EDB Postgres AI for ClickHouse

Version 26.8

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

Built at 2026-10-21T14:40:06

1 EDB Postgres AI for ClickHouse 3
2 Release notes 4
2.1 EDB Postgres AI for ClickHouse 26.8.15.10 release notes 5
3 Overview 7
3.1 ClickHouse architecture 9
3.2 EDB Postgres AI for ClickHouse compatibility 12
3.3 Known issues 14
4 Installing and upgrading EDB Postgres AI for ClickHouse 15
4.1 Installing EDB Postgres AI for ClickHouse on Linux 16
4.2 Installing EDB Postgres AI for ClickHouse with Docker 25
4.3 Installing EDB Postgres AI for ClickHouse on Kubernetes 37
5 Getting started with ClickHouse 42
6 Connecting ClickHouse and WarehousePG 43
7 Integrating with Iceberg catalogs 54
8 Connecting AI agents to ClickHouse 57
9 Monitoring Postgres with pg_stat_ch 65

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
26.8.15.10	6 October 2026

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

2.1 EDB Postgres AI for ClickHouse 26.8.15.10 release notes

Released: 6 October 2026

EDB Postgres AI for ClickHouse 26.8.15.10 is the first release of the 26.8 LTS. It updates the ClickHouse server from 26.3.22.7 and updates the Kubernetes operator, MCP server, and pg_stat_ch.

ClickHouse server

Type	Description
Upstream merge	Updated to the open-source ClickHouse 26.8.15.10 LTS release, from 26.3.22.7.
Feature	Added Debian 13 (trixie) `.deb` packages for `amd64`, `arm64`, and `ppc64el`, published alongside the RPM packages.
Bug fix	Disabled sending crash reports and `LOGICAL_ERROR` exceptions to an external endpoint. The `send_crash_reports` configuration section is still accepted for compatibility with existing configuration files, but it no longer has any effect.

Kubernetes operator

Type	Description
Upstream merge	Updated to `0.27.4-2edb1`, tracking upstream Altinity clickhouse-operator release-0.27.4, from `0.27.3`. The EDB release adds no upstream changes. Images are tagged `0.27.4-2edb1` and the Helm chart is versioned `0.27.4+2edb1`.
Bug fix	Updated `google.golang.org/grpc` to `v1.84.0`, `golang.org/x/text` to `v0.42.0`, and `github.com/go-openapi/swag` to `v0.29.2` to address security vulnerabilities, including GHSA-2v4p-qf9q-27wj, GHSA-hrxh-6v49-42gf, and GHSA-vp52-pcj8-j9qc.
Bug fix	The Helm chart and the operator's default images now come from the production repository, `docker.enterprisedb.com/clickhouse`, so a default installation works with a production subscription.
Bug fix	The operator's default ClickHouse and Keeper image is now pinned to `edb-clickhouse-server:26.3.22.7-1edb1` instead of `latest`, so the deployed ClickHouse version no longer changes without an operator release.

MCP server

Type	Description
Upstream merge	Synced to upstream v0.7.0, from `v0.4.1`. Highlights include connection caching, query cancellation, `Host` and `Origin` validation for HTTP and SSE, and named query parameters on `run_query`. Requires clickhouse-connect 1.x. See the upstream release notes for the full list of changes.
Feature	Added Debian 13 `.deb` packages for `amd64` and `arm64`, published alongside the RPM packages.
Change	`CLICKHOUSE_SECURE` now accepts only `true` or `false`, and `CLICKHOUSE_VERIFY` accepts only `true`, `false`, or `proxy`. Other values fail at startup.

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

pg_stat_ch

Type	Description
Change	Repackaged as `0.4.0-1edb2`. The upstream version remains pg_stat_ch 0.4.0.
Feature	Added Debian 12 (bookworm) and Debian 13 (trixie) `.deb` packages alongside the RPM packages. Debian 13 packages cover community PostgreSQL only. See Compatibility.
Bug fix	The background worker no longer registers or connects to `postgres` when the postmaster is started by `pg_upgrade` in `-b` mode. Previously, that open connection blocked the `DROP DATABASE "postgres"` step in `pg_restore` and caused every binary upgrade to fail when `pg_stat_ch` was in `shared_preload_libraries`.

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 RPM and DEB packages for enterprise Linux distributions and Debian, container images, and a Kubernetes operator, and supported by EDB.

EDB's contribution is:

- Building the official open-source ClickHouse source at a pinned upstream release
- Packaging the binary as RPM and DEB packages, 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 version();
```

The output includes the EDB release version, for example `26.8.15.10+1edb1`.

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

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.

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

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.

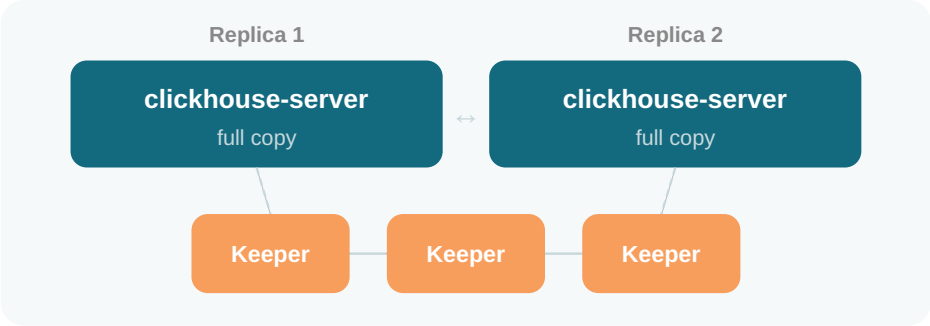
Sharding doesn't require cluster coordination. Query routing is handled by the `Distributed` table engine, which sends queries to all shards in parallel and merges the results transparently. See How ClickHouse routes queries for details.

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.

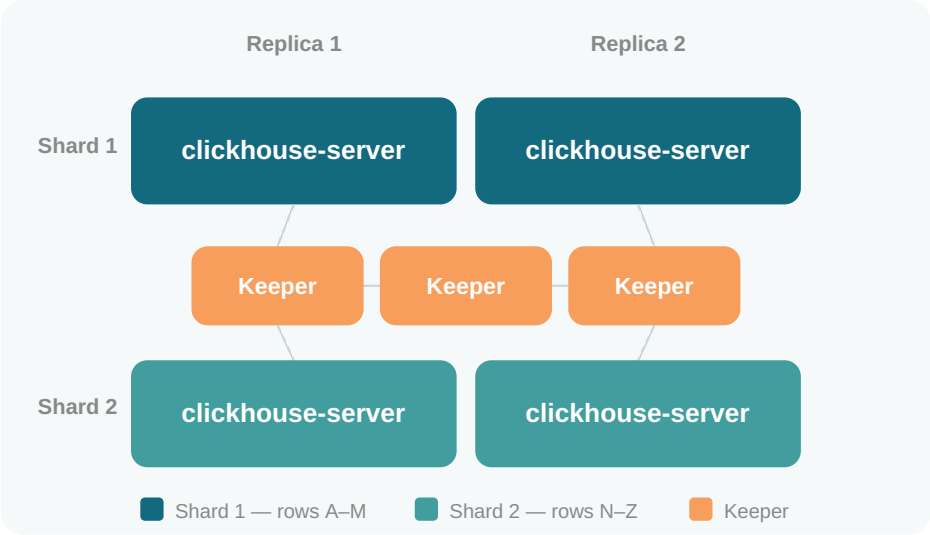
Keeper can run in two ways. In a co-located deployment, Keeper runs alongside the ClickHouse server on the same nodes. Co-location is simpler to manage and works well for smaller clusters. In a dedicated deployment, Keeper runs on separate nodes with no server workload, which gives better isolation and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

- RHEL 8, 9, and 10, and Rocky Linux 9 and 10, on x86_64, aarch64, and ppc64le.
- Debian 13 (trixie) on amd64, arm64, and ppc64el.

System requirements

Minimum recommended hardware per node:

	Single-node	Cluster node
CPU	16 cores	16 cores
RAM	128 GB (8 GB per core)	128 GB (8 GB per core)
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.

pg_stat_ch

Postgres 16, 17, and 18, on community PostgreSQL, EDB Postgres Extended Server, and EDB Postgres Advanced Server. Packages are built for Rocky Linux 9 on x86-64 and arm64, and for Debian 12 (bookworm) and Debian 13 (trixie) on amd64 and arm64. These are the operating systems of the Postgres host, independent of the platform requirements for ClickHouse itself. Debian 12 packages cover all three Postgres variants. Debian 13 packages cover community PostgreSQL only, because EDB publishes no EDB Postgres Advanced Server or EDB Postgres Extended Server packages for Debian 13. See Monitoring Postgres with pg_stat_ch.

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`

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

Provider	Table engines
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 upgrading 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:

- Linux packages, for RHEL, Rocky Linux, and Debian
- Docker
- Kubernetes operator

If you're already running EDB Postgres AI for ClickHouse, see the upgrade section on the page for your installation method.

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

4.1 Installing EDB Postgres AI for ClickHouse on Linux

EDB distributes ClickHouse as RPM packages for RHEL 8, 9, and 10, and Rocky Linux 9 and 10, on x86_64, aarch64, and ppc64le, and as DEB packages for Debian 13 (trixie) on amd64, arm64, and ppc64el. The following packages are available on both:

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. `dnf` and `apt-get` install and update it automatically as a dependency, so you don't need to list it explicitly.	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

- RHEL 8, 9, or 10, Rocky Linux 9 or 10, or Debian 13 (trixie)
- `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-server \
  edb-clickhouse-client
```

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

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

The package manager pulls in `edb-clickhouse-common-static` automatically as a dependency.

1. Start the server:

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

   Verify the server is running:

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

2. 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
   ```

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

   ```
   SELECT version();
   ```

   The output includes the EDB release version, for example `26.8.15.10+1edb1`.

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
```

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

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

   On each server node:

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

```
sudo apt-get install -y \
  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
```

```
sudo apt-get 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 package 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:

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

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

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());
```

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());
```

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

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     │
└──────────┴─────────┴────────────┴─────────────────────────┘
```

Upgrading

Check Release notes for the ClickHouse server version you're upgrading to before you start.

On a cluster, repeat the following steps one node at a time, keeping at least one replica per shard on its current version until the upgrade completes. See the upstream ClickHouse upgrade guidance for version-skipping limits and other considerations for large clusters.

1. Refresh the EDB repository metadata, so the package manager sees the new version instead of what it last cached:

```
sudo dnf
makecache
```

```
sudo apt-get
update
```

1. Download and install the updated packages:

```
sudo dnf update -y
\
  edb-clickhouse-server \
  edb-clickhouse-client
```

```
sudo apt-get install -y --only-upgrade \
  edb-clickhouse-server \
  edb-clickhouse-client
```

The package manager pulls in a matching update to `edb-clickhouse-common-static` automatically, the same way it does at install time.

On a dedicated Keeper node, update the Keeper package instead:

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

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

```
sudo apt-get install -y --only-upgrade edb-clickhouse-keeper
```

1. Restart the updated service:

   ```
   sudo systemctl restart clickhouse-
   server
   ```

   Or, on a dedicated Keeper node:

   ```
   sudo systemctl restart clickhouse-
   keeper
   ```

2. Verify the running version:

   ```
   clickhouse-client --query "SELECT
   version();"
   ```

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 Red Hat Universal Base Image (UBI) 10 minimal, is available for x86_64, aarch64, and ppc64le, 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>
echo "$EDB_SUBSCRIPTION_TOKEN" | docker login docker.enterprisedb.com \
  --username clickhouse \
  --password-stdin
docker pull docker.enterprisedb.com/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
```

> **Note**
>
> The username must match the repository you're pulling from. Docker stores credentials keyed by registry with no distinction between repositories, so each login overwrites the previous one. If you need to pull images from a different repository on `docker.enterprisedb.com`, log in again with the appropriate username.

Running a single-node server

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

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

1. Start the server:

```
docker run -d
\
  --name clickhouse-server
\
  --ulimit nofile=262144:262144 \
  -p 8123:8123 \
  -p 9000:9000 \
  docker.enterprisedb.com/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
```

   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/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
```

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/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
```

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

3. Connect with the ClickHouse client. The default password for the `default` user is `admin123`:

```
docker exec -it clickhouse-server clickhouse-client --password
admin123
```

4. Verify the EDB build:

```
SELECT version();
```

   The output includes the EDB release, for example `26.8.15.10+1edb1`.

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

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:

```
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

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:

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

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>
                    <host>server-
02</host>

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

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

```
<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/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
    container_name: keeper-01
    hostname: keeper-01
    entrypoint: ["/usr/bin/clickhouse-keeper", "--config-file=/etc/clickhouse-keeper/keeper_config.xml"]
```

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

```
    volumes:
      - ./keeper-01/keeper_config.xml:/etc/clickhouse-keeper/keeper_config.xml

  keeper-02:
    image: docker.enterprisedb.com/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
    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/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
    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/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
    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/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
    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/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
    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/clickhouse/edb-clickhouse-server:26.8.15.10-
1edb1
    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`. The default password for the `default` user is `admin123`:

   ```
   docker exec -it server-01 clickhouse-client --password
   admin123
   ```

3. Confirm all server nodes are registered. `system.clusters` lists server nodes only, so the dedicated Keeper containers don'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 │ 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     │
└──────────┴─────────┴────────────┴─────────────────────────┘
```

Upgrading

Check Release notes for the ClickHouse server version you're upgrading to before you start.

On a Compose-based cluster, upgrade one service at a time, keeping at least one replica per shard on its current version until the upgrade completes. See the upstream ClickHouse upgrade guidance for version-skipping limits and other considerations for large clusters.

1. Pull the new image tag:

```
docker pull docker.enterprisedb.com/clickhouse/edb-clickhouse-server:<new-version>-
1edb1
```

   Replace `<new-version>` with the target version from Release notes.

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

2. For a single-node deployment, recreate the container with the new image, keeping the same volume mounts and ports as the original `docker run` command:

```
docker stop clickhouse-
server
docker rm clickhouse-
server
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/clickhouse/edb-clickhouse-server:<new-version>-
1edb1
```

   With a persistent volume mounted at `/var/lib/clickhouse`, data survives the container recreation.

3. For a Compose-based cluster, in the `docker-compose.yml` you created under Creating the Compose file, edit one service's `image` line to the new tag. For example, for `server-01`:

```
server-01:
  image: docker.enterprisedb.com/clickhouse/edb-clickhouse-server:<new-version>-
1edb1
```

   Then recreate just that container:

```
docker compose up -d --no-deps server-01
```

   `--no-deps` recreates only `server-01`, leaving the other, still-healthy containers running. Confirm it rejoins and reports the new version, then repeat for each remaining service in turn.

4. Verify the running version. For a single-node deployment:

```
docker exec -it clickhouse-server clickhouse-client --password admin123 --query "SELECT
version();"
```

   Or, for a Compose-based cluster, against the container you just upgraded:

```
docker exec -it server-01 clickhouse-client --password admin123 --query "SELECT
version();"
```

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 subscription token, available from the EDB customer portal. You'll use this both in the Helm chart repo URL and as the Docker password when Kubernetes pulls images.

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:

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
kubectl create namespace
clickhouse
kubectl create secret docker-registry cs-pull
\
  --namespace=clickhouse \
  --docker-server=docker.enterprisedb.com \
  --docker-username=clickhouse \
  --docker-password="$EDB_SUBSCRIPTION_TOKEN"
```

You'll reference `cs-pull` 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`.

> **Note**
>
> The username must match the repository you're pulling from. Docker stores credentials keyed by registry with no distinction between repositories, so each login overwrites the previous one. If you need to pull images from a different repository on `docker.enterprisedb.com`, log in again with the appropriate username.

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

Installing the operator

1. Add the EDB Helm repository:

```
helm repo add edb
\
  "https://dl.cloudsmith.io/$EDB_SUBSCRIPTION_TOKEN/enterprisedb/clickhouse/helm/charts/"
helm repo update
edb
```

2. Install the chart:

```
helm upgrade --install clickhouse-operator
\
  edb/edb-clickhouse-operator-chart
\
  --version 0.27.4+2edb1 \
  --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
clickhouse
```

   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.

```
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/clickhouse/edb-clickhouse-server:<server-version>-
1edb1
              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.

Replace `<server-version>` with the `edb-clickhouse-server` version you want to run, for example, `26.8.15.10`. The operator runs the image you set here, so the same operator version works with every supported ClickHouse LTS release, including 26.8 and 26.3. Check Release notes for available versions.

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.

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. The default password for the `default` user is `admin123`:

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

```
clickhouse-client --host 127.0.0.1 --port 9000 --password
admin123
```

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 --password admin123 --query "SELECT
version()"
```

The version returned includes the EDB release, for example `26.8.15.10+1edb1`, confirming you're running the EDB distribution.

Upgrading

Check Release notes for the combination of `edb-clickhouse-server` and operator versions validated together before you upgrade either.

1. Update the Helm repository and upgrade the release to the target operator version:

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

   Replace `<new-operator-version>` with the target version from Release notes. `helm upgrade` updates the operator Deployment and its CRDs in place, without touching existing `ClickHouseInstallation` resources.

2. Confirm the operator pod restarted on the new version:

   ```
   kubectl rollout status deployment/clickhouse-operator -n
   clickhouse
   ```

Upgrading the operator doesn't upgrade the ClickHouse server pods it manages. To upgrade the pods:

1. Pull the new `edb-clickhouse-server` image to confirm it's available before rolling it out to the cluster:

   ```
   docker pull docker.enterprisedb.com/clickhouse/edb-clickhouse-server:<new-version>-
   1edb1
   ```

   Replace `<new-version>` with the target version from Release notes.

2. Edit the `image` field under `podTemplates` in your `ClickHouseInstallation` resource to the same tag, then reapply it:

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

The operator rolls the new image out to pods one at a time, respecting shard and replica placement, until `STATUS` shows `Completed` again.

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

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 version();
```

The output includes the EDB release, for example `26.8.15.10+1edb1`.

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
- Connecting AI agents with MCP: let AI agents and LLM-based tools query ClickHouse using the MCP server
- Monitoring Postgres with pg_stat_ch: capture per-query execution telemetry from Postgres and analyze it in ClickHouse

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 and WarehousePG

Connect ClickHouse and WarehousePG in either direction, without ETL pipelines or data duplication:

- Connecting ClickHouse to WarehousePG — query and write WarehousePG data from ClickHouse using the PostgreSQL table function, table engine, or database engine.
- Connecting WarehousePG to ClickHouse — read ClickHouse data from WarehousePG using PXF.

Connecting ClickHouse to WarehousePG

ClickHouse connects to WarehousePG over the standard Postgres wire protocol through three built-in mechanisms:

- Table function — ad-hoc, one-off queries with no permanent object in ClickHouse.
- Table engine — repeated access, with the connection defined once as a named table.
- Database engine — exposing an entire WarehousePG database, with all tables becoming queryable automatically.

> **Note**
>
> All three mechanisms connect to the WarehousePG coordinator only and don't engage WarehousePG's massively parallel processing (MPP) architecture. Queries run through the coordinator as single-node Postgres queries. `SELECT` and `INSERT` are supported across all three mechanisms. `UPDATE`, `DELETE`, `TRUNCATE`, and DDL operations aren't supported. Make those changes directly in WarehousePG.

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 your ClickHouse host or hosts. Each node in a ClickHouse cluster connects to the coordinator independently, so the entry must cover all node addresses. A CIDR range works if all nodes are on the same subnet:

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

   Replace `<clickhouse-host-network>` with the network address of your ClickHouse host or subnet.

3. Reload the coordinator configuration:

   ```
   gpstop -u
   ```

4. From any ClickHouse node, open a `clickhouse-client` session and verify the WarehousePG coordinator is reachable:

   ```
   -- in ClickHouse
   SELECT nspname
   FROM postgresql('<coordinator-ip>:<port>', '<database>', 'pg_namespace', '<user>', '<password>',
   'pg_catalog')
   LIMIT 1;
   ```

   A successful connection returns a row. An authentication error confirms the network path is open but credentials or `pg_hba.conf` need attention.

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

Using the table function

Use the `postgresql` table function to query a WarehousePG table inline, without creating any object in ClickHouse. Each call opens a connection, runs the statement, and closes it.

The function takes five positional arguments: `host:port`, database, table, user, and password. An optional sixth argument specifies the schema name and defaults to `public`. All arguments are positional, so you can't omit the password even if WarehousePG uses `trust` authentication. Pass an empty string `''` in that case.

1. Open a `clickhouse-client` session on any node, then pass the connection details inline with each query:

```
-- in ClickHouse
SELECT *
FROM postgresql('<coordinator-ip>:<port>', '<database>', '<table>', '<user>',
'<password>')
ORDER BY id;
```

2. Use `INSERT INTO TABLE FUNCTION` to write rows to WarehousePG:

```
-- in ClickHouse
INSERT INTO TABLE FUNCTION
  postgresql('<coordinator-ip>:<port>', '<database>', '<table>', '<user>',
'<password>')
  (id, region,
amount)
VALUES (100, 'new-row',
999);
```

Using the table engine

Use the `PostgreSQL` table engine to register a WarehousePG table as a named object in ClickHouse and query it by name. Queries go directly to WarehousePG in real time, with no data copied or cached locally.

Creating the linked table

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

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

```
-- in
WarehousePG
CREATE TABLE analytics_data
(
    id         integer,
    event_type varchar(50),
    event_time timestamp,
    value      numeric(12, 4)
) DISTRIBUTED BY
(id);

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 10:05:00',
2.5);
```

2. From any ClickHouse node, open a `clickhouse-client` session and create a table mapped to the WarehousePG table using `ENGINE = PostgreSQL`. On a cluster, use `ON CLUSTER <cluster-name>` to create the table on all nodes, where `<cluster-name>` is your ClickHouse cluster name (check with `SELECT DISTINCT cluster FROM system.clusters`):

```
-- in ClickHouse
CREATE TABLE whpg_analytics ON CLUSTER
<cluster-name>
(
    id
Int32,
    event_type String,
    event_time DateTime,
    value      Decimal(12, 4)
)
ENGINE =
PostgreSQL(
    '<coordinator-ip>:<port>',
    '<database>',
    'analytics_data',
    '<user>',
    '<password>'
);
```

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

3. In `clickhouse-client`, verify the table was created with the expected column types:

```
-- in ClickHouse
DESCRIBE TABLE whpg_analytics;
```

Or view the full `CREATE TABLE` statement:

```
-- in ClickHouse
SHOW CREATE TABLE whpg_analytics;
```

The password appears as `[HIDDEN]` in the output.

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

Reading data

Query the ClickHouse table as you would any local table:

```
-- in ClickHouse
SELECT * FROM whpg_analytics;
```

```
output
┌─id─┬─event_type─┬──────────event_time─┬──────value─┐
│  1 │ page_view  │ 2025-01-01 10:00:00│     1.0000 │
│  2 │ click      │ 2025-01-01 10:05:00│     2.5000 │
└────┴────────────┴─────────────────────┴────────────┘
```

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

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:

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

Confirm the row is visible by querying the table in WarehousePG:

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

Using the database engine

Use the `PostgreSQL` database engine to expose an entire WarehousePG database in ClickHouse. Every table becomes queryable automatically without individual table definitions.

1. Create the database:

   ```
   -- in ClickHouse
   CREATE DATABASE whpg_db ON CLUSTER <cluster-name>  -- ClickHouse cluster name
     ENGINE = PostgreSQL('<coordinator-ip>:<port>', '<database>', '<user>',
   '<password>');
   ```

2. List the available tables:

   ```
   -- in ClickHouse
   SHOW TABLES FROM
   whpg_db;
   ```

3. Query any table directly by its WarehousePG name:

   ```
   -- in ClickHouse
   SELECT count() FROM whpg_db.analytics_data;
   ```

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

Storing credentials in a named collection

Store connection credentials once in a named collection and reference it by name across queries and `CREATE TABLE` statements. Named collections work with all three mechanisms.

1. Enable named collection management for your ClickHouse user by creating `/etc/clickhouse-server/users.d/named_collection_control.xml` on each server node with the following content:

```
<clickhouse>

<users>

<default>

<named_collection_control>1</named_collection_control>
        </default>
    </users>
</clickhouse>
```

2. From `clickhouse-client` on each server node, reload the configuration (server nodes only, not Keeper nodes):

```
-- in ClickHouse
SYSTEM RELOAD
CONFIG;
SYSTEM RELOAD
USERS;
```

3. Create the named collection on all server nodes using `ON CLUSTER <cluster-name>`:

```
-- in ClickHouse
CREATE NAMED COLLECTION whpg ON CLUSTER <cluster-name> AS
  host='<coordinator-ip>', port=<port>, user='<user>',
  password='<password>', database='<database>';
```

4. Reference it by name in any of the three mechanisms:

```
-- in ClickHouse

-- Table
function
SELECT * FROM postgresql(whpg, table =
'<table>');

-- Table
engine
CREATE TABLE <table-name> (id Int32, event_type String, value Decimal(12,
4))
ENGINE = PostgreSQL(whpg, table =
'<table>');

-- Database
engine
CREATE DATABASE whpg_db ON CLUSTER <cluster-name>
ENGINE = PostgreSQL(whpg);
```

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

Understanding the query flow

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

```
-- in ClickHouse
EXPLAIN SELECT id, value FROM whpg_analytics WHERE id =
1;
```

```
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 row-oriented stream into ClickHouse's columnar format. Simple filters and column projections are pushed down to WarehousePG, so only the matching rows and columns travel over the network. Each query opens a fresh connection and closes it on completion.

Connecting WarehousePG to ClickHouse

Read ClickHouse data from WarehousePG using the WarehousePG Platform Extension Framework (PXF), reading in parallel across every WarehousePG segment. Query it either through a `pxf://` external table or, on WarehousePG 7 and later, through a standard foreign table using `pxf_fdw`. Both options read through the same PXF server and offer identical parallelism, so the choice comes down to whether external tables or foreign tables fit your existing workflow better.

Preparing ClickHouse for incoming connections

1. Confirm ClickHouse listens on an address reachable from your WarehousePG hosts. By default, it listens only on `127.0.0.1`. Add an override file at `/etc/clickhouse-server/config.d/listen.xml` on the ClickHouse node your PXF server connects to (just that one node, not the whole cluster):

   ```
   <!-- /etc/clickhouse-server/config.d/listen.xml --
   >
   <clickhouse>

   <listen_host>127.0.0.1</listen_host>
       <listen_host><clickhouse-private-ip>
   </listen_host>
   </clickhouse>
   ```

   Keep `127.0.0.1` in the list to preserve local `clickhouse-client` access.

2. Restart ClickHouse to apply it:

   ```
   sudo systemctl restart clickhouse-
   server
   ```

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

3. From a `clickhouse-client` session on that node, create a dedicated ClickHouse user for WarehousePG to connect with, and grant it read access to the database you want to expose. PXF's JDBC profile accepts only plaintext passwords, so create the user accordingly:

```
-- in ClickHouse
CREATE USER whpg_reader IDENTIFIED WITH plaintext_password BY
'<password>';
GRANT SELECT, INSERT ON <database>.* TO whpg_reader;
```

   Replace `<database>` with the ClickHouse database you want WarehousePG to read from and write to. Drop `INSERT` from the grant if you only need to read.

4. Make sure every WarehousePG host can resolve the ClickHouse hostname.

Configuring PXF's JDBC connection to ClickHouse

Use the WarehousePG Platform Extension Framework (PXF) with the `Jdbc` profile to read ClickHouse tables in parallel across all WarehousePG segments, through ClickHouse's HTTP interface.

Install and start PXF first if you haven't already. See Installing PXF for WarehousePG and Configuring and starting PXF for WarehousePG, including creating the `pxf` extension in the database you want to query from.

Run the following steps on the WHPG coordinator, as the `gpadmin` user. `pxf cluster sync` preserves file ownership when it copies the jar and server directory to every segment, so files owned by another user (for example, if you downloaded the jar as `root`) stay wrong on every segment too.

1. Download the ClickHouse JDBC driver's shaded jar and add it to PXF's library directory:

```
curl -fsSL \
  "https://repo1.maven.org/maven2/com/clickhouse/clickhouse-jdbc/0.9.0/clickhouse-jdbc-0.9.0-all.jar"
\
  -o "$PXF_BASE/lib/clickhouse-jdbc-0.9.0-all.jar"
```

2. Create a directory for the new PXF server:

```
mkdir -p "$PXF_BASE/servers/clickhouse"
```

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

3. Add a `jdbc-site.xml` file pointing at ClickHouse's HTTP interface. Credentials go in separate properties, not the JDBC URL. A `?user=` parameter in `jdbc.url` overrides `jdbc.user`, so leave the URL clean:

```
<!-- $PXF_BASE/servers/clickhouse/jdbc-site.xml --
>
<?xml version="1.0" encoding="UTF-8"?
>
<configuration>

<property>

<name>jdbc.driver</name>

<value>com.clickhouse.jdbc.ClickHouseDriver</value>
    </property>

<property>

<name>jdbc.url</name>
        <!-- Replace <clickhouse-host> and <database> --
>
        <value>jdbc:clickhouse://<clickhouse-host>:8123/<database>
</value>
    </property>

<property>

<name>jdbc.user</name>

<value>whpg_reader</value>
    </property>

<property>

<name>jdbc.password</name>
        <!-- Replace <password> --
>
        <value><password>
</value>
    </property>
</configuration>
```

4. Sync the new jar and server configuration to every segment, then restart PXF to pick them up:

```
pxf cluster sync
pxf cluster restart
```

Querying ClickHouse

Once the server is configured, choose one of the following two options to query it:

- Querying ClickHouse with a pxf:// external table works on any supported WarehousePG version.
- Querying ClickHouse with pxf_fdw is available on WarehousePG 7 and later and queries through standard foreign table syntax instead.

Querying ClickHouse with a pxf:// external table

Create an external table over the ClickHouse table, then query it like any other WarehousePG table:

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

```
-- in
WarehousePG
CREATE EXTERNAL TABLE ch_events_ext
(
    id         text,
    event_type text,
    event_time timestamp,
    value
float8
)
LOCATION ('pxf://<database>.<table>?
PROFILE=Jdbc&SERVER=clickhouse')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');

SELECT * FROM
ch_events_ext;
```

> **Note**
>
> Leave out `UNIQUE` and `PRIMARY KEY` constraints. WarehousePG doesn't support them on external tables and rejects the `CREATE EXTERNAL TABLE` statement if they're present.

Write data back to ClickHouse with a separate writable external table at the same `LOCATION`, using `FORMATTER='pxfwritable_export'`:

```
-- in
WarehousePG
CREATE WRITABLE EXTERNAL TABLE ch_events_write_ext
(
    id         text,
    event_type text,
    event_time timestamp,
    value
float8
)
LOCATION ('pxf://<database>.<table>?
PROFILE=Jdbc&SERVER=clickhouse')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_export');

INSERT INTO ch_events_write_ext VALUES ('evt-1', 'purchase', now(),
99.99);
```

For general PXF and JDBC server configuration, including connection pooling and other JDBC settings, see Connecting to a SQL database over JDBC. For settings specific to the ClickHouse driver itself, see ClickHouse's JDBC driver documentation.

Querying ClickHouse with pxf_fdw

Query the same server through standard `CREATE FOREIGN TABLE` syntax instead, using the `jdbc_pxf_fdw` foreign data wrapper:

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

```
-- in
WarehousePG
CREATE EXTENSION
pxf_fdw;

CREATE SERVER
clickhouse_svr
    FOREIGN DATA WRAPPER jdbc_pxf_fdw
    OPTIONS (config 'clickhouse');

CREATE USER MAPPING FOR CURRENT_USER SERVER
clickhouse_svr;

CREATE FOREIGN TABLE ch_events_ft
(
    id         text,
    event_type text,
    event_time timestamp,
    value
float8
)
SERVER
clickhouse_svr
OPTIONS (resource '<database>.
<table>');

SELECT * FROM ch_events_ft;
```

A `pxf_fdw` foreign table supports both reading and writing, unlike a `pxf://` external table, which needs a separate `READABLE` / `WRITABLE` definition for each direction. Insert directly into the same foreign table:

```
-- in
WarehousePG
INSERT INTO ch_events_ft VALUES ('evt-1', 'purchase', now(),
99.99);
```

See Querying data with the PXF foreign data wrapperfor the full syntax, including per-user credentials in the user mapping and standard FDW grants.

Confirming parallel execution

Confirm parallel execution with `EXPLAIN`. Both interfaces read through the same PXF service, with each segment's PXF instance querying ClickHouse independently:

```
EXPLAIN SELECT * FROM
ch_events_ext;
```

output

```
                        QUERY PLAN
----------------------------------------------------------
 Gather Motion 4:1  (slice1; segments: 4)
   ->  Foreign Scan on ch_events_ext
 Optimizer: Postgres query optimizer
```

A `Foreign Scan` under a `Gather Motion` with `segments: <N>` matching your WHPG segment count confirms every segment is reading its own share of the ClickHouse table in parallel, rather than one segment pulling all the data.

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

Troubleshooting

- **Connection refused from a segment host:** confirm `listen_host` in ClickHouse's configuration covers the interface that segment reaches it on, and that the segment can resolve and reach the ClickHouse host over the network.
- **Authentication errors even with correct credentials:** check `jdbc.url` for a `?user=` or `?password=` parameter. Either one overrides `jdbc.user` / `jdbc.password` from `jdbc-site.xml` and is a common source of confusing auth failures.
- **Write fails with a ClickHouse access-denied error:** the ClickHouse user needs `INSERT` granted, not just `SELECT`. See Preparing ClickHouse for incoming connections.
- `pxf_fdw` **writes fail with a** `NullPointerException`**:** upgrade the driver jar to 0.10.0 or later.
- **PXF stops responding after a large read:** the PXF JVM can run out of heap and terminate itself rather than throttling. Run `pxf restart` on the affected host, or tune the JVM heap as described in Managing the PXF cluster if your ClickHouse tables are large.

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.

8 Connecting AI agents to ClickHouse

Connect an AI agent to your ClickHouse server with `edb-clickhouse-mcp` to list databases, describe schemas, and run queries in response to plain-language questions instead of requiring hand-written SQL.

`edb-clickhouse-mcp` is EDB's packaging of the open-source ClickHouse MCP server for Model Context Protocol (MCP), an open standard that lets AI agents call external tools over a common interface. EDB provides RPM, DEB, and container packaging, and a set of opt-in diagnostic tools covered in Understanding the available tools.

> **Before you start**
>
> This page assumes you already have an MCP-compatible AI client, such as Claude Desktop, Claude Code, or Cursor.

Understanding the integration

Three separate pieces come together to answer a natural-language question against ClickHouse, and each one can run on its own machine and its own platform:

- **AI client**: Claude Desktop, Claude Code, Cursor, or another MCP-compatible client, running on macOS, Windows, or Linux. Configure it in the client's own configuration file, covered in Connecting an AI client.
- `edb-clickhouse-mcp` : the RPM package on a RHEL-compatible Linux host, the DEB package on a Debian 13 host, or the container image on macOS, Windows, or Linux. Configure it in `mcp.env` for the packages, or with environment variables for the container, covered in Configuration parameters.
- **ClickHouse server**: wherever your ClickHouse cluster or node already runs, in Docker, on Linux, or on Kubernetes. Set it up as described in Installing and upgrading.

Setup workflow

Work through these steps in order:

1. Install `edb-clickhouse-mcp` as an RPM package, a DEB package, or a container image, on whichever machine you want it to run, not necessarily the same one as your ClickHouse server.
2. Edit `edb-clickhouse-mcp` 's configuration to point it at your ClickHouse server and give it credentials to authenticate.
3. Review the available tools it registers, and turn on the EDB diagnostic tools if you want them.
4. Connect your AI client to `edb-clickhouse-mcp` .
5. Ask it a question in natural language.

Installing edb-clickhouse-mcp

Install `edb-clickhouse-mcp` as an RPM package, a DEB package, or a container image, all available for x86_64 and aarch64 (amd64 and arm64 for the DEB package). Install it on whichever machine you want the MCP server to run on, as long as it can reach your ClickHouse node over the network. `edb-clickhouse-mcp` connects to a single ClickHouse endpoint, not a cluster as a whole, as described in Configuration parameters.

Installing the RPM or DEB package

The RPM package requires a RHEL-compatible Linux host, and the DEB package requires a Debian 13 host.

1. Set up the EDB repository and install the package, using the same `clickhouse` subscription plan as the ClickHouse server and client packages:

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

```
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-
mcp
```

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=clickhouse
curl -1sSLf
"https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_SUBSCRIPTION_PLAN/setup.deb.sh" | sudo -E
bash
sudo apt-get install -y edb-clickhouse-mcp
```

Replace `<your-token>` with the token you received when you registered for the EDB subscription. This command installs a self-contained virtual environment under `/opt/edb/clickhouse-mcp`, with `edb-clickhouse-mcp` as the only command added to `PATH`, and connection settings in `/etc/edb-clickhouse-mcp/mcp.env`.

1. Point it at your ClickHouse node. Edit `/etc/edb-clickhouse-mcp/mcp.env` and set at least `CLICKHOUSE_HOST`, `CLICKHOUSE_USER`, and `CLICKHOUSE_PASSWORD`, described in Configuration parameters.

Installing the container image

This image runs on macOS, Windows, or Linux, through Docker or Podman.

1. Log in to the EDB container registry and pull the image, using the same `clickhouse` repository as the ClickHouse server image:

   ```
   export EDB_SUBSCRIPTION_TOKEN=<your-token>
   echo "$EDB_SUBSCRIPTION_TOKEN" | docker login docker.enterprisedb.com \
     --username clickhouse \
     --password-stdin
   docker pull docker.enterprisedb.com/clickhouse/edb-clickhouse-mcp:0.4.1-
   1edb1
   ```

   Replace `<your-token>` with the token you received when you registered for the EDB subscription.

2. Run it:

   ```
   docker run --rm -i
   \
     -e CLICKHOUSE_HOST=<clickhouse-host>
   \
     -e CLICKHOUSE_USER=<clickhouse-user>
   \
     -e CLICKHOUSE_PASSWORD=<clickhouse-password> \
     docker.enterprisedb.com/clickhouse/edb-clickhouse-mcp:0.4.1-
   1edb1
   ```

   Where `<clickhouse-host>`, `<clickhouse-user>`, and `<clickhouse-password>` are your ClickHouse connection details, described in Configuration parameters.

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

Configuration parameters

These parameters configure the connection to your ClickHouse server, set either in `/etc/edb-clickhouse-mcp/mcp.env` for the RPM or DEB package, or passed directly to the container:

Variable	Required	Description
`CLICKHOUSE_HOST`	Yes	Hostname of your ClickHouse server.
`CLICKHOUSE_USER`	Yes	Username for authentication. Grant this user only the privileges the agent needs, and avoid a default or administrative user.
`CLICKHOUSE_PASSWORD`	Yes	Password for authentication.
`CLICKHOUSE_PORT`	No	Defaults to `8443` when `CLICKHOUSE_SECURE` is `true`, `8123` otherwise.
`CLICKHOUSE_DATABASE`	No	Default database to connect to.
`CLICKHOUSE_SECURE`	No	Enables HTTPS. Defaults to `true`.
`CLICKHOUSE_ROLE`	No	Role to use for authentication, if your user requires one.
`CLICKHOUSE_ALLOW_WRITE_ACCESS`	No	Allows structure and data changes (DDL and DML statements). Defaults to `false`, so queries are read-only.
`CLICKHOUSE_ALLOW_DROP`	No	Allows destructive operations (`DROP TABLE`, `DROP DATABASE`, `DROP VIEW`, `DROP DICTIONARY`, `TRUNCATE TABLE`), on top of `CLICKHOUSE_ALLOW_WRITE_ACCESS`. Defaults to `false`.

Secret values, such as `CLICKHOUSE_PASSWORD`, and common `key=value` secret patterns are redacted from every tool's output and error messages before they reach the client or the logs.

> **Connecting to a cluster node**
>
> `edb-clickhouse-mcp` connects to one ClickHouse endpoint, set by `CLICKHOUSE_HOST`. On a multi-node cluster, `edb_cluster_health` and the other EDB diagnostic tools report on that node only, not the whole cluster.

> **Applying configuration changes**
>
> Restart `edb-clickhouse-mcp` after changing any of these variables, or any of the diagnostic tool settings in Tuning the diagnostic tools, since it only reads them at startup. For the RPM or DEB package, run `sudo systemctl restart edb-clickhouse-mcp`. For the container, the client launches a fresh container each time it connects, so edit the environment variables in your AI client's own configuration file and restart the client to apply them.

Understanding the available tools

Review the tools `edb-clickhouse-mcp` registers before connecting your AI client, so you know what an agent can do once it's connected. When a client connects, it discovers the names and descriptions of the tools already registered, and the AI model behind it uses that information to decide when to call each one.

Default tools

These four tools register by default, regardless of whether the server is installed from a package or the container image:

Tool	Description
`run_query`	Runs a SQL query against your ClickHouse server. Read-only unless write access is explicitly enabled.

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

Tool	Description
`list_databases`	Lists all databases on the server.
`list_tables`	Lists tables in a database, with filtering and pagination.
`edb_version`	Reports the EDB build identity as `<package version>+<edb release>`.

EDB diagnostic tools

Set `EDB_MCP_ENABLED=true`, either in `/etc/edb-clickhouse-mcp/mcp.env` for the RPM or DEB package, or passed directly to the container, to register four additional read-only tools that give an agent operational visibility into the server beyond running queries:

Tool	Description
`edb_cluster_health`	Server version, uptime, database count, a sample of running queries, per-disk free space, and replication delay.
`edb_system_table_snapshot`	Every column of one allowlisted `system` table, such as `settings`, `metrics`, or `replication_queue`, for information the health snapshot doesn't cover.
`edb_query_stats`	Aggregated statistics from `system.query_log` over a trailing window: completed and failed counts, duration percentiles, rows and bytes read, and a per-user breakdown. Never returns query text.
`edb_resource_metrics`	Aggregated resource usage from `system.metric_log` over a trailing window: concurrent queries, memory, CPU time, and disk I/O.

Tuning the diagnostic tools

Set these variables the same way as the connection settings, either in `mcp.env` for the RPM or DEB package, or passed directly to the container. Restart `edb-clickhouse-mcp` to apply changes, as described in Configuration parameters.

Variable	Default	Description
`EDB_MCP_ENABLED`	`false`	Toggle for the EDB diagnostic tools.
`EDB_HEALTH_MAX_QUERIES`	`50`	Running queries sampled per `edb_cluster_health` call, from 1 to 1000.
`EDB_HEALTH_INCLUDE_QUERY_TEXT`	`false`	SQL text of running queries in the health snapshot. Query text can carry sensitive values, so it stays off unless you opt in.
`EDB_SNAPSHOT_MAX_ROWS`	`200`	Row cap for `edb_system_table_snapshot`, from 1 to 10000.
`EDB_METRICS_WINDOW_MINUTES`	`15`	Trailing window for `edb_query_stats` and `edb_resource_metrics`, from 1 to 1440 minutes. A wider window costs a larger scan.
`EDB_METRICS_BREAKDOWN_ROWS`	`10`	Cap on the error and per-user breakdown lists in `edb_query_stats`, from 1 to 100.

Connecting an AI client

Connect your AI client to `edb-clickhouse-mcp` so it can call the tools registered on it. Where the client runs relative to `edb-clickhouse-mcp`, same machine or different, determines the transport, client configuration, and authentication, as shown here:

	Same machine	Different machine
Transport	`stdio` (default)	`http` or `sse`
Client configuration	Launches `edb-clickhouse-mcp` as a subprocess	Connects to a URL over the network

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

	Same machine	Different machine
Authentication	None needed	Required

Setting up the remote server

If `edb-clickhouse-mcp` runs on a different machine than your AI client, configure it for network access first, then pick**Connect to a remote server** in Editing the Claude Desktop configuration. If it runs on the same machine as the client, skip this section.

Set the following, either in `/etc/edb-clickhouse-mcp/mcp.env` for the RPM or DEB package, or passed directly to the container:

1. Set `CLICKHOUSE_MCP_SERVER_TRANSPORT` to `http` or `sse`, and `CLICKHOUSE_MCP_BIND_HOST=0.0.0.0`. `CLICKHOUSE_MCP_BIND_HOST` otherwise defaults to `127.0.0.1` and only accepts connections from the same host.

2. Set up authentication. Startup fails without it, for both transports. Pick one mode:

Mode	Setting	When to use
Static bearer token	`CLICKHOUSE_MCP_AUTH_TOKEN`	Simple or internal deployments. Generate a token with `uuidgen` or `openssl rand -hex 32`.
OAuth / OpenID Connect (OIDC)	`FASTMCP_SERVER_AUTH` plus provider-specific `FASTMCP_SERVER_AUTH_*` variables	Production deployments behind Azure Entra, Google, GitHub, WorkOS, or another FastMCP-supported provider.
Disabled	`CLICKHOUSE_MCP_AUTH_DISABLED=true`	Local development only. Avoid disabling authentication on a server reachable over a network.

3. Start `edb-clickhouse-mcp`. For the RPM or DEB package, start the packaged systemd service:

```
sudo systemctl enable --now edb-clickhouse-
mcp
```

   For the container, pass the settings from the previous two steps as `-e` flags, map the bind port, and run it detached:

```
docker run -d
\
  -p 8000:8000 \
  -e CLICKHOUSE_MCP_SERVER_TRANSPORT=http \
  -e CLICKHOUSE_MCP_BIND_HOST=0.0.0.0 \
  -e CLICKHOUSE_MCP_AUTH_TOKEN=<token>
\
  -e CLICKHOUSE_HOST=<clickhouse-host>
\
  -e CLICKHOUSE_USER=<clickhouse-user>
\
  -e CLICKHOUSE_PASSWORD=<clickhouse-password> \
  docker.enterprisedb.com/clickhouse/edb-clickhouse-mcp:0.4.1-
1edb1
```

4. Also open the corresponding inbound rule for the bind port, such as an EC2 security group or other cloud firewall rule, or your network firewall otherwise.

5. Check the unauthenticated `/health` endpoint, which reports server and connectivity status for load balancers and orchestrator probes:

```
curl http://<mcp-host>:8000/health
```

   It returns `200 OK` when the server can reach ClickHouse, and `503` otherwise, without version or error details in the body.

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

Editing the Claude Desktop configuration

The following steps use Claude Desktop as an example. Other MCP-compatible clients, such as Claude Code or Cursor, need the same configuration, added to their own configuration file instead.

1. Open **Settings > Developer > Edit Config** in Claude Desktop to reach `claude_desktop_config.json` directly:

   - macOS: `~/Library/Application Support/Claude/claude_desktop_config.json`
   - Windows: `%APPDATA%/Claude/claude_desktop_config.json`

2. Add one of the following configurations to its `mcpServers` object, alongside any servers already listed there, rather than replacing the file. Pick the one that matches your setup.

   > **Tip**
   >
   > Wherever a configuration uses `"command": "docker"`, replace `"docker"` with its full path if Claude Desktop can't find it, since it doesn't always inherit your shell's `PATH`.

   - **Point at your own ClickHouse server**: launches `edb-clickhouse-mcp` as a subprocess on the same machine as the client, over `stdio`, pointed at your ClickHouse instance. This setup covers the everyday case for a laptop or workstation running both the client and the server. The command depends on how you installed `edb-clickhouse-mcp`:

     If you installed the RPM or DEB package, on a Linux host:

     ```
     {
       "mcpServers": {
         "clickhouse": {
           "command": "/usr/bin/edb-clickhouse-mcp",
           "env": {
             "CLICKHOUSE_HOST": "<clickhouse-host>",
             "CLICKHOUSE_USER": "<clickhouse-user>",
             "CLICKHOUSE_PASSWORD": "<clickhouse-password>"
           }
         }
       }
     }
     ```

     If you installed the Docker container image:

     ```
     {
       "mcpServers": {
         "clickhouse": {
           "command": "docker",
           "args": [
             "run", "--rm", "-i",
             "-e", "CLICKHOUSE_HOST=<clickhouse-host>",
             "-e", "CLICKHOUSE_USER=<clickhouse-user>",
             "-e", "CLICKHOUSE_PASSWORD=<clickhouse-password>",
             "docker.enterprisedb.com/clickhouse/edb-clickhouse-mcp:0.4.1-1edb1"
           ]
         }
       }
     }
     ```

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

- **Try the ClickHouse SQL Playground**: the same subprocess setup as pointing at your own ClickHouse server, but aimed at ClickHouse's public, read-only demo instance instead. Use this option to confirm the configuration mechanism works, with no ClickHouse instance of your own needed, before pointing it at real data.

  If you installed the RPM or DEB package, on a Linux host:

```
{
  "mcpServers": {
    "clickhouse": {
      "command": "/usr/bin/edb-clickhouse-mcp",
      "env": {
        "CLICKHOUSE_HOST": "sql-clickhouse.clickhouse.com",
        "CLICKHOUSE_PORT": "8443",
        "CLICKHOUSE_USER": "demo",
        "CLICKHOUSE_PASSWORD": ""
      }
    }
  }
}
```

  If you installed the Docker container image:

```
{
  "mcpServers": {
    "clickhouse": {
      "command": "docker",
      "args": [
        "run", "--rm", "-i",
        "-e", "CLICKHOUSE_HOST=sql-clickhouse.clickhouse.com",
        "-e", "CLICKHOUSE_PORT=8443",
        "-e", "CLICKHOUSE_USER=demo",
        "-e", "CLICKHOUSE_PASSWORD=",
        "docker.enterprisedb.com/clickhouse/edb-clickhouse-mcp:0.4.1-1edb1"
      ]
    }
  }
}
```

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

- **Connect to a remote server**: when `edb-clickhouse-mcp` runs on a different machine than the client, such as your ClickHouse server. Claude Desktop's `mcpServers` only supports launching local subprocesses, not connecting to a URL directly, so this configuration launches `mcp-remote` as a bridge instead, which forwards requests over the network with a bearer token. Requires the remote server setup above.

```
{
  "mcpServers": {
    "clickhouse": {
      "command": "npx",
      "args": [
        "-y",
        "mcp-remote",
        "http://<mcp-host>:8000/mcp",
        "--header",
        "Authorization: Bearer
<token>"
      ]
    }
  }
}
```

  Where `<mcp-host>` is the host running `edb-clickhouse-mcp`, and `<token>` is the value of `CLICKHOUSE_MCP_AUTH_TOKEN`. `-y` lets `npx` install `mcp-remote` without prompting, since Claude Desktop launches it with no terminal attached to answer one.

3. Restart Claude Desktop to apply the configuration.

Querying ClickHouse in natural language

Ask the AI agent a plain-language question once your client connects to `edb-clickhouse-mcp`:

1. When the client connects to `edb-clickhouse-mcp`, it fetches the full list of registered tools, along with each tool's description.
2. Ask a question, for example "What databases are on my ClickHouse server?"
3. The agent matches your question against the tool descriptions and calls the relevant one itself. For this question, that's `list_databases`.
4. `edb-clickhouse-mcp` runs the equivalent of `SHOW DATABASES` against your server and returns the result.
5. The agent turns that into an answer, such as "Your server has these databases: default, system, analytics." Most clients, including Claude Desktop, show somewhere in the conversation that a tool was called.

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

9 Monitoring Postgres with pg_stat_ch

Capture per-query execution telemetry from Postgres and stream it to ClickHouse in real time using pg_stat_ch, EDB's build of the open-source Postgres extension of the same name. pg_stat_ch instruments Postgres query execution and error reporting through a background worker, capturing every statement, including `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and utility statements such as `COPY` and `VACUUM`, with no changes to application code, buffering events in shared memory and flushing them to ClickHouse over the native protocol, either on a fixed interval or on demand.

Keeping enough per-query telemetry, at enough detail, for long enough to debug a problem gets expensive fast inside Postgres itself. Instead of aggregating statistics inside Postgres, pg_stat_ch exports raw events and lets ClickHouse handle the analytical workload through materialized views, opening up investigations that are impractical with aggregated statistics alone:

- **Root-causing slow queries fast**, down to the exact SQL, application, and parameters responsible, without needing to reproduce the issue.
- **Diagnosing buffer cache and disk-spill pressure**, spotting queries whose working set no longer fits in memory or that spill sorts and hashes to disk.
- **Tracking WAL footprint before it becomes replication lag**, at the individual query level rather than after the fact.
- **Telling whether JIT compilation is helping or hurting** a given query, rather than assuming.
- **Attributing load by application, user, and database**, useful for chargeback and spotting a noisy neighbor.
- **Catching errors outside normal query paths**, including failed `DDL`, `COPY`, and `VACUUM` statements, not just DML.
- **Spotting parallel query starvation**, when queries plan for parallelism but don't get the workers.

Installing pg_stat_ch

Configure ClickHouse first, then install and configure pg_stat_ch on your Postgres installation.

Configuring ClickHouse

Set up ClickHouse before configuring Postgres.

> **Note**
>
> This procedure targets a single ClickHouse node. Data isn't replicated or sharded automatically, so the node you provision here is also the one pg_stat_ch must point at later, and the one you query back. For a multi-node cluster, see the note on `pg_stat_ch.clickhouse_host` under Installing and configuring pg_stat_ch.

1. Provision the `pg_stat_ch` schema before configuring Postgres. Apply the migration from the upstream pg_stat_ch repository, which creates the `pg_stat_ch` database along with the `events_raw` table and its materialized views. The file is a goose migration with an `Up` and a `Down` section, so extract only the `Up` section:

   ```
   curl -O
   https://raw.githubusercontent.com/ClickHouse/pg_stat_ch/main/schema/migrations/20260519000001_create_ini
   l_schema.sql
   awk '/-- \+goose Down/{exit} {print}' 20260519000001_create_initial_schema.sql >
   up_only.sql
   ```

2. Apply it to ClickHouse:

   ```
   clickhouse-client --host <your-clickhouse-host> --multiquery <
   up_only.sql
   ```

   Replace `<your-clickhouse-host>` with your server's hostname or IP, adding `--port`, `--user`, and `--password` as needed. Check `schema/migrations` for later files and apply those the same way, in order.

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

3. Verify the schema was created:

```
-- in ClickHouse
SHOW TABLES FROM
pg_stat_ch;
```

4. Create a dedicated user for pg_stat_ch and grant it access to `events_raw`:

```
-- in ClickHouse
CREATE USER pg_stat_ch IDENTIFIED WITH sha256_password BY 'your-password';
GRANT INSERT ON pg_stat_ch.events_raw TO
pg_stat_ch;
GRANT SELECT ON pg_stat_ch.events_raw TO
pg_stat_ch;
```

The `SELECT` grant is required too, ClickHouse runs the materialized views as the inserting user on every insert.

Installing and configuring pg_stat_ch

Run the following steps on your Postgres instance. EDB distributes pg_stat_ch as RPM packages for Rocky Linux 9 on x86-64 and arm64, and as DEB packages for Debian 12 (bookworm) and Debian 13 (trixie) on amd64 and arm64. Packages cover community Postgres, EDB Postgres Extended Server, and EDB Postgres Advanced Server (EPAS), for Postgres versions 16 or later. Debian 13 packages cover community Postgres only, because EDB publishes no EPAS or EDB Postgres Extended Server packages for Debian 13.

1. Set up the EDB repository for ClickHouse and install the package for your Postgres distribution and version:

```
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 <package-
name>
```

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=clickhouse
curl -1sSLf
"https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_SUBSCRIPTION_PLAN/setup.deb.sh" | sudo -E
bash
sudo apt-get install -y <package-name>
```

```
Where:
- `<your-token>` is the token you received when you registered for the EDB subscription.
- `<package-name>` is the package for your Postgres distribution and version, where `<version>` is the
Postgres major version (16, 17, or 18):

  | Postgres distribution | Package name |
  |-----------------------|--------------|
  | Community Postgres | `edb-pg<version>-pg-stat-ch` |
  | EDB Postgres Advanced Server | `edb-as<version>-pg-stat-ch` |
  | EDB Postgres Extended Server | `edb-postgresextended-<version>-pg-stat-ch` |

  For example, [edb-pg18-pg-stat-ch]
(https://cloudsmith.io/~enterprisedb/repos/clickhouse/packages/detail/rpm/edb-pg18-pg-stat-ch/0.4.0-
1edb2.el9/a=aarch64;d=el%252F9;t=binary/) installs pg_stat_ch for Postgres 18.
```

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

1. Edit `postgresql.conf` and append pg_stat_ch to the list of shared libraries using a comma separator:

```
# postgresql.conf
shared_preload_libraries = '<other_libraries>,pg_stat_ch'
```

2. Still from `postgresql.conf`, point pg_stat_ch at your ClickHouse server:

```
# postgresql.conf
pg_stat_ch.clickhouse_host = '<clickhouse-host>'
pg_stat_ch.clickhouse_port = 9000
pg_stat_ch.clickhouse_database = 'pg_stat_ch'
pg_stat_ch.clickhouse_user = '<clickhouse-user>'
pg_stat_ch.clickhouse_password = '<clickhouse-password>'
pg_stat_ch.enabled = on
```

Where:

- `clickhouse_host` is the hostname or IP address of the ClickHouse server to send telemetry to, the same node you provisioned in Configuring ClickHouse. Connect to this same host when querying the data back.
- `clickhouse_port` is ClickHouse's native protocol port. Defaults to `9000`.
- `clickhouse_database` is a database on the ClickHouse server. It's where `events_raw` and its materialized views live. Defaults to `pg_stat_ch`, the database created in Configuring ClickHouse. If you provisioned the schema under a different database name, update this setting to match.
- `clickhouse_user` and `clickhouse_password` are credentials for the ClickHouse user created in Configuring ClickHouse, with `INSERT` and `SELECT` on `events_raw`.
- `enabled` turns telemetry collection on or off without unloading the extension. Unlike the connection settings, this setting takes effect on a reload (`pg_ctl reload` or `SELECT pg_reload_conf();`), no restart needed.

See the configuration reference from the pg_stat_ch documentation for the complete list of configuration parameters, including queue sizing, batching, and TLS options.

> **Note**
>
> `pg_stat_ch.clickhouse_host` takes a single host, there's no cluster-aware setting. If your ClickHouse deployment is a multi-node cluster, point it at a single entry point, a specific node, a `Distributed` table target, or a load balancer in front of the cluster, rather than listing multiple hosts.
>
> The migration in Configuring ClickHouse creates plain `MergeTree` and `AggregatingMergeTree` tables, so data only lives on whichever node received it, it isn't replicated or sharded automatically. To make it visible from any node in a cluster, you'd need to modify that schema yourself: replace those engines with `ReplicatedMergeTree` and `ReplicatedAggregatingMergeTree` for replication, or add `Distributed` tables on top for sharding, backed by a configured ClickHouse cluster and Keeper. See Architecture for these concepts.

3. Restart Postgres for both changes to take effect, then create the extension and verify the EDB build:

```
-- in
Postgres
CREATE EXTENSION
pg_stat_ch;
SELECT
pg_stat_ch_version();
```

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

4. Verify telemetry is flowing:

```
-- in
Postgres
SELECT * FROM
pg_stat_ch_stats();
```

`pg_stat_ch_stats()` returns counters for events queued, exported, and failed. pg_stat_ch captures activity across every database on this Postgres instance, not only the one where you ran `CREATE EXTENSION`, the `db_name` column on each event records which database it came from. See the SQL functions reference from the pg_stat_ch documentation for the complete list. Trigger an immediate flush instead of waiting for the next scheduled batch, or reset the counters:

```
-- in
Postgres
SELECT
pg_stat_ch_flush();
SELECT
pg_stat_ch_reset();
```

5. Check the background worker is running:

```
-- in
Postgres
SELECT pid, backend_type, state,
wait_event
FROM
pg_stat_activity
WHERE backend_type = 'pg_stat_ch exporter';
```

If it doesn't appear, check the Postgres log for errors, under the data directory's `log` subdirectory by default, or via `journalctl -u <service-name>` if Postgres runs under systemd.

Understanding the ClickHouse schema

pg_stat_ch exports telemetry to ClickHouse in two layers: raw events in `events_raw`, and materialized views that pre-aggregate those events for faster dashboard queries.

Understanding events_raw

Every statement captured on the Postgres side lands as one row in `pg_stat_ch.events_raw` in ClickHouse. Each row carries:

- Identity fields: database, user, application name, client address
- Timing: start timestamp and duration
- Row counts
- Buffer cache activity: shared, local, and temp block hits, reads, and writes
- WAL volume
- CPU and JIT compilation time
- Error details, when the statement failed

Other columns such as `instance_ubid`, `server_role`, and `region` are unpopulated and stay empty for self-hosted deployments. See the events schema reference from the pg_stat_ch documentation for every column, though it documents an older revision of the schema and uses different column names than the ones on this page.

You can query the raw events directly for ad hoc investigation:

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

```
-- in ClickHouse
SELECT ts, db_operation, duration_us, rows,
query_text
FROM
pg_stat_ch.events_raw
ORDER BY ts DESC
LIMIT 20;
```

This example pulls the 20 most recent statements, with when each ran, its command type, how long it took, how many rows it touched, and its query text.

> **Note**
>
> pg_stat_ch doesn't identify which tables a query references, so it can't answer which queries accessed a given table, the way plan-based tools can. I/O timing fields, such as `shared_blk_read_time_us`, require `track_io_timing` enabled on the Postgres instance. Without it, these fields report zero.

Query text is truncated at a configurable limit (2 KB by default) to bound event size. For longer queries, join on `query_id` against `pg_stat_statements` for the full text.

Querying pre-aggregated views

Query the materialized views layered on top of `events_raw` for dashboards, rather than recomputing percentiles and rollups from raw events on every query, an approach that grows costly at volume:

- A per-query, time-bucketed view (`query_stats_5m`), giving call counts, average duration, and latency percentiles by query and command type
- A per-application and per-user view (`db_app_user_1m`), for identifying which application or user is generating the most load or errors
- A recent-errors view (`errors_recent`), filtered to failed statements for incident investigation

See the materialized views reference from the pg_stat_ch documentation for additional views and analytics patterns.

For example, this query ranks your ten worst-performing query and command type combinations by p99 latency, using the matching `-Merge` function to read the aggregate columns in `query_stats_5m`:

```
-- in ClickHouse
SELECT
    query_id,
    db_operation,
    countMerge(calls_state) AS
calls,
    round(sumMerge(duration_sum_state) / countMerge(calls_state) / 1000, 2) AS avg_ms,
    round(quantilesTDigestMerge(0.95, 0.99)(duration_q_state)[1] / 1000, 2) AS p95_ms,
    round(quantilesTDigestMerge(0.95, 0.99)(duration_q_state)[2] / 1000, 2) AS
p99_ms
FROM pg_stat_ch.query_stats_5m
GROUP BY query_id, db_operation
ORDER BY p99_ms
DESC
LIMIT 10;
```

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

Monitoring common scenarios

Most performance questions follow the same shape: aggregate from a materialized view to spot the outlier, then drill into `events_raw` for specifics. The monitoring queries guide from the pg_stat_ch documentation has further recipes along the same lines.

Finding slow queries

Combine an aggregated view with a drill-down into `events_raw` to go from a slow `query_id` to the exact SQL and application responsible. Take the slowest `query_id` from the p99 ranking query, then pull the matching raw events:

```
-- in ClickHouse
SELECT app, query_text,
duration_us
FROM
pg_stat_ch.events_raw
WHERE query_id = <id from previous
query>
ORDER BY ts DESC
LIMIT 5;
```

> **Tip**
>
> This approach, which combines aggregating to find an outlier, then drilling into `events_raw` for detail, extends to other cases as well: buffer cache efficiency, using `shared_blks_hit` and `shared_blks_read`, WAL generation, using `wal_bytes`, and error tracking, using `err_sqlstate` and `err_message`.

Diagnosing cache and disk-spill pressure

Find individual executions with the most disk reads:

```
-- in ClickHouse
SELECT

ts,
    query_id,
    shared_blks_read,

shared_blks_hit,
    round(100 * shared_blks_read / (shared_blks_hit + shared_blks_read), 2) AS
miss_pct,
    temp_blks_written,
    duration_us / 1000 AS
ms,
    substring(query_text, 1, 100) AS
query_preview
FROM
pg_stat_ch.events_raw
WHERE shared_blks_read >
100
  AND ts > now() - INTERVAL 1 HOUR
ORDER BY shared_blks_read
DESC
LIMIT 20;
```

A high `miss_pct` on a row points to a cache miss, the working set no longer fits in memory. A non-zero `temp_blks_written` on that same row points to disk spill instead, a sort or hash operation exceeded `work_mem` and wrote temp file blocks to disk.

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

Tracking WAL footprint

Track WAL generation per minute to catch a write-heavy query before it becomes replication lag:

```
-- in ClickHouse
SELECT
    toStartOfMinute(ts) AS
bucket,
    sum(wal_bytes) AS
total_wal_bytes,
    round(sum(wal_bytes) / 1048576, 2) AS wal_mb,
    sum(wal_fpi) AS
full_page_images
FROM
pg_stat_ch.events_raw
WHERE db_operation IN ('INSERT', 'UPDATE',
'DELETE')
  AND ts > now() - INTERVAL 24 HOUR
GROUP BY
bucket
ORDER BY bucket;
```

Analyzing JIT overhead

Compare JIT compilation time against total execution time to determine whether it improves performance:

```
-- in ClickHouse
SELECT
    query_id,
    count() AS
executions,
    round(avg(duration_us) / 1000, 2) AS avg_total_ms,
    round(avg(jit_generation_time_us + jit_inlining_time_us +
              jit_optimization_time_us + jit_emission_time_us) / 1000, 2) AS
avg_jit_ms,
    round(100 * avg(jit_generation_time_us + jit_inlining_time_us +
                    jit_optimization_time_us + jit_emission_time_us)
/
          greatest(avg(duration_us), 1), 1) AS
jit_pct_of_total
FROM
pg_stat_ch.events_raw
WHERE jit_functions > 0
  AND ts > now() - INTERVAL 24 HOUR
GROUP BY
query_id
HAVING avg_jit_ms > 10
ORDER BY jit_pct_of_total
DESC
LIMIT 10;
```

If `jit_pct_of_total` runs high for a frequently run query, consider raising `jit_above_cost` to skip JIT for it.

Attributing load by application

Rank applications by total query time to find the heaviest consumer:

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

```
-- in ClickHouse
SELECT
    app,
    countMerge(calls_state) AS
total_queries,
    round(sumMerge(duration_sum_state) / 1000000, 2) AS
total_seconds,
    round(quantilesTDigestMerge(0.95, 0.99)(duration_q_state)[2] / 1000, 2) AS p99_ms,
    sumMerge(errors_sum_state) AS
errors
FROM pg_stat_ch.db_app_user_1m
WHERE bucket >= now() - INTERVAL 24
HOUR
GROUP BY
app
ORDER BY total_seconds DESC;
```

Tracking errors

Break failed statements down by `SQLSTATE` to spot the most common failure:

```
-- in ClickHouse
SELECT
    err_sqlstate,
    count() AS errors,
    uniq(query_id) AS
unique_queries,
    any(err_message) AS
sample_message
FROM
pg_stat_ch.events_raw
WHERE err_elevel >= 21
  AND ts > now() - INTERVAL 24 HOUR
GROUP BY err_sqlstate
ORDER BY errors
DESC;
```

`err_elevel` encodes Postgres severity as a number, documented in the events schema reference from the pg_stat_ch documentation: `19` is `WARNING`, `21` is `ERROR`, `22` is `FATAL`, and `23` is `PANIC`. Filtering on `err_elevel >= 21` excludes warnings and keeps only actual errors.

Spotting parallel query starvation

Find queries that planned for parallelism but didn't get the workers, available on Postgres 18 and later:

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

```
-- in ClickHouse
SELECT
    query_id,
    db_operation,
    count() AS
executions,
    round(avg(parallel_workers_planned), 1) AS avg_planned,
    round(avg(parallel_workers_launched), 1) AS avg_launched,
    round(avg(parallel_workers_planned - parallel_workers_launched), 1) AS avg_missed
FROM
pg_stat_ch.events_raw
WHERE parallel_workers_planned >
parallel_workers_launched
  AND ts > now() - INTERVAL 24 HOUR
GROUP BY query_id, db_operation
ORDER BY avg_missed DESC
LIMIT 10;
```

If `avg_missed` stays high, consider raising `max_parallel_workers` or `max_worker_processes`.

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