# Snowflake Connector for MySQL

## Introduction

The Snowflake Connector for MySQL allows you to:

* Load data into Snowflake from a MySQL database.
* Configure replication so that changes in your MySQL database are replicated to Snowflake.

To handle connections between Snowflake and MySQL, the connector uses an agent. The agent is distributed as a Docker image. The agent is run within your network and is used to push data into your Snowflake account.

**Note::**

The Snowflake Connector for MySQL requires exactly one instance of the agent application to be running at all times.

The ongoing incremental updates use the Change Data Capture (CDC) technique that captures changes performed on the source database. The changes include INSERT, UPDATE, and DELETE operations, which are automatically replicated on the destination database in Snowflake.

## Multiple application instances

You can install multiple instances of Snowflake Connector for MySQL on your Snowflake account. For more information, see [Optional: Installing multiple instances of Snowflake Connector for MySQL](https://other-docs.snowflake.com/en/connectors/mysql6/install-snowsight.html#label-mysql-connector-configure-multi-instance).

## Private links

The Snowflake connector for MySQL supports private links. For more information, see:

* [AWS PrivateLink & Snowflake](https://docs.snowflake.com/en/user-guide/admin-security-privatelink)
* [Azure Private Link & Snowflake](https://docs.snowflake.com/en/user-guide/privatelink-azure)
* [Google Cloud Private Service Connect & Snowflake](https://docs.snowflake.com/en/user-guide/private-service-connect-google)

## Agent & Connector App compatibilities

The Snowflake Connector for MySQL is being released against a specific version, described as **x.y.z version** where x is major, y is minor and z is patch. Agents on docker hub are also released with the X.Y.Z version. Each x.y.z version of Snowflake Connector for MySQL supports all agents with the same major version X=x and not greater minor version of the agent. Moreover each x.0.0 version of the Snowflake Connector for MySQL supports all (x-1).Y.Z versions of the agent for all Y and Z.

## Known limitations

The following sections describe the known limitations for the connector.

## Maximum number of tables

The connector works well with up to 200 source tables added to the replication. Adding more tables may cause the connector to become unstable.

## Transaction size

The connector is subject to the [same limitations as MySQL’s group replication](https://dev.mysql.com/doc/refman/8.4/en/group-replication-limitations.html#group-replication-limitations-transaction-size). This means that a single transaction must fit into a binary log message of no more than 4GB. Transactions exceeding this size will cause the source table to be marked as permanently failed, and require a full snapshot reload of the associated table.

## Connector availability

When installing the connector note the following limitations:

* Accounts in government regions are not supported.
* To install and configure the connector, you must be logged in as a user with the ACCOUNTADMIN role. Other roles are not supported at this time.

## Types compatibility

Due to the differences between the source database and Snowflake column types, some values cannot be converted and written into Snowflake because of the maximum column capacity or allowed ranges. For example:

* Snowflake BINARY type has a maximum length of 8 MB (8,388,608 bytes)
* Snowflake date types, like DATE, DATETIME, and TIMESTAMP, have a maximum year of 9999
* Snowflake VARCHAR type has a maximum length of 16 MB (16,777,216 bytes)

If such incompatibility happens, the replication of a table is stopped with a failure.

## Source table schema changes

The following table shows different types of changes to the source table schema and whether they are supported, and if so how.

New column names are subject to the same limitations as described in the Identifiers limitations section.

| **Type of schema change** | **Supported?** | **Notes** |
| --- | --- | --- |
| Adding a new column | Yes | The new column will be visible in the destination table just like any other column that existed at the start of the replication.  It is not possible to add a new column with the same name as a previously deleted or renamed column.  For example, if columns A and B existed initially, but A was deleted and B was renamed to B2 - it is not possible to add a column named A or B. |
| Deleting an existing column | Yes | If a column is deleted in the source table, it will not be deleted in the destination table. Instead, a soft-delete approach is followed and the column will be renamed to <previous name>\_\_SNOWFLAKE\_DELETED so that historical values can still be queried. All rows replicated after the column is deleted will have a NULL value in this column.  For example, if a column A is deleted, it will be renamed to A\_\_SNOWFLAKE\_DELETED in the destination table and the contents of the column remain unchanged. |
| Renaming a column | Yes | Renaming a column is equal to deleting the column and creating a new one with the new name. The deletion follows the soft-delete approach explained in the previous row.  For example, if column A was renamed to B - in the destination table A was renamed to A\_\_SNOWFLAKE\_DELETED, and a new column B was added. All rows existing before the change keep the column’s values in the A\_\_SNOWFLAKE\_DELETED column, while new rows added after the change have the values in the B column.  It is not possible to rename a column to the same name as a previously deleted or renamed column. For example, if columns A, B and C existed initially, but A was deleted and B was renamed to B2 - it is not possible to rename the column C to A or B. |
| Changing the type of column | Partially | Changing the type of source table column is only possible if both the previous and the new type are mapped to the same type in Snowflake.  In any other case, the replication will fail permanently. |
| Changing the precision of a numeric column | No | Changing the precision of a source table column will result in replication failing permanently. |
| Changing the scale of a numeric column | No | Changing the scale of a source table column will result in replication failing permanently. |
| Changing the primary key definition | No | Changing the primary key definition of the source table column will result in replication failing permanently. |

## High-capacity columns

An active agent is continuously reading all events from the binary log, even if some events refer to source tables that were not added for replication. If the binary log contains very large events, like updates of the BLOB-like columns, the agent might crash due to the lack of available memory.

## Primary keys

Tables without primary keys are not supported.

## Identifiers limitations

Currently, the connector does not support the " character in replicated schema, table or column names. Additionally, the following keywords are not supported:

## For schema names:

* INFORMATION\_SCHEMA

## For column names:

* \_SNOWFLAKE\_INSERTED\_AT
* \_SNOWFLAKE\_UPDATED\_AT
* \_SNOWFLAKE\_DELETED
* names with suffix \_\_SNOWFLAKE\_DELETED
* Column names marked as Cannot be used as column name in [Snowflake Reserved and Limited Keywords](https://docs.snowflake.com/en/sql-reference/reserved-keywords)

## MySQL version >= 8.0.0

Currently, the connector depends on binlog\_row\_metadata = full configuration property that was introduced in MySQL, version 8.

## Source database authorization

Private key authorization to the source database is not supported. Only authorization via user and password is supported.

# Working with the MySQL connector for Snowflake

## Introduction

Welcome to our tutorial on using the Snowflake Database Connectors. This guide will help you seamlessly transfer data from relational databases into Snowflake.

In this tutorial, you’ll gain the skills to:

* Set up MySQL and PostgreSQL in Docker, complete with sample data for ingestion.
* Install and configure two native applications, one for each database.
* Set up and fine-tune two agents, again one for each database.
* Initiate and manage data ingestion processes.
* Monitor the data ingestion workflow.

## Prerequisites

Before beginning this tutorial, ensure you meet the following requirements:

* Docker is installed and operational on your local machine.
* You have a tool available for connecting to the database. This can be a database-specific tool such as Dbeaver or a general-purpose tool such as IntelliJ or Visual Studio Code.

# Creating MySQL Database

* Docker is installed and operational on your local machine.

You have a tool available for connecting to the database. This can be a database-specific tool such as Dbeaver or a general-purpose tool such as IntelliJ or Visual Studio Code

In this section, we will go through the following steps:

* Starting the Database Instances - Learn how to launch your MySQL and PostgreSQL instances using Docker.
* Connecting to the Database - Instructions on how to establish a connection to your databases.
* Loading Sample Data - A walkthrough on how to populate your databases with sample data.

## Starting the database instances

To begin the MySQL and PostgreSQL database configuration process using Docker, create the file docker-compose.yaml. The content of the file should resemble:

services:

mysql:

container\_name: mysql8

restart: always

image: mysql:8.0.28-oracle

command: --log-bin=/var/lib/mysql/mysql-bin

--max-binlog-size=4096

--binlog-format=ROW

--binlog-row-image=FULL

--binlog-row-metadata=FULL

--sql\_mode="ONLY\_FULL\_GROUP\_BY,STRICT\_TRANS\_TABLES,NO\_ZERO\_IN\_DATE,NO\_ZERO\_DATE,ERROR\_FOR\_DIVISION\_BY\_ZERO,NO\_ENGINE\_SUBSTITUTION,PAD\_CHAR\_TO\_FULL\_LENGTH"

environment:

MYSQL\_ROOT\_PASSWORD: 'mysql'

volumes:

- ./mysql-data:/var/lib/mysql

ports:

- "3306:3306"

postgres:

image: "postgres:11"

container\_name: "postgres11"

environment:

POSTGRES\_USER: 'postgres'

POSTGRES\_PASSWORD: 'postgres'

ports:

- "5432:5432"

command:

- "postgres"

- "-c"

- "wal\_level=logical"

volumes:

- ./postgres-data:/var/lib/postgresql/data

Once your docker-compose.yaml is ready, follow these steps:

1. Open a terminal.
2. Navigate to the directory containing the docker-compose.yaml file.
3. Execute the following command to start source databases in containers:
4. docker compose up -d

After running this command, you should see two containers actively running the source databases.

## Connecting to the Database

To connect to the pre-configured databases using IntelliJ’s or Visual Studio Code database connections, perform the following steps with the provided credentials:

MySQL/PostgreSQL

1. Open your tool of choice for connecting to the MySQL.
2. Click the ‘+’ sign or similar to add data source.
3. Fill in the connection details:
   * User: root
   * Password: mysql
   * URL: jdbc:mysql://localhost:3306
4. Test the connection and save.

## Loading Sample Data

To initialize and load sample please execute those scripts in those connections.

MySQLPostgreSQL

Execute the script to generate sample data

CREATE DATABASE mysql\_ingest\_database;

USE mysql\_ingest\_database;

CREATE TABLE mysql\_rows(

id INT AUTO\_INCREMENT PRIMARY KEY,

random\_string VARCHAR(255),

random\_number INT);

INSERT INTO mysql\_rows (random\_string, random\_number) VALUES

('fukjxyiteb', 100),

('ndhbbipodi', 37),

('laebpztxzh', 83);

SELECT \* FROM mysql\_ingest\_database.mysql\_rows;

*You should see three rows in each populated database.*

# Install and configure the Native App

During this step you will:

* Install the Native Applications
* Configuring the Native Applications

## Install the Native Applications

Follow these steps to install the Application from the Snowflake Native Apps Marketplace:

1. Sign in to Snowsight.
2. In the navigation menu, select **Data Products** » **Marketplace**.
3. Install the **Snowflake Connector for MySQL** and **Snowflake Connector for PostgreSQL** applications.
4. Install both applications.

After installation, you will see the new applications listed in **Data Products** » **Apps**.

## Configuring the Native Applications

1. Sign in to Snowsight.
2. In the navigation menu, select **Data Products** » **Apps**.
3. Open each application and do the following:

MySQL/PostgreSQL

1. Select **Download Driver** and save the file. The file name will resemble mariadb-java-client-3.4.1.jar or with newer version when available. Save this file for use during agent configuration.
2. Select **Mark all as done** as we will create and populate source databases from scratch.

**Note**

No addition additional network configuration is required at this point as we’ll configure the agent later in the tutorial.

1. Click **Start configuration**.
2. On the **Configure Connector** screen, select **Configure**. The **Verify Agent Connection** page will display.
3. Select **Generate file** to generate an agent configuration file. The file name should resemble snowflake.json. Save this file for later use in the Agent Configuration section.

# Configure the agents

In this section, we’ll configure the agent that will operate with your source databases.

The first step is to create directories agent-mysql and agent-postgresql.

Within each directory, create subdirectories agent-keys and configuration. Your directory structure should resemble:

.

├── agent-mysql

│ ├── agent-keys

│ └── configuration

└── agent-postgresql

├── agent-keys

└── configuration

## Creating configuration files

In this section, we’ll add content to the configuration files for each agent to operate correctly. The configuration files include:

* snowflake.json file to connect to the Snowflake.
* datasources.json file to connect to the source databases.
* postgresql.conf/mysql.conf files with additional agent environment variables.
* JDBC Driver file for MySQL agent.

MySQL/PostgreSQL

1. In a terminal, navigate to the agent-mysql directory.
2. Create the Docker Compose file docker-compose.yaml with the following content:

services:

mysql-agent:

container\_name: mysql-agent

image: snowflakedb/database-connector-agent:latest

volumes:

- ./agent-keys:/home/agent/.ssh

- ./configuration/snowflake.json:/home/agent/snowflake.json

- ./configuration/datasources.json:/home/agent/datasources.json

- ./configuration/mariadb-java-client-3.4.1.jar:/home/agent/libs/mariadb-java-client-3.4.1.jar

env\_file:

- configuration/mysql.conf

mem\_limit: 6g

1. Move the previously downloaded snowflake.json file into the configuration directory.
2. Move the previously downloaded mariadb-java-client-3.4.1.jar file into the configuration directory.
3. In the configuration directory create datasources.json with content:

{

"MYSQLDS1": {

"url": "jdbc:mariadb://host.docker.internal:3306/?allowPublicKeyRetrieval=true&useSSL=false",

"username": "root",

"password": "mysql",

"ssl": false

}

}

1. In the configuration directory create mysql.conf with content:

JAVA\_OPTS=-Xmx5g

MYSQL\_DATASOURCE\_DRIVERPATH=/home/agent/libs/mariadb-java-client-3.4.1.jar

1. Start the agent using the following command. There shouldn’t be any error message and the agent should generate a public and private key pair for authentication to Snowflake.

docker compose stop # stops the previous container in case you've launched it before

docker compose rm -f # removes the agent container to recreate it with the latest image in case you had one before

docker compose pull # refresh remote latest tag in case you have cached previous version

docker compose up -d # run the agent

1. Please note that the **driver jar file** name should be **identical** to the one downloaded and used in the docker-compose.yaml and mysql.conf files.

When complete, your directory structure should resemble the following.

**Please note the inclusion of the automatically generated private and public keys within the agent-keys directories.**

├── agent-mysql

│ ├── agent-keys

│ │ ├── database-connector-agent-app-private-key.p8

│ │ └── database-connector-agent-app-public-key.pub

│ ├── configuration

│ │ ├── datasources.json

│ │ ├── mariadb-java-client-3.4.1.jar

│ │ ├── mysql.conf

│ │ └── snowflake.json

│ └── docker-compose.yaml

└── agent-postgresql

├── agent-keys

│ ├── database-connector-agent-app-private-key.p8

│ └── database-connector-agent-app-public-key.pub

├── configuration

│ ├── datasources.json

│ ├── postgresql.conf

│ └── snowflake.json

└── docker-compose.yaml

## Verifying connection with Snowflake

Go back to your previously created native apps. Click on the **Refresh** button in the Agent Connection section.

When successfully Configured you should see:

A screenshot of a computer

Description automatically generated

Agent is fully set up and connected. To select data to ingest Open Worksheet.

# Configure and monitor the data ingestion process

In this step, we will instruct the Connector to begin replicating the selected tables. First, let’s create a shared sink database in Snowflake.

CREATE DATABASE CONNECTORS\_DEST\_DB;

GRANT CREATE SCHEMA ON DATABASE CONNECTORS\_DEST\_DB TO APPLICATION SNOWFLAKE\_CONNECTOR\_FOR\_POSTGRESQL;

GRANT CREATE SCHEMA ON DATABASE CONNECTORS\_DEST\_DB TO APPLICATION SNOWFLAKE\_CONNECTOR\_FOR\_MYSQL;

Once the database is ready, we can move on to the configuration process.

MySQL/PostgreSQL

1. To begin table replication, you must first add a datasource from which to replicate and then specify the table to be replicated.
2. CALL SNOWFLAKE\_CONNECTOR\_FOR\_MYSQL.PUBLIC.ADD\_DATA\_SOURCE('MYSQLDS1', 'CONNECTORS\_DEST\_DB');
3. CALL SNOWFLAKE\_CONNECTOR\_FOR\_MYSQL.PUBLIC.ADD\_TABLES('MYSQLDS1', 'mysql\_ingest\_database', ARRAY\_CONSTRUCT('mysql\_rows'));
4. To monitor the replication, execute the following queries:
5. SELECT \* FROM SNOWFLAKE\_CONNECTOR\_FOR\_MYSQL.PUBLIC.REPLICATION\_STATE;
6. SELECT \* FROM SNOWFLAKE\_CONNECTOR\_FOR\_MYSQL.PUBLIC.CONNECTOR\_STATS;

## Understanding connector status

The REPLICATION\_STATE view is crucial for monitoring the status of table replication. This process encompasses three distinct phases:

1. SCHEMA\_INTROSPECTION: Ensures that the schema of the source table is accurately replicated.
2. INITIAL\_LOAD: Transfers the existing data from the source table to the destination.
3. INCREMENTAL\_LOAD: Continuously replicates ongoing changes from the source.

Upon successful replication, the status display will resemble the following:

|  |  |  |  |
| --- | --- | --- | --- |
| REPLICATION\_PHASE | SCHEMA\_INTROSPECTION\_STATUS | SNAPSHOT\_REPLICATION\_STATUS | INCREMENTAL\_REPLICATION\_STATUS |
| INCREMENTAL\_LOAD | DONE | DONE | IN PROGRESS |

# View data

## Executing the SQL Commands

Execute the following commands to view data, which should include roughly 3 rows per database.

SELECT \* FROM CONNECTORS\_DEST\_DB."psql\_rows\_schema"."postgres\_rows";

SELECT \* FROM CONNECTORS\_DEST\_DB."mysql\_ingest\_database"."mysql\_rows";

# Clean up and additional resources

To clean up your environment, execute the commands listed below. Failing to do so will leave the connector running and generating costs.

## Remove the native app

DROP APPLICATION SNOWFLAKE\_CONNECTOR\_FOR\_POSTGRESQL CASCADE;

DROP APPLICATION SNOWFLAKE\_CONNECTOR\_FOR\_MYSQL CASCADE;

## Remove warehouses, roles and users

During the installation multiple warehouses, roles and users were created. Execute the following queries to drop those objects.

MySQL/PostgreSQL

DROP ROLE MYSQL\_ADMINISTRATIVE\_AGENT\_ROLE;

DROP ROLE MYSQL\_AGENT\_ROLE;

DROP USER MYSQL\_AGENT\_USER;

DROP WAREHOUSE MYSQL\_COMPUTE\_WH;

DROP WAREHOUSE MYSQL\_OPS\_WH;

## Stop database containers

To stop the running containers with MySQL and PostgreSQL, navigate to the directory containing the docker-compose.yaml files, then execute the docker compose down -v.

## Additional resources

Continue learning about connectors using the following resources:

* [About the Snowflake Connector for MySQL](https://other-docs.snowflake.com/en/connectors/mysql6/about)
* [About the Snowflake Connector for PostgreSQL](https://other-docs.snowflake.com/en/connectors/postgres6/about)
* <https://other-docs.snowflake.com/en/connectors/tutorials/dbtutorial>

# END

## The End of document.