MariaDB Replication – A Step-by-Step Guide

MariaDB Replication is a process that allows data from a **Master** server to be copied to one or more **Slave** servers. This is useful for **backup, high availability, and load balancing**.

Why is Replication Important?

1. **Data Security** – If one server fails, data can be recovered from another server.
2. **Load Balancing** – Queries can be distributed across multiple servers for better performance.
3. **Backup** – Backups can be taken from the Slave server without affecting the Master server's performance.

How MariaDB Replication Works

1. Master Configuration:

- The **Master Server** maintains a **binary log (binlog)** that records all changes to the database (e.g., **INSERT, UPDATE, DELETE**).
- Each transaction is sequentially written to the binary log as **events**.

2. Slave Configuration:

- The **Slave Server** reads the **binary log** from the Master and applies the same changes to its database.
- The Slave maintains **two threads**:
    - **IO Thread:** Reads the binary log from the Master and writes it to a **relay log** on the Slave.

- **SQL Thread:** Reads the relay log and applies the SQL queries to replicate the data on the Slave.

3. Replication Process:

- The **Master Server** creates a binary log when a **write operation** (**INSERT, UPDATE, DELETE**) occurs.
- The **Slave's IO Thread** connects to the Master, retrieves the **binary log events**, and writes them into its **relay log**.
- The **SQL Thread** on the Slave reads the relay log and executes the queries.
- **Changes are propagated from Master to Slave in a consistent order.**

4. Replication Types:

- **Asynchronous Replication:** The Slave **does not** confirm whether it has applied the changes, so the Master does **not wait** for the Slave.
- **Semi-Synchronous Replication:** The Master **waits for at least one Slave** to acknowledge that it has received the changes before continuing other operations.
- **Synchronous Replication:** Both Master and Slave ensure that **changes are applied simultaneously**, though this method is **less common in MariaDB** and is usually achieved using **Galera Cluster**.

5. Replication Topologies:

- **Master-Slave:** One Master and multiple Slaves, commonly used for **backups and read scalability**.
- **Master-Master:** Two Masters that **replicate to each other**, providing redundancy and allowing **both servers** to handle write operations.
- **Circular Replication:** A variation of **Master-Master Replication** where **multiple servers are connected in a circular chain**, replicating changes in sequence.

6. Error Handling:

- If the **Slave loses connection** with the Master, it can **resume replication** from the last processed event in the **relay log** when the connection is restored.

7. Lag Monitoring:

- **Replication Lag** occurs when the **Slave is slower** in applying changes than the Master is in generating them.
- **Monitoring tools** can help track and mitigate replication lag to ensure **data consistency and real-time synchronization**.

Use Cases of `server_id` in MariaDB Replication

1. Master-Slave Setup:

- Each **Slave** must have a **unique `server_id`** so that the Master can identify where **replication requests** are coming from.
- A unique `server_id` ensures that each Slave can track its own **replication stream** without confusion.

2. Master-Master Replication:

- Both **Masters** must have **unique `server_id` values** to prevent changes made by one Master from being **re-applied** by the other in an infinite loop.
- A unique server_id also helps with **conflict resolution** when updates originate from different sources.

3. Failover and Recovery:

- When configuring replication for **failover scenarios**, the `server_id` ensures continuity after a failover event.
- A unique server_id prevents **replication conflicts** by ensuring that the **new Master and all Slaves** are correctly identified.

4. Replication Monitoring:

- The server_id is useful for **monitoring replication status** across multiple servers.
- It helps identify **which server is replicating from which source**, making **troubleshooting and performance analysis** easier.

Where Does the Master Server Store the Binlog?

1. Binary Log Files:

- The **binary log files** are stored on disk in the directory specified by the **`datadir`** setting in the **MariaDB configuration file (`my.cnf`)**.
- By default, this location is where all MariaDB **data files** are stored, typically **`/var/lib/mysql`** on Linux systems.
- However, this location can be changed by modifying the **`datadir`** parameter.

2. Log File Naming:

- The **binlog files** are named sequentially and typically follow this format:

```
mysql-bin.000001
mysql-bin.000002
mysql-bin.000003
```

- Each time the server writes **new transactions**, they are added to the **active binary log file**.
- When the **current log file** reaches a certain size or a **new binary log** is created, MariaDB rolls over to the **next file in the sequence**.

3. Configuring the Binlog in `my.cnf`:

- To **enable and configure** the binary log, you need to specify it in the **`my.cnf`** (or **`my.ini`** on Windows) configuration file.

Step-by-Step MariaDB Replication Setup

Step 1: Configure the Master Node

The **Master** server is the main database server from which all **Slave** servers will copy data.

1. **Edit MySQL Configuration File**

```
sudo vim /etc/my.cnf.d/mysql-server.cnf
```

Add the following lines under [mysqld]:

```
[mysqld]
bind-address = 10.128.0.14  # Master Server IP Address
server-id = 1
log_bin = mysql-bin  # Enable Binary Logging
```

1. **Save Changes & Restart MySQL**

```
sudo systemctl restart mysqld
```

2. **Create a New User for Replication**

```
mysql -u root -p
CREATE USER 'replica'@'10.128.15.211' IDENTIFIED BY 'P@ssword321';
GRANT REPLICATION SLAVE ON *.* TO 'replica'@'10.128.15.211';
```

```
FLUSH PRIVILEGES;
EXIT;
```

3. **Check the Master Server Status**

```
SHOW MASTER STATUS\G;
```

Step 2: Configure the Slave Node

1. **Edit MySQL Configuration File on Slave Server**

```
sudo vim /etc/my.cnf.d/mysql-server.cnf
```

1. **Add the following settings:**

```
[mysqld]
bind-address = 10.128.15.211  # Slave Server IP Address
server-id = 2
log_bin = mysql-bin
```

2. **Save Changes & Restart MySQL**

```
sudo systemctl restart mysqld
```

3. **Connect the Slave to the Master Server**

```
mysql -u root -p
STOP SLAVE;
CHANGE MASTER TO
MASTER_HOST='10.128.0.14',
MASTER_USER='replica',
MASTER_PASSWORD='P@ssword321',
MASTER_LOG_FILE='mysql-bin.000001',
MASTER_LOG_POS=1232;
START SLAVE;
```

Step 3: Verify the Replication

Run the following command on the **Slave Server** to check if replication is working:

```
SHOW SLAVE STATUS\G;
```

If **Slave_IO_Running** and **Slave_SQL_Running** show YES, it means the replication is successfully working.

Scenario-Based Example

Use Case: E-Commerce Website Database Scaling

Imagine you are running a **large e-commerce website** that handles thousands of transactions every second. If all users' requests go to a single **Master Server**, it may become slow and eventually crash due to overload.

Solution:

- **Master-Slave Replication** can be implemented.
- The **Master Server** will handle **write** operations (INSERT, UPDATE, DELETE).
- The **Slave Servers** will handle **read** operations (SELECT queries).

Benefits:
✓ Users will experience **faster website performance**.
✓ Data remains **consistent and backed up** on Slave servers.
✓ If the Master Server fails, one of the Slave servers can be **promoted to Master**.

Conclusion

By setting up MariaDB Replication:

- Data from the Master server is **automatically copied** to the Slave servers.

- The system **remains available** even if one server fails.
- **Performance improves** due to load distribution.

Now, try setting this up on your system and see how **MariaDB Replication** can make your database more **scalable and efficient**!