node	`<shard>`	`<replica>`
server-01	01	01
server-02	02	01
server-03	01	02
server-04	02	02

config.xml

```
<clickhouse
replace="true">

<logger>

<level>information</level>

<console>1</console>
    </logger>
    <display_name>cluster node
01</display_name>

<listen_host>0.0.0.0</listen_host>

<http_port>8123</http_port>

<tcp_port>9000</tcp_port>

<distributed_ddl>

<path>/clickhouse/task_queue/ddl</path>
    </distributed_ddl>

<remote_servers>

<my_cluster>

<shard>

<internal_replication>true</internal_replication>

<replica>
                    <host>server-
01</host>

<port>9000</port>
                </replica>

<replica>
                    <host>server-
03</host>

<port>9000</port>
                </replica>
            </shard>

<shard>

<internal_replication>true</internal_replication>

<replica>
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```
                    <host>server-
02</host>

<port>9000</port>
                </replica>

<replica>
                    <host>server-
04</host>

<port>9000</port>
                </replica>
            </shard>
        </my_cluster>
    </remote_servers>

<zookeeper>

<node>
            <host>keeper-
01</host>

<port>9181</port>
        </node>

<node>
            <host>keeper-
02</host>

<port>9181</port>
        </node>

<node>
            <host>keeper-
03</host>

<port>9181</port>
        </node>
    </zookeeper>

<macros>

<shard>01</shard>

<replica>01</replica>
    </macros>
</clickhouse>
```

User configuration

All four server nodes use the same `users.xml`. Copy it into each server directory and set a password after starting the cluster.

users.xml

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```
<clickhouse
replace="true">

<profiles>

<default>

<max_memory_usage>10000000000</max_memory_usage>

<load_balancing>in_order</load_balancing>

<log_queries>1</log_queries>
        </default>
    </profiles>

<users>

<default>

<profile>default</profile>

<networks>

<ip>::/0</ip>
            </networks>

<quota>default</quota>

<access_management>1</access_management>
        </default>
    </users>

<quotas>

<default>

<interval>

<duration>3600</duration>

<queries>0</queries>

<errors>0</errors>

<result_rows>0</result_rows>

<read_rows>0</read_rows>

<execution_time>0</execution_time>
            </interval>
        </default>
    </quotas>
</clickhouse>
```

Creating the Compose file

Create `docker-compose.yml` in the `cluster/` directory:

```
services:
  keeper-01:
    image: docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
    container_name: keeper-01
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```
    hostname: keeper-01
    entrypoint: ["/usr/bin/clickhouse-keeper", "--config-file=/etc/clickhouse-keeper/keeper_config.xml"]
    volumes:
      - ./keeper-01/keeper_config.xml:/etc/clickhouse-keeper/keeper_config.xml

  keeper-02:
    image: docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
    container_name: keeper-02
    hostname: keeper-02
    entrypoint: ["/usr/bin/clickhouse-keeper", "--config-file=/etc/clickhouse-keeper/keeper_config.xml"]
    volumes:
      - ./keeper-02/keeper_config.xml:/etc/clickhouse-keeper/keeper_config.xml

  keeper-03:
    image: docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
    container_name: keeper-03
    hostname: keeper-03
    entrypoint: ["/usr/bin/clickhouse-keeper", "--config-file=/etc/clickhouse-keeper/keeper_config.xml"]
    volumes:
      - ./keeper-03/keeper_config.xml:/etc/clickhouse-keeper/keeper_config.xml

  server-01:
    image: docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
    container_name: server-01
    hostname: server-01
    ulimits:
      nofile: { soft: 262144, hard: 262144 }
    ports:
      - "127.0.0.1:18123:8123"
      - "127.0.0.1:19000:9000"
    volumes:
      - ./server-01/config.xml:/etc/clickhouse-server/config.d/config.xml
      - ./server-01/users.xml:/etc/clickhouse-server/users.d/users.xml
    depends_on: [keeper-01, keeper-02, keeper-
03]

  server-02:
    image: docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
    container_name: server-02
    hostname: server-02
    ulimits:
      nofile: { soft: 262144, hard: 262144 }
    ports:
      - "127.0.0.1:18124:8123"
      - "127.0.0.1:19001:9000"
    volumes:
      - ./server-02/config.xml:/etc/clickhouse-server/config.d/config.xml
      - ./server-02/users.xml:/etc/clickhouse-server/users.d/users.xml
    depends_on: [keeper-01, keeper-02, keeper-
03]

  server-03:
    image: docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
    container_name: server-03
    hostname: server-03
    ulimits:
      nofile: { soft: 262144, hard: 262144 }
    ports:
      - "127.0.0.1:18125:8123"
      - "127.0.0.1:19002:9000"
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```
    volumes:
      - ./server-03/config.xml:/etc/clickhouse-server/config.d/config.xml
      - ./server-03/users.xml:/etc/clickhouse-server/users.d/users.xml
    depends_on: [keeper-01, keeper-02, keeper-
03]

  server-04:
    image: docker.enterprisedb.com/edb-clickhouse-server:26.3.12.3
    container_name: server-04
    hostname: server-04
    ulimits:
      nofile: { soft: 262144, hard: 262144 }
    ports:
      - "127.0.0.1:18126:8123"
      - "127.0.0.1:19003:9000"
    volumes:
      - ./server-04/config.xml:/etc/clickhouse-server/config.d/config.xml
      - ./server-04/users.xml:/etc/clickhouse-server/users.d/users.xml
    depends_on: [keeper-01, keeper-02, keeper-
03]
```

Starting and verifying the cluster

1. From the `cluster/` directory, start all containers:

```
docker compose up -d
```

2. Connect to `server-01`:

```
docker exec -it server-01 clickhouse-
client
```

3. Confirm all server nodes are registered. `system.clusters` lists server nodes only, so the dedicated Keeper containers don't appear:

```sql
SELECT cluster, shard_num, replica_num, host_name
FROM system.clusters
WHERE cluster = 'my_cluster';
```

output

```
┌─cluster────┬─shard_num─┬─replica_num─┬─host_name─┐
│ my_cluster │         1 │           1 │ server-01 │
│ my_cluster │         1 │           2 │ server-03 │
│ my_cluster │         2 │           1 │ server-02 │
│ my_cluster │         2 │           2 │ server-04 │
└────────────┴───────────┴─────────────┴───────────┘
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

4. Check Keeper connectivity across all nodes. The query uses `clusterAllReplicas` to run against every server node and returns the root znodes each one sees. Every node returning rows confirms all servers can reach Keeper:

```
SELECT hostName(), name,
path
FROM clusterAllReplicas('my_cluster',
system.zookeeper)
WHERE path = '/';
```

```
output
┌─hostName()─┬─name───────┬─path─┐
│ server-01  │ keeper     │ /    │
│ server-01  │ clickhouse │ /    │
│ server-02  │ keeper     │ /    │
│ server-02  │ clickhouse │ /    │
│ server-03  │ keeper     │ /    │
│ server-03  │ clickhouse │ /    │
│ server-04  │ keeper     │ /    │
│ server-04  │ clickhouse │ /    │
└────────────┴────────────┴──────┘
```

Testing replication and sharding

Create a local `ReplicatedMergeTree` table and a `Distributed` table on top of it. Insert some rows and query both tables to confirm that replication is working within each shard and that the distributed table aggregates data across all shards.

1. Create a local storage table. The `ReplicatedMergeTree` engine stores data physically on each node and replicates it within the shard:

```
CREATE TABLE events ON CLUSTER my_cluster
(
    event_id
UInt32,
    user_id    UInt32,
    event_type String,
    created_at
DateTime
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/events',
'{replica}')
ORDER BY event_id;
```

2. Create a `Distributed` table on top of `events`. Unlike the local table, it stores no data. It routes queries to `events` across all shards and aggregates the results:

```
CREATE TABLE events_dist ON CLUSTER my_cluster
(
    event_id
UInt32,
    user_id    UInt32,
    event_type String,
    created_at
DateTime
) ENGINE = Distributed(my_cluster, default, events,
rand());
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

3. Insert some rows via `events_dist`. The `rand()` sharding key distributes each row across shards:

```
INSERT INTO events_dist
VALUES
    (1, 101, 'login',
now()),
    (2, 102, 'purchase',
now()),
    (3, 103, 'logout',
now()),
    (4, 104, 'login',
now());
```

4. Query the local table on any server node. Each node returns only the rows on its shard. The specific rows vary depending on how `rand()` distributed them:

```
SELECT * FROM events;
```

```
output
┌─event_id─┬─user_id─┬─event_type─┬──────────created_at─┐
│        2 │     102 │ purchase   │ 2026-06-10 14:48:15 │
│        3 │     103 │ logout     │ 2026-06-10 14:48:15 │
└──────────┴─────────┴────────────┴─────────────────────┘
```

5. Query `events_dist` from any node to confirm all rows are returned across shards:

```
SELECT * FROM events_dist ORDER BY
event_id;
```

```
output
┌─event_id─┬─user_id─┬─event_type─┬──────────created_at─┐
│        1 │     101 │ login      │ 2026-06-10 14:48:15 │
│        2 │     102 │ purchase   │ 2026-06-10 14:48:15 │
│        3 │     103 │ logout     │ 2026-06-10 14:48:15 │
│        4 │     104 │ login      │ 2026-06-10 14:48:15 │
└──────────┴─────────┴────────────┴─────────────────────┘
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

4.3 Installing EDB Postgres AI for ClickHouse on Kubernetes

The EDB ClickHouse Kubernetes operator manages ClickHouse deployments on Kubernetes, handling cluster provisioning, configuration, upgrades, and scaling.

When installed via the Helm chart, it creates:

- Four CustomResourceDefinitions (CRDs):
  - `clickhouseinstallations` (abbreviated `chi`)
  - `clickhouseinstallationtemplates`
  - `clickhouseoperatorconfigurations`
  - `clickhousekeeperinstallations` (abbreviated `chk`)
- A Deployment running the operator and metrics-exporter sidecar (`2/2 Ready`).
- A ServiceAccount, Role, ClusterRole, and bindings the operator needs to manage ClickHouse resources cluster-wide.

Prerequisites

- A running Kubernetes cluster (1.26 or later)
- `kubectl` configured to talk to your cluster
- Helm 3.8 or later
- An EDB Cloudsmith account with read access to the `clickhouse` repository. You'll use two credentials from it:
  - **Entitlement token** — embedded in the Helm chart repo URL. Get it from your EDB account at https://www.enterprisedb.com/accounts/login.
  - **Cloudsmith API key** — used as the Docker password when Kubernetes pulls images from `docker.enterprisedb.com`. Get it from https://cloudsmith.io/user/settings/api/ (log in via EDB SSO).

Creating the image pull secret

Kubernetes needs credentials to pull EDB container images from `docker.enterprisedb.com`. Create a `docker-registry` secret in the namespace you'll install the operator into:

```
kubectl create namespace
clickhouse
kubectl create secret docker-registry cs-pull
\
  --namespace=clickhouse \
  --docker-server=docker.enterprisedb.com \
  --docker-username=<your-cloudsmith-username> \
  --docker-password=<your-cloudsmith-api-key>
```

You'll reference `cs-pull` both in the Helm install (so the operator pod can pull its own images) and in every `ClickHouseInstallation` (so the ClickHouse pods can pull `edb-clickhouse-server`).

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

Installing the operator

1. Add the EDB Helm repository. Replace `<entitlement-token>` with the token from your EDB account and `<cs-repo>` with the EDB Cloudsmith repository name listed in your release notes:

```
helm repo add edb
\
  "https://dl.cloudsmith.io/<entitlement-token>/enterprisedb/<cs-repo>/helm/charts/"
helm repo update
edb
```

2. Install the chart. Specify `--version` explicitly because the EDB release suffix (`-NedbM`) is a SemVer pre-release tag and `helm search` filters those out by default:

```
helm upgrade --install clickhouse-operator
\
  edb/edb-clickhouse-operator-chart
\
  --version <operator-version> \
  --namespace clickhouse \
  --set 'imagePullSecrets[0].name=cs-pull'
```

3. Verify the operator is running:

```
kubectl get pods -n
clickhouse
```

The operator pod reports `2/2 Ready` within about 30 seconds — the two containers are the operator and the metrics-exporter sidecar.

4. Confirm the CRDs registered:

```
kubectl get crds | grep
altinity
```

All four CRDs listed above appear in the output.

Creating a ClickHouse cluster

The operator watches for `ClickHouseInstallation` resources and reconciles them into running ClickHouse pods. Create a file named `clickhouse-installation.yaml`:

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```yaml
apiVersion: clickhouse.altinity.com/v1
kind: ClickHouseInstallation
metadata:
  name: my-clickhouse
  namespace: clickhouse
spec:
  configuration:
    clusters:
      - name: my-cluster
        templates:
          podTemplate: edb-clickhouse
        layout:
          shardsCount: 1
          replicasCount: 1
  templates:
    podTemplates:
      - name: edb-clickhouse
        spec:
          imagePullSecrets:
            - name: cs-pull
          containers:
            - name: clickhouse
              image: docker.enterprisedb.com/<cs-repo>/edb-clickhouse-server:<server-version>
              ports:
                - name: http
                  containerPort: 8123
                - name: client
                  containerPort: 9000
```

Note that `imagePullSecrets` appears inside the pod template — this applies to the ClickHouse pods the operator creates, not the operator pod itself. Both need it.

`<server-version>` is the Docker image tag for `edb-clickhouse-server`. The Docker tag doesn't carry the `-NedbM` suffix that the RPM packages use; only the upstream version (for example, `26.3.12.1`). Check your release notes for the version compatible with your operator version.

Apply the resource and watch the cluster come up:

```
kubectl apply -f clickhouse-installation.yaml
kubectl get chi -n clickhouse -w
```

Wait until `STATUS` shows `Completed`, which takes about 60 to 90 seconds.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

Connecting to ClickHouse

The operator creates a headless Service per cluster. Connect via the Service name using port forwarding:

```
kubectl port-forward -n clickhouse svc/clickhouse-my-clickhouse 9000:9000
```

Then in another terminal:

```
clickhouse-client --host 127.0.0.1 --port 9000
```

Or run a one-shot query inside the cluster:

```
kubectl exec -it -n clickhouse \
  "$(kubectl get pods -n clickhouse -l clickhouse.altinity.com/chi=my-clickhouse -o name | head -1)"
\
  -- clickhouse-client --query "SELECT
version()"
```

The version returned includes the `(EDB Build)` suffix, confirming you're running the EDB distribution.

Uninstalling

```
kubectl delete chi my-clickhouse -n clickhouse        # operator deletes pods and
services
helm uninstall clickhouse-operator -n clickhouse      # removes the operator Deployment; CRDs
survive
kubectl delete secret cs-pull -n clickhouse           #
optional
```

CRDs are intentionally not deleted by `helm uninstall`. Deleting CRDs deletes every dependent `ClickHouseInstallation`, including any in other namespaces. Remove them only if you have no other operator instances in the cluster:

```
kubectl delete crd
\
  clickhouseinstallations.clickhouse.altinity.com
\
  clickhouseinstallationtemplates.clickhouse.altinity.com
\
  clickhouseoperatorconfigurations.clickhouse.altinity.com \
  clickhousekeeperinstallations.clickhouse-keeper.altinity.com
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

5 Getting started with ClickHouse

Connect to your ClickHouse instance, explore EDB-specific capabilities, and find resources for going further.

Connecting to ClickHouse

Once ClickHouse is installed and running, connect using `clickhouse-client`:

```
clickhouse-client
```

Verify you're running an EDB-packaged build:

```
SELECT value FROM system.build_options WHERE name = 'VERSION_OFFICIAL';
```

Integrations

- Connecting to WarehousePG: read from and write to WarehousePG tables using the `PostgreSQL` table engine
- Reading Apache Iceberg® tables: query Iceberg tables stored in Amazon S3 or Azure Blob Storage

See also

For SQL reference, configuration, and operational guidance beyond what's covered here, refer to the upstream ClickHouse documentation:

- Creating tables
- Inserting data
- Writing queries
- SQL reference
- Table engines
- Server configuration
- Best practices

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

6 Connecting ClickHouse to WarehousePG

Query WarehousePG data from ClickHouse, and write data back to WarehousePG, without ETL pipelines or data duplication. ClickHouse connects to WarehousePG using the `PostgreSQL` table engine, which maps a WarehousePG table to a ClickHouse table definition and handles reads and writes over the standard Postgres wire protocol.

> **Note**
>
> The `PostgreSQL` engine connects only to the WarehousePG coordinator. It doesn't engage WarehousePG's massively parallel processing (MPP) architecture, so queries run through the coordinator as single-node Postgres queries.

Configuring WarehousePG to accept connections from ClickHouse

1. On the coordinator host, confirm `$COORDINATOR_DATA_DIRECTORY/postgresql.conf` has `listen_addresses` set to `'*'` or your network range. If not, update it:

```
listen_addresses = '*'
```

2. Add an entry to `pg_hba.conf` that allows connections from the ClickHouse host. Replace `<clickhouse-host-network>` with the network range of your ClickHouse host:

```
echo "host    <database>    <user>    <clickhouse-host-network>/24    md5" >>
$COORDINATOR_DATA_DIRECTORY/pg_hba.conf
```

3. Reload the coordinator configuration:

```
gpstop -u
```

4. From the ClickHouse host, verify the WarehousePG coordinator is reachable:

```
psql -h <coordinator-ip> -p <port> -d <database> -U
<user>
```

Creating the linked table in ClickHouse

Map a WarehousePG table to a ClickHouse table definition using the `PostgreSQL` table engine. Once created, queries against the ClickHouse table go directly to WarehousePG in real time, with no data copied or cached locally.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

1. In WarehousePG, create the table to expose to ClickHouse. For example:

```
CREATE TABLE analytics_data
(
    id          integer primary key,
    event_type varchar(50),
    event_time timestamp,
    value       numeric(12, 4)
);

INSERT INTO analytics_data (id, event_type, event_time,
value)
VALUES
    (1, 'page_view', '2025-01-01 10:00:00',
1.0),
    (2, 'click',      '2025-01-01 10:01:00',
2.5);
```

2. From your ClickHouse server, open a `clickhouse-client` session:

```
clickhouse-client
```

3. Create a database and table in ClickHouse that maps to the WarehousePG table using `ENGINE = PostgreSQL`:

```
CREATE DATABASE whpg_connect;

CREATE TABLE whpg_connect.analytics_data
(
    id          UInt64,
    event_type String,
    event_time DateTime,
    value       Float64
)
ENGINE =
PostgreSQL(
    '<coordinator-ip>:<port>',  -- host:port of the WarehousePG
coordinator
    '<database>',               -- WarehousePG database
name
    'analytics_data',           -- table name in
WarehousePG
    '<user>',                   -- WarehousePG
user
    '<password>'                -- password (stored encrypted in
ClickHouse)
);
```

The `ENGINE = PostgreSQL` syntax takes five positional parameters: `host:port`, database, table, user, and password. An optional sixth parameter specifies the schema name (defaults to `public`). If WarehousePG is configured with `trust` authentication, pass an empty string `''` for the password.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

4. Verify the table is correctly linked by inspecting its definition:

```
DESCRIBE TABLE whpg_connect.analytics_data;
```

Or view the full `CREATE TABLE` statement:

```
SHOW CREATE TABLE whpg_connect.analytics_data;
```

The password is shown as `[HIDDEN]` in the output.

Reading WarehousePG data from ClickHouse

Query the ClickHouse table. ClickHouse connects to WarehousePG, executes a `COPY ... TO STDOUT` statement, and converts the row-oriented stream into columnar format as it arrives:

```
SELECT * FROM whpg_connect.analytics_data;
```

```
output
┌─id─┬─event_type─┬─────────event_time─┬─value─┐
│  1 │ page_view  │ 2025-01-01 10:00:00│   1   │
│  2 │ click      │ 2025-01-01 10:01:00│  2.5  │
└────┴────────────┴────────────────────┴───────┘
```

Data inserted into WarehousePG after the ClickHouse table is created is immediately visible on the next `SELECT`. ClickHouse doesn't cache WarehousePG data locally when using the `PostgreSQL` engine.

Writing data to WarehousePG from ClickHouse

Use `INSERT` on the ClickHouse table to write data to WarehousePG. ClickHouse translates the insert into a `COPY ... FROM STDIN` statement on the WarehousePG side:

```
-- Run in
ClickHouse
INSERT INTO whpg_connect.analytics_data (id, event_type, event_time,
value)
VALUES (3, 'purchase', '2025-01-01 10:05:00',
99.99);
```

Confirm the row is visible in WarehousePG:

```
-- Run in
WarehousePG
SELECT * FROM analytics_data;
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

Understanding the query flow

Use `EXPLAIN` in ClickHouse to see how data moves between the systems:

```
EXPLAIN SELECT * FROM whpg_connect.analytics_data;
```

```
output
┌─explain─────────────────────────────────────────────────────────────────────┐
│ Expression ((Project names + (Projection + Change column names to column identifiers))) │
│   ReadFromPostgreSQL                                                                     │
└──────────────────────────────────────────────────────────────────────────────────────────┘
```

The `ReadFromPostgreSQL` step establishes a TCP connection to the coordinator, sends the SQL query, and converts the incoming row-oriented stream into ClickHouse's columnar format. By the time data reaches the `Expression` step, it's already in columnar form.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

7 Integrating with Iceberg catalogs

Connect ClickHouse to an external Iceberg catalog to run fast analytical queries over your data lake without moving data. An Iceberg catalog is a governance and metadata layer that tracks which tables exist, where their data lives in object storage, and who has access.

Different pipelines and tools, such as Spark jobs, Databricks, or dbt, can write data to object storage, and the catalog provides a single consistent view of all of it. Any query engine that implements the Iceberg protocol, such as ClickHouse, Spark, and Trino, can connect to the same catalog and query the same data independently.

> **Note**
>
> ClickHouse's Iceberg catalog integration is read-only. `INSERT`, `UPDATE`, and `DELETE` operations aren't supported. See Known issues for details.

Connecting to a catalog

Use the `DataLakeCatalog` database engine to connect ClickHouse to an external Iceberg catalog. The engine is experimental and requires opt-in before use.

1. Enable the experimental setting for your catalog type. The required setting varies by catalog. See the DataLakeCatalog reference for the full list. For a REST catalog:

```
SET allow_experimental_database_iceberg = 1;
```

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

2. Create a database that maps to your external catalog. For example:

```
CREATE DATABASE my_catalog
ENGINE =
DataLakeCatalog('https://catalog.example.com/api/iceberg/prj_abc123')
SETTINGS
    catalog_type = 'rest',
    warehouse = 'my-warehouse',
    auth_header = 'Authorization: Bearer
eyJhbGciOiJSUzI1NiJ9.example';
```

Where:

- `catalog_type` is the catalog type. See the DataLakeCatalog reference for valid values.
- `warehouse` is the warehouse or project identifier within the catalog.
- `auth_header` is the HTTP header for bearer token authentication.

For most REST catalogs, authenticate using a static bearer token via `auth_header`, as shown above. For catalogs that support OAuth2 client credentials, use `catalog_credential` instead:

```
CREATE DATABASE my_catalog
ENGINE =
DataLakeCatalog('https://catalog.example.com/api/iceberg/prj_abc123')
SETTINGS
    catalog_type = 'rest',
    warehouse = 'my-warehouse',
    catalog_credential = 'my-client-id:my-client-secret',
    oauth_server_uri =
'https://auth.example.com/oauth/token',
    auth_scope = 'catalog:read';
```

Where:

- `catalog_credential` is the OAuth2 client credentials in `client_id:client_secret` format.
- `oauth_server_uri` is the OAuth2 authorization server URI.
- `auth_scope` is the OAuth2 scope. Defaults to `PRINCIPAL_ROLE:ALL` if not specified.

Querying data

Once the database is created, list all available tables:

```
SHOW TABLES FROM
my_catalog;
```

```
output
┌─name──────────────────────────────┐
│ public.calls                      │
│ public.notes                      │
│ analytics.events                  │
│ duckdb_tpcds_sf_1.catalog_sales   │
└───────────────────────────────────┘
```

Tables are listed in `namespace.table_name` format. Reference them using backticks:

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.

```
SELECT * FROM my_catalog.`public.calls` LIMIT 5;
```

```
output
┌─id─┬─────────────────────────ts─┬─data───┬─category─┐
│  1 │ 2025-05-01 00:01:00.000000 │ data 1 │ bread    │
│  2 │ 2025-05-01 00:02:00.000000 │ data 2 │ fruit    │
│  3 │ 2025-05-01 00:03:00.000000 │ data 3 │ juice    │
└────┴────────────────────────────┴────────┴──────────┘
```

Standard SQL applies, including aggregations, filters, joins, and window functions:

```
SELECT category, count() AS total
FROM my_catalog.`public.calls`
GROUP BY category
ORDER BY total DESC;
```

ClickHouse reads only the Parquet data files required by the query, using Iceberg metadata for partition pruning and predicate pushdown.

Copyright © 2009 - 2026 EnterpriseDB Corporation. All rights reserved.