**MySQL Master/Slave install from Docker**

**install docker:**

# Update package list

sudo apt update

# Install required dependencies

sudo apt install apt-transport-https ca-certificates curl software-properties-common

# Add Docker GPG key

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

# Add Docker repository to system

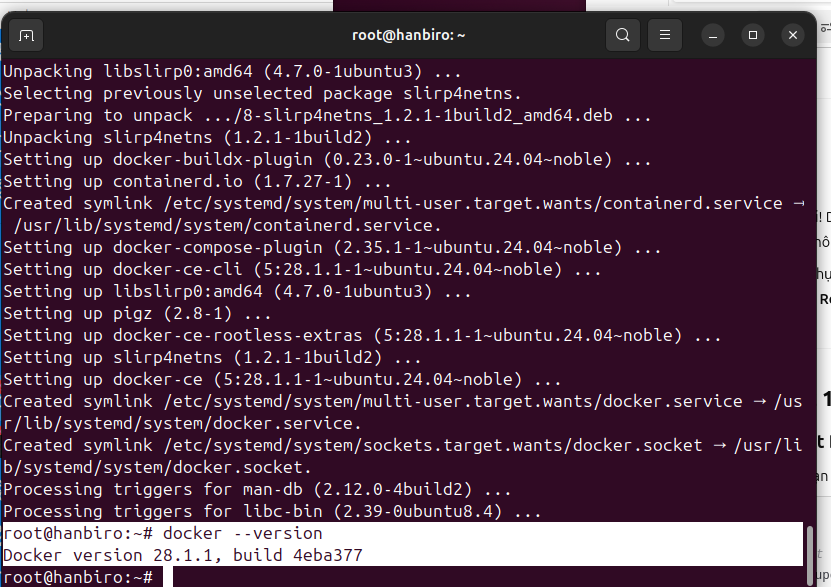
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"

# Update package list again after adding Docker repo

sudo apt update

# Install Docker

sudo apt install docker-ce



**install docker compose (easily manage and configure multiple containers at once)**

# Install Docker Compose

sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

# Grant executable permissions to docker-compose

sudo chmod +x /usr/local/bin/docker-compose

# Check the Docker Compose version installed successfully

docker-compose --version

**create configuration Docker for MySQL Master-Slave**

* create mount directories on host according to correct path

sudo mkdir -p /home/MYSQLDATA/master

sudo mkdir -p /home/MYSQLDATA/slave

sudo mkdir -p /home/MYSQLLOGS/master

sudo mkdir -p /home/MYSQLLOGS/slave

# Grant Docker writable permissions (if needed)

sudo chown -R 999:999 /home/MYSQLDATA

sudo chown -R 999:999 /home/MYSQLLOGS

\*the default user inside the container is the user with UID 999 (mysql v5.7)

* create a directory for the Docker project

mkdir mysql-replication

cd mysql-replication

* in this directory, create a file called **nano docker-compose.yml**

*version: '3.7'*

*services:*

*mysql-master:*

*image: mysql:5.7*

*container\_name: mysql-master*

*restart: always*

*environment:*

*MYSQL\_ROOT\_PASSWORD: rootpassword*

*MYSQL\_DATABASE: testdb*

*volumes:*

*- /home/MYSQLDATA/master:/var/lib/mysql*

*- /home/MYSQLLOGS/master:/var/log/mysql*

*ports:*

*- "3307:3306"*

*networks:*

*- mysql\_network*

*command: --server-id=1 --log-bin=mysql-bin --binlog-do-db=testdb*

*mysql-slave:*

*image: mysql:5.7*

*container\_name: mysql-slave*

*restart: always*

*environment:*

*MYSQL\_ROOT\_PASSWORD: rootpassword*

*MYSQL\_DATABASE: testdb*

*volumes:*

*- /home/MYSQLDATA/slave:/var/lib/mysql*

*- /home/MYSQLLOGS/slave:/var/log/mysql*

*ports:*

*- "3308:3306"*

*networks:*

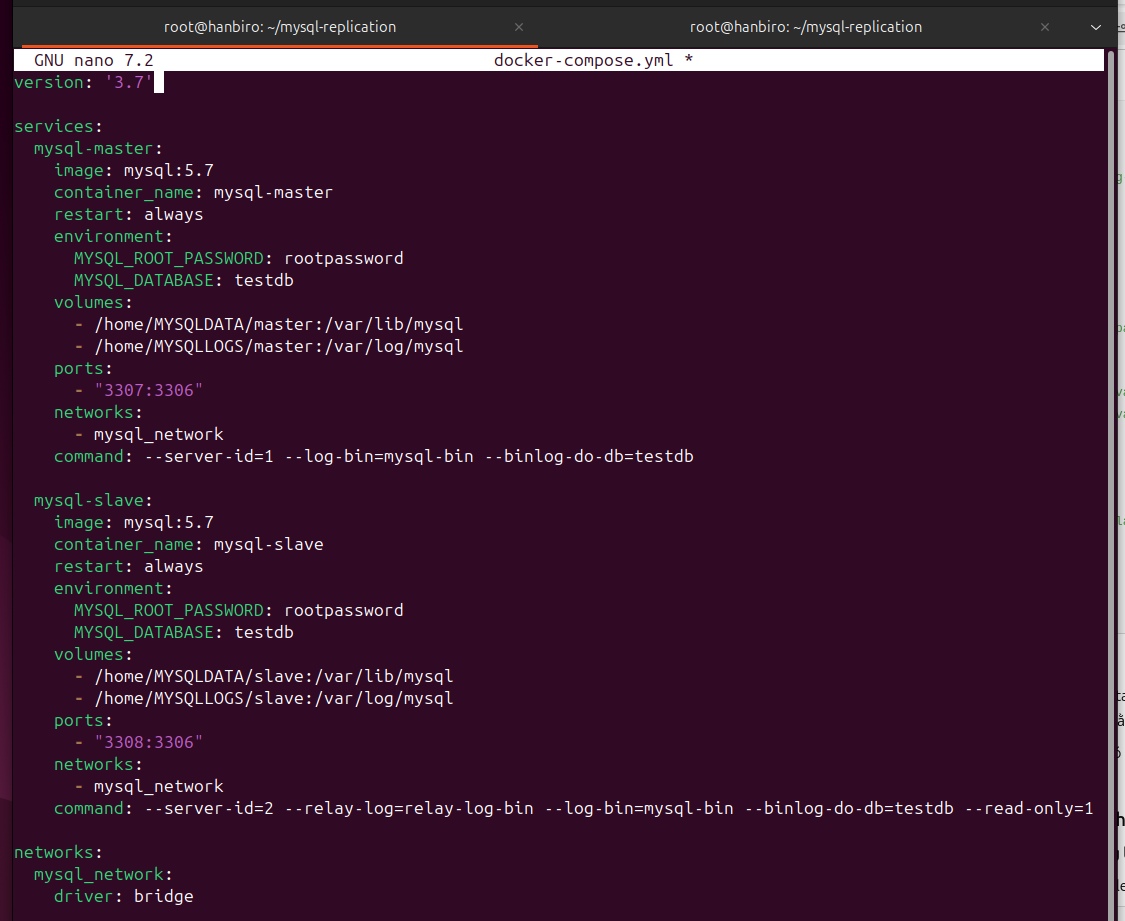
*- mysql\_network*

*command: --server-id=2 --relay-log=relay-log-bin --log-bin=mysql-bin --binlog-do-db=testdb --read-only=1*

*networks:*

*mysql\_network:*

*driver: bridge*



\* with:

* mysql-master: Configure MySQL Master with server-id=1, log-bin enabled and only testdb database synced
* mysql-slave: Configure MySQL Slave with server-id=2, read-only mode --read-only=1 and only testdb database synced
* Volumes are mapped to directories on the host to store data and logs
* create directories for data and logs

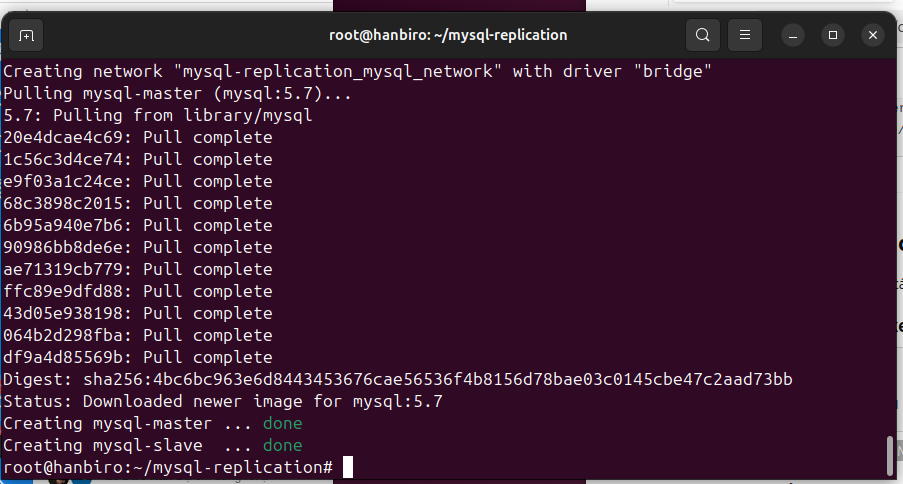
mkdir -p mysql-master/data mysql-master/logs

mkdir -p mysql-slave/data mysql-slave/logs

**start Docker Compose**

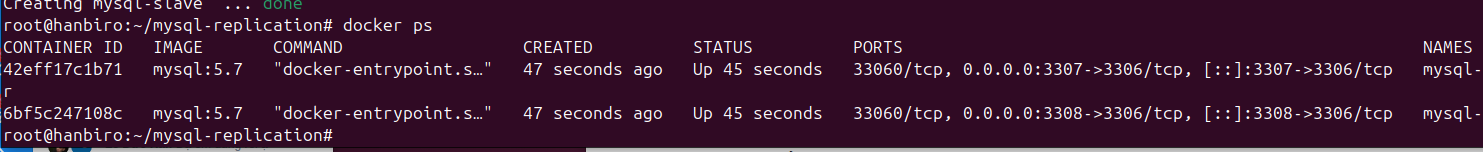
* download MySQL images, create and start 2 MySQL Master and Slave containers

docker-compose up -d



* check Docker running

docker ps



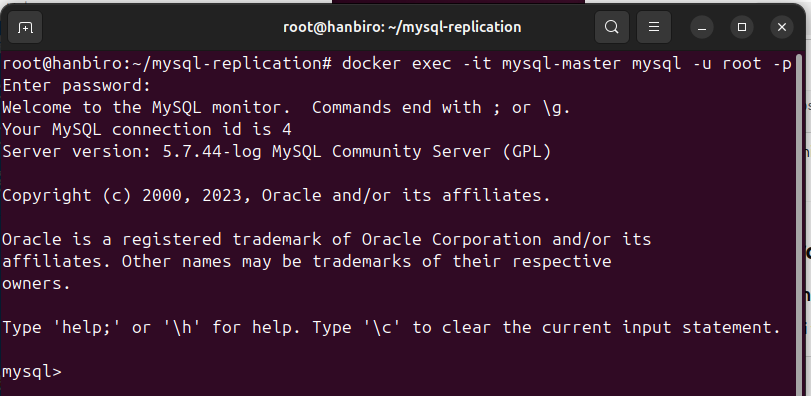
**configure Replication for Master and Slave**

**config Master**

* connect to MySQL Master

docker exec -it mysql-master mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

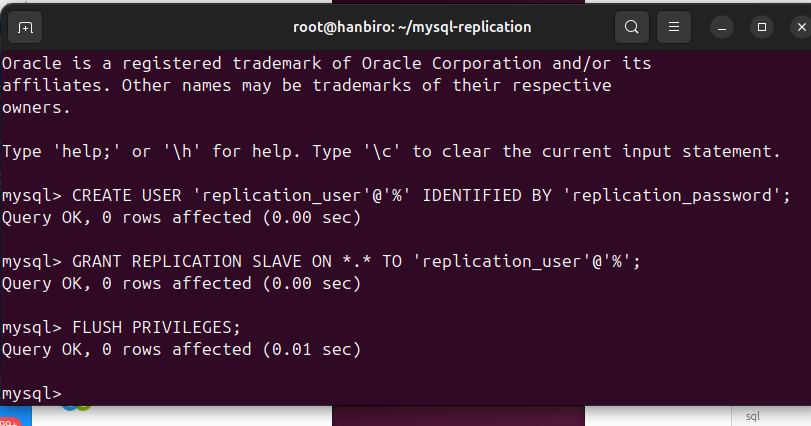
**

* create a user for replication:

CREATE USER 'replication\_user'@'%' IDENTIFIED BY 'replication\_password';

GRANT REPLICATION SLAVE ON \*.\* TO 'replication\_user'@'%';

FLUSH PRIVILEGES;



* get current log file information (for Slave sync)

SHOW MASTER STATUS;

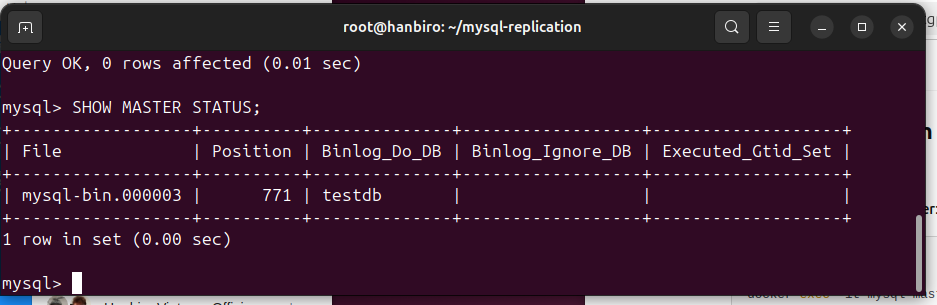
+------------------+----------+--------------+------------------+-------------------+

| File | Position | Binlog\_Do\_DB | Binlog\_Ignore\_DB | Executed\_Gtid\_Set |

+------------------+----------+--------------+------------------+-------------------+

| mysql-bin.000003 | 771 | testdb | | |

+------------------+----------+--------------+------------------+-------------------+



**config Slave**

* connect to MySQL Slave

docker exec -it mysql-slave mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

* configure replication on Slave

CHANGE MASTER TO

MASTER\_HOST='mysql-master',

MASTER\_USER='replication\_user',

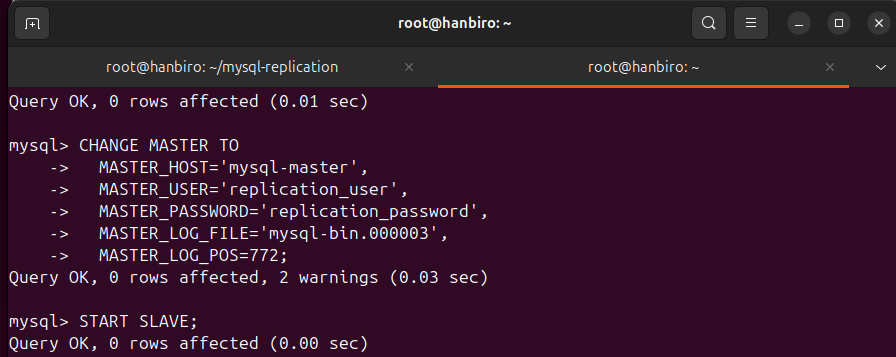
MASTER\_PASSWORD='replication\_password',

MASTER\_LOG\_FILE='mysql-bin.000003',

MASTER\_LOG\_POS=772;

START SLAVE;

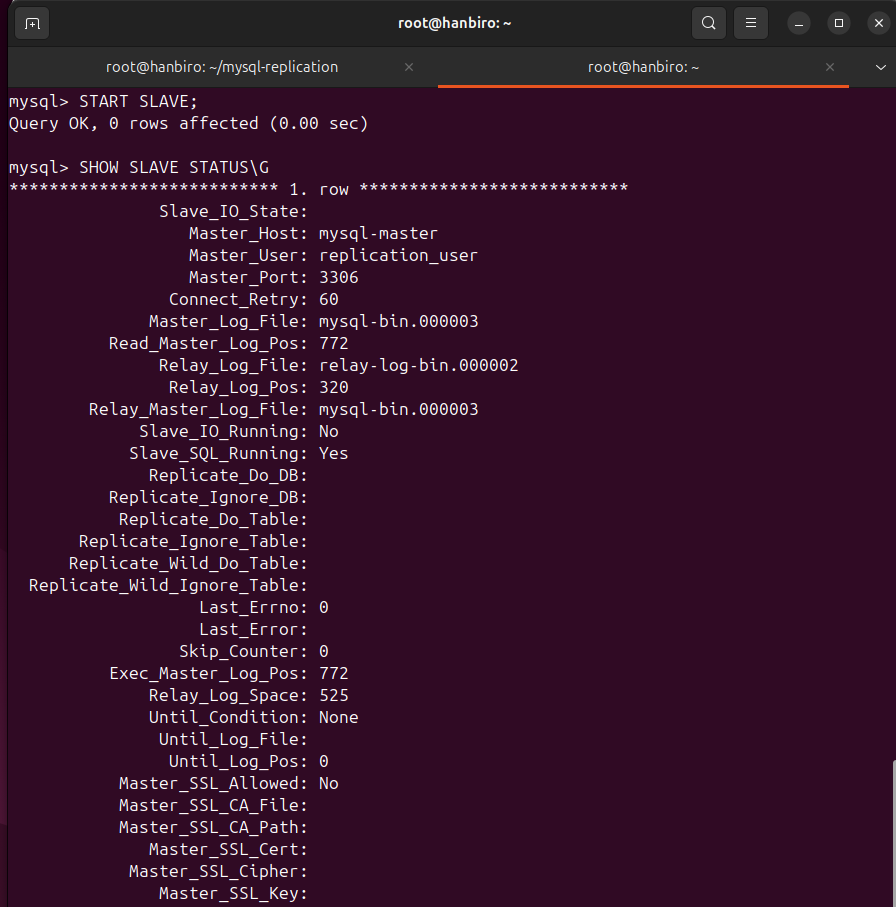
\*with: Log file and Position from Master (taken from the step above)

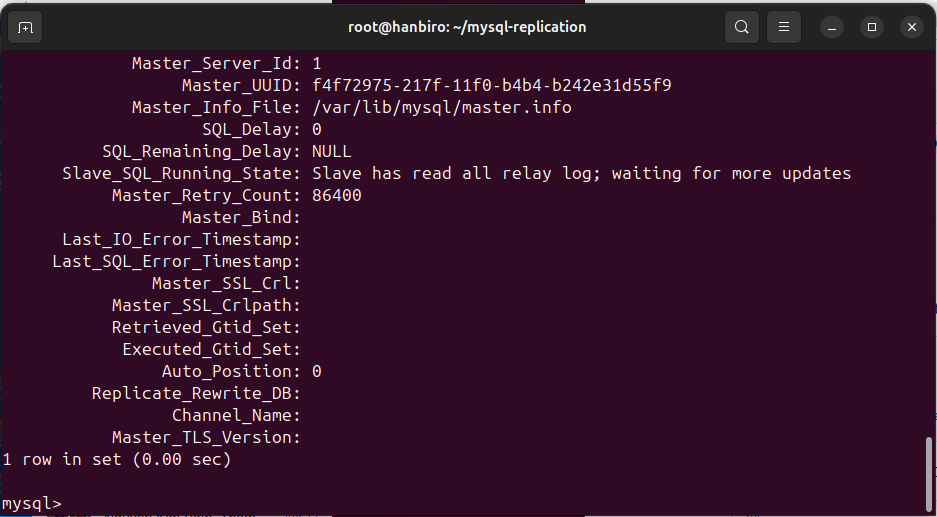


* check Replication status on Slave

SHOW SLAVE STATUS\G

\*If all Slave\_IO\_Running and Slave\_SQL\_Running fields have the value Yes, it means replication is working properly





**check Replication**

* connect to **MySQL Master**

docker exec -it mysql-master mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

* create table **test\_table** before insert

*USE testdb;*

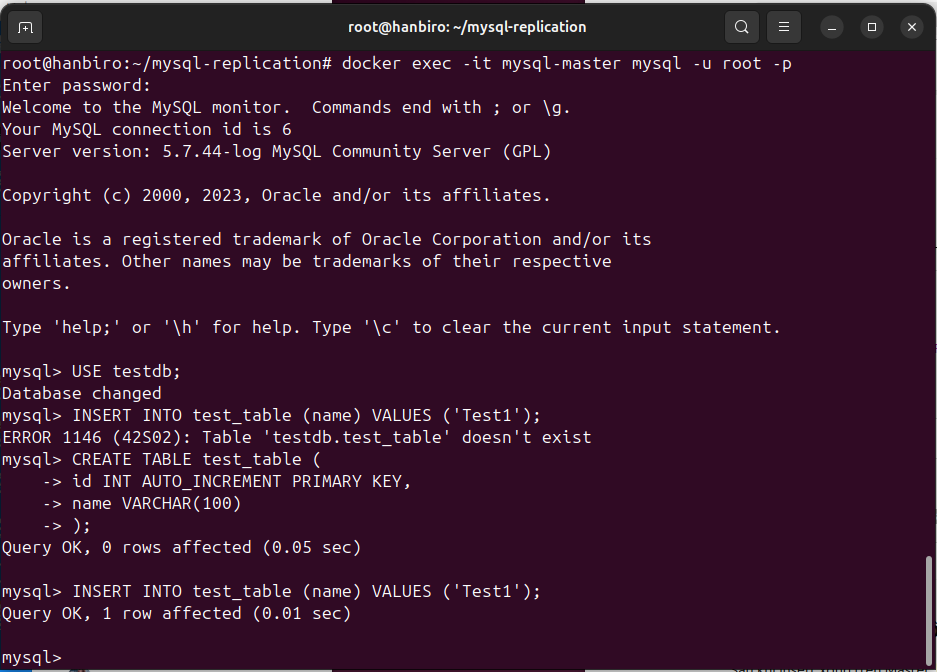
*CREATE TABLE test\_table (*

*id INT AUTO\_INCREMENT PRIMARY KEY,*

*name VARCHAR(100)*

*);*

*INSERT INTO test\_table (name) VALUES ('Test1');*

**

***!!! After inserting on Master, check on Slave to see if the data has been replicated by:***

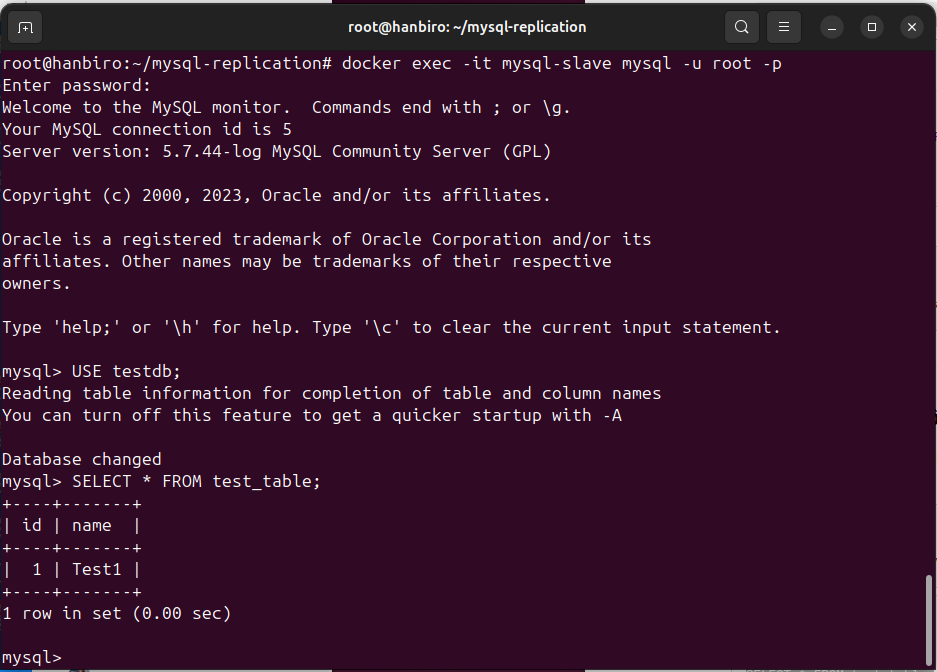
* connect to **MySQL Slave**

docker exec -it mysql-slave mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

USE testdb;

SELECT \* FROM test\_table;



**CHECK Update on Master**

* connect to **MySQL Master**

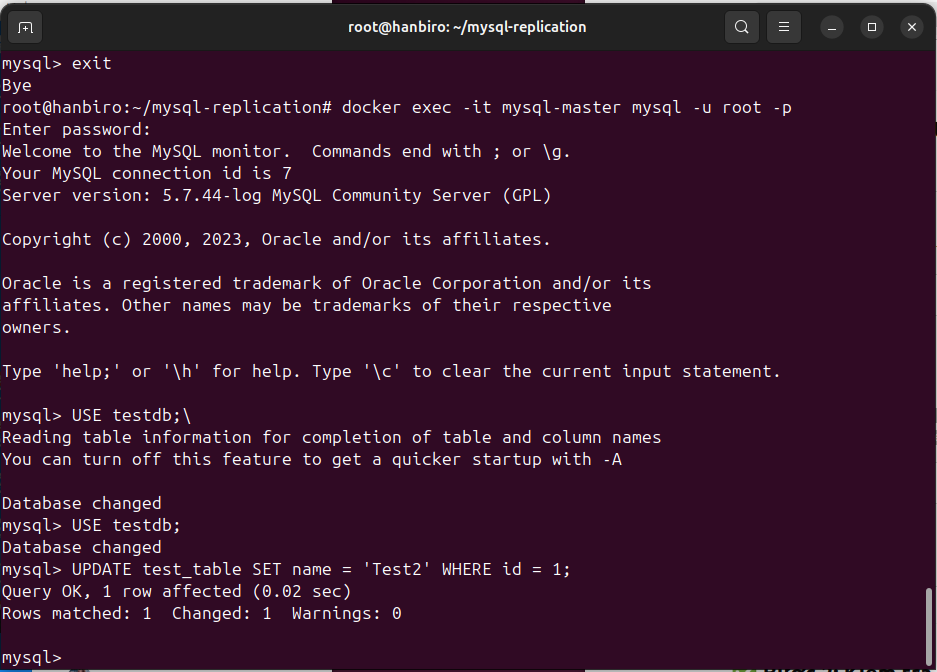
docker exec -it mysql-master mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

USE testdb;

-- Update record with id = 1

UPDATE test\_table SET name = 'Test2' WHERE id = 1;



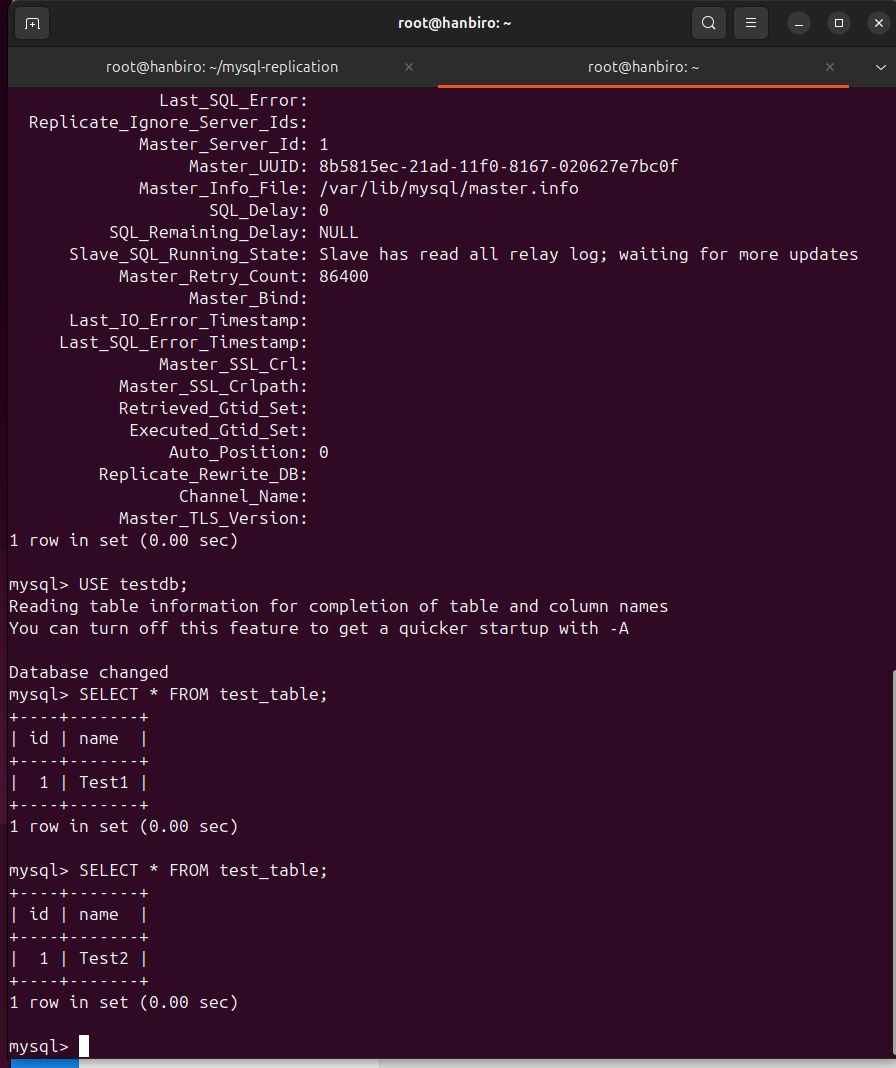
* connect to **MySQL Slave**

docker exec -it mysql-slave mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

USE testdb;

SELECT \* FROM test\_table;



**CHECK Delete on Master**

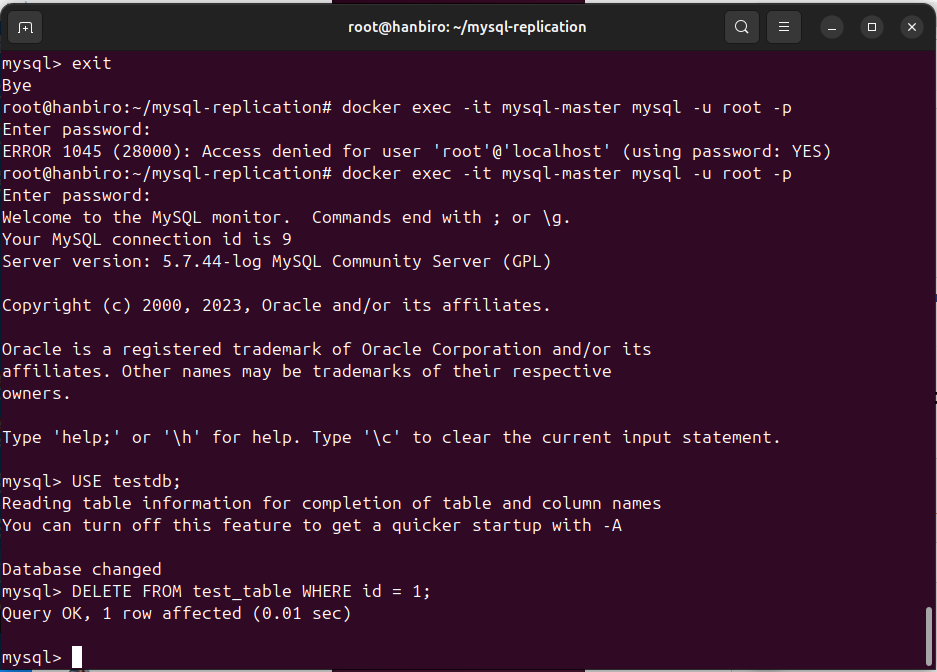
* connect to **MySQL Master**

docker exec -it mysql-master mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

USE testdb;

DELETE FROM test\_table WHERE id = 1;



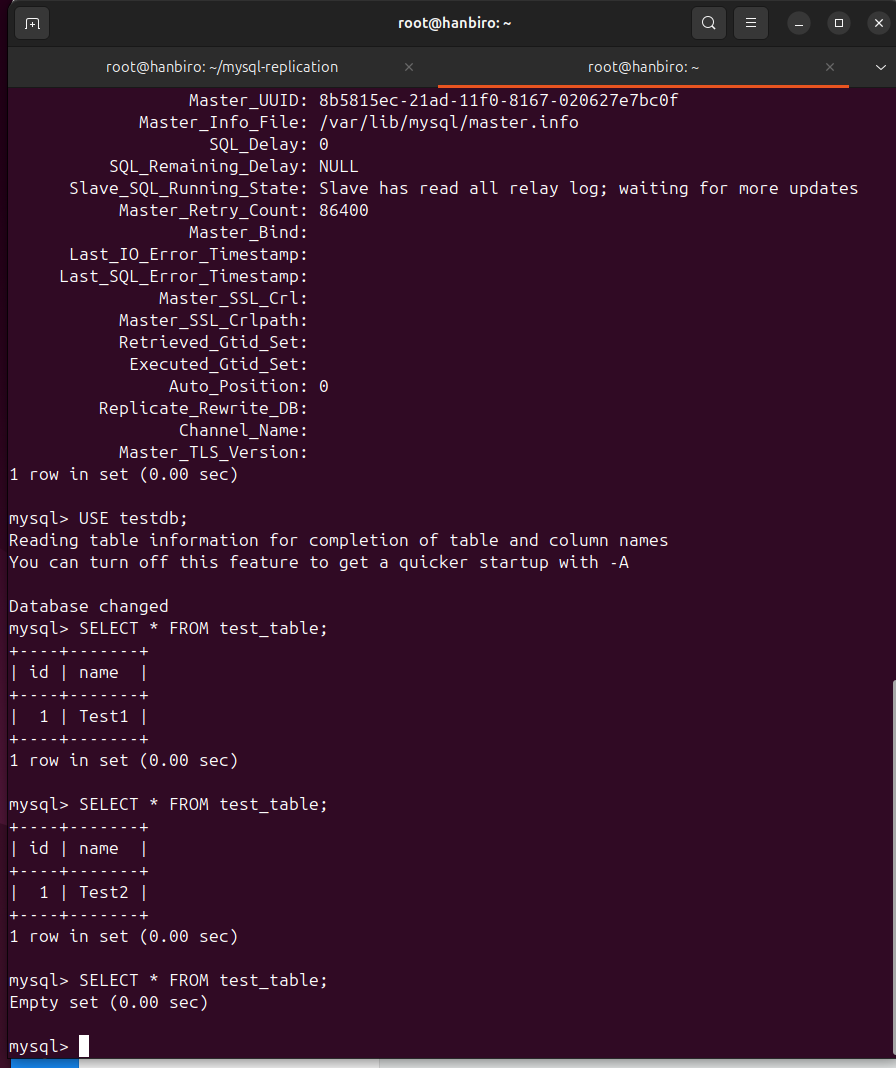
* connect to **MySQL Slave**

docker exec -it mysql-slave mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

USE testdb;

SELECT \* FROM test\_table;



**recheck**:

* connect to **MySQL Master**

docker exec -it mysql-master mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

USE testdb;

-- Create 3 more records

INSERT INTO test\_table (name) VALUES ('Alice'), ('Bob'), ('Charlie');

-- Update 1 record

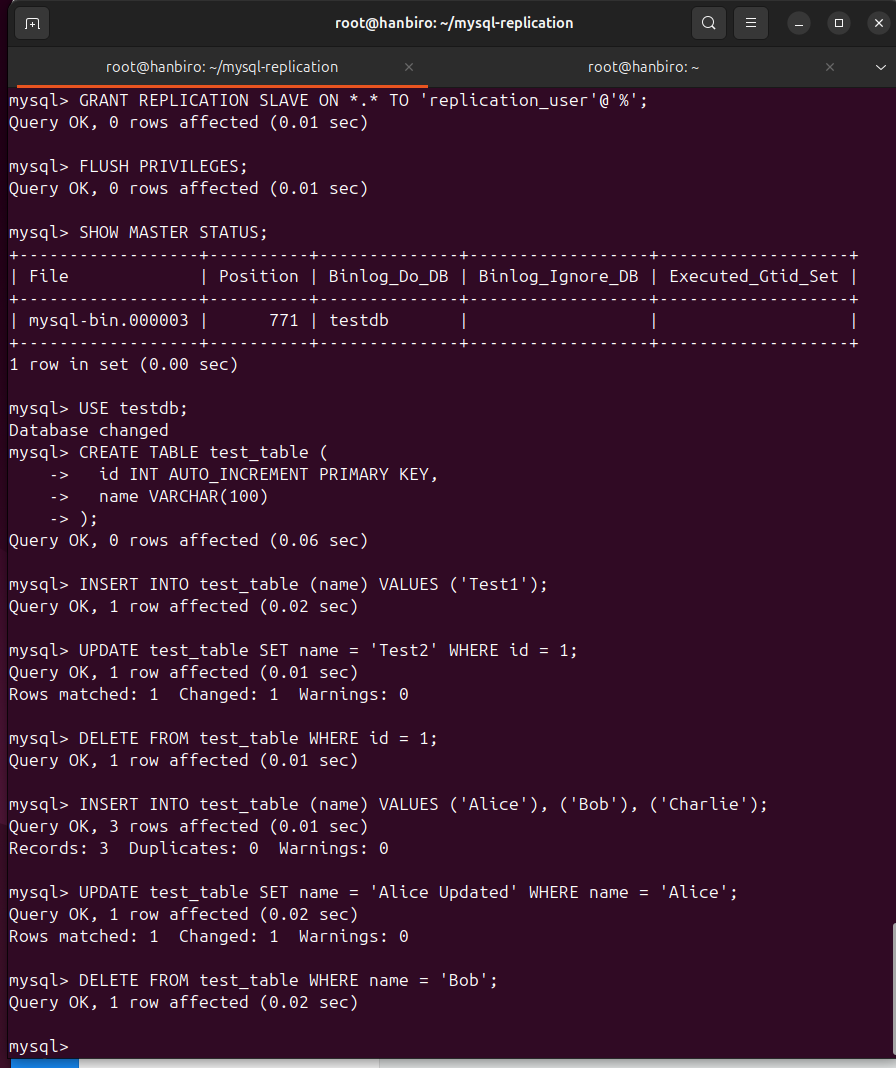
UPDATE test\_table SET name = 'Alice Updated' WHERE name = 'Alice';

-- Delete 1 record

DELETE FROM test\_table WHERE name = 'Bob';

-- Check the result on Master

SELECT \* FROM test\_table;



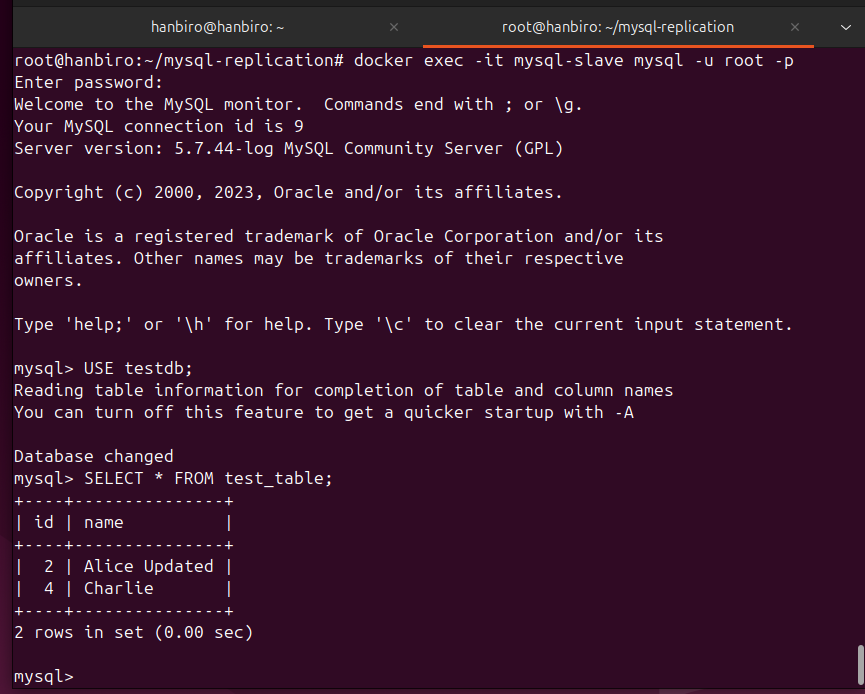
* connect to **MySQL Slave**

docker exec -it mysql-slave mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

USE testdb;

SELECT \* FROM test\_table;



\*check **CRUD** on **Slave** with **user normal**

* in **docer-composr.yml** has config for **slave**:

command: --server-id=2 --relay-log=relay-log-bin --log-bin=mysql-bin --binlog-do-db=testdb **--read-only=1**

\*with: --read-only=1 prevents normal users from performing INSERT/UPDATE/DELETE directly on the Slave, but it does not block root users or users with SUPER privileges

* connect to **MySQL Slave**

docker exec -it mysql-slave mysql -u root -p

\* *MYSQL\_ROOT\_PASSWORD: rootpassword*

* *run:*

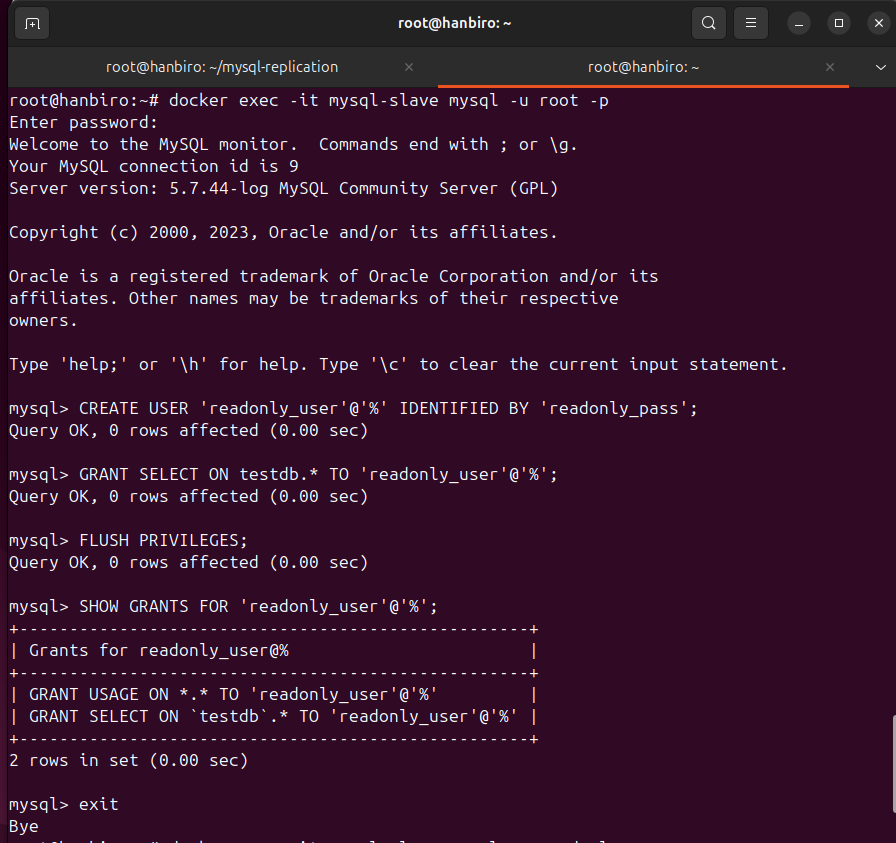
*CREATE USER 'readonly\_user'@'%' IDENTIFIED BY 'readonly\_pass';*

*GRANT USAGE ON \*.\* TO 'readonly\_user'@'%'*

*GRANT SELECT ON `testdb`.\* TO 'readonly\_user'@'%'*

\*with:

* readonly user has SELECT privilege on all tables in the testdb database
* **No** INSERT, UPDATE, DELETE, DROP, or ALTER privileges



* login **readonly\_user** into MySQL to check:

docker exec -it mysql-slave mysql -u readonly\_user -p

\*check **SELECT**:

SELECT \* FROM test\_table;

\*check **INSERT**:

INSERT INTO test\_table (name) VALUES ('Hack Attempt');

\*check **UPDATE**:

UPDATE test\_table SET name = 'Updated Name' WHERE id = 2;

\*check **DELETE**:

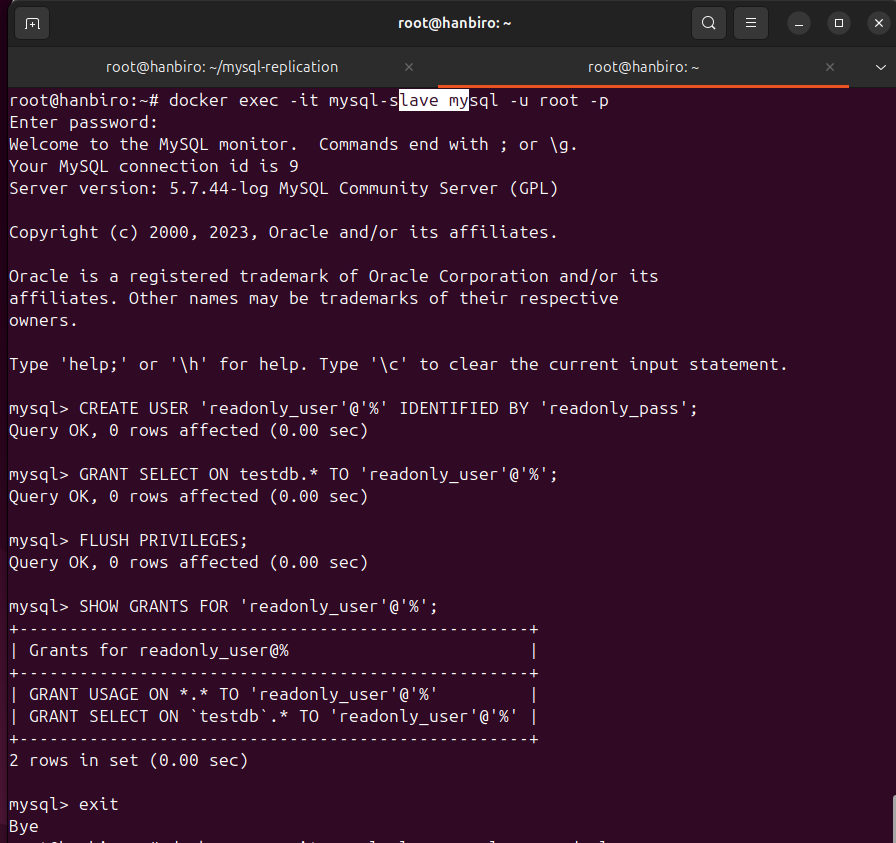
DELETE FROM test\_table WHERE id = 2;

\*check **DROP**:

DROP TABLE test\_table;

\*check **ALTER**:

ALTER TABLE test\_table ADD COLUMN age INT;



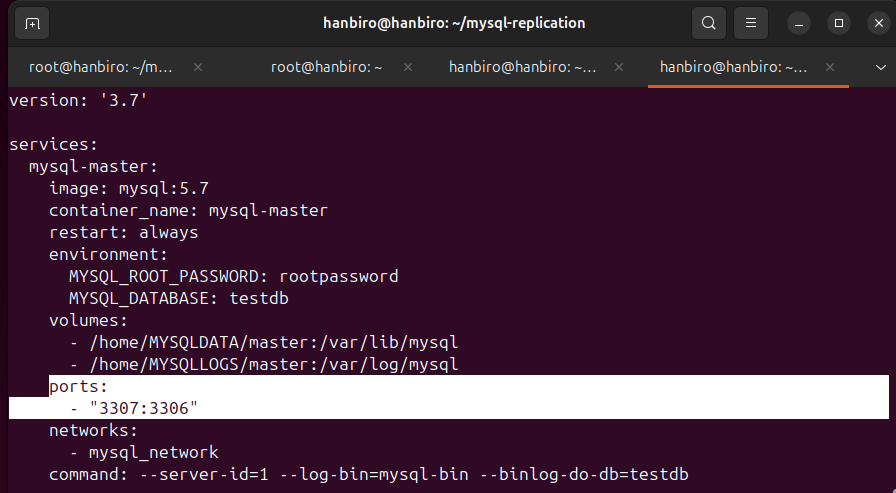
## **Does the data entered on the Master server applied to all to the Slave server? [ additional ]**

Yes, the data entered on the Master server will be automatically replicated to the Slave server. In a MySQL Master-Slave replication setup, the Master server records all changes (INSERT, UPDATE, DELETE) in its binary log. The Slave server then reads these logs and applies the changes to keep its data synchronized with the Master

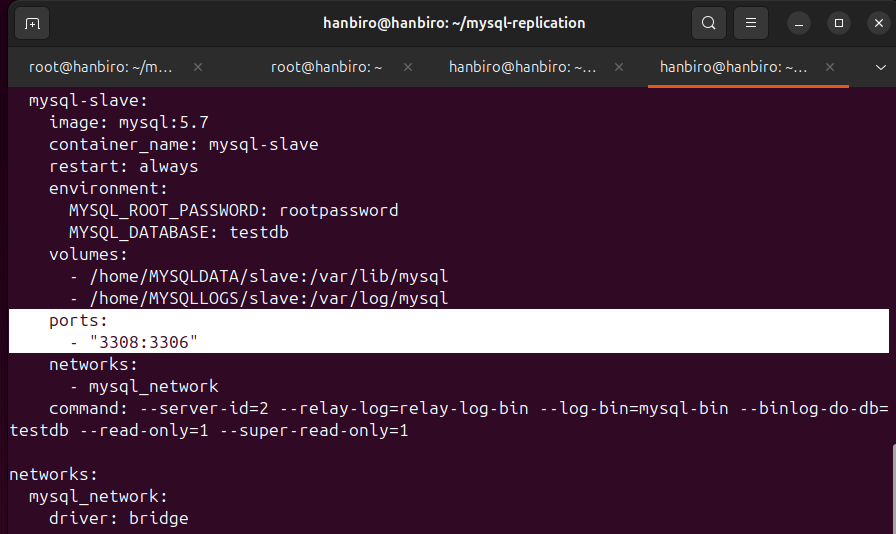
This means that any data added, modified, or deleted on the Master server will be automatically replicated to the Slave server, ensuring consistency between them without the need for manual intervention. However, replication is typically one-way: changes on the Master are propagated to the Slave, but not vice versa unless specifically configured for multi-master replication

**Required Firewall Settings for MySQL Replication**

* in the MySQL Replication configuration between Master and Slave (which can be applied to Dual Master environment later), ports need to be opened on the firewall to allow the servers to communicate with each other over network connections
* especially in this case, when using Docker, ports need to be opened between MySQL containers and the server to ensure MySQL Replication works properly
* the default MySQL port is 3306, but in this configuration, you have mapped the ports from the containers to other ports on the host to avoid conflicts with other services
* Port 3307: This port is mapped from port 3306 in the MySQL Master container. The firewall needs to open port 3307 on the host server to allow connections from the Slave server and other clients to the Master server

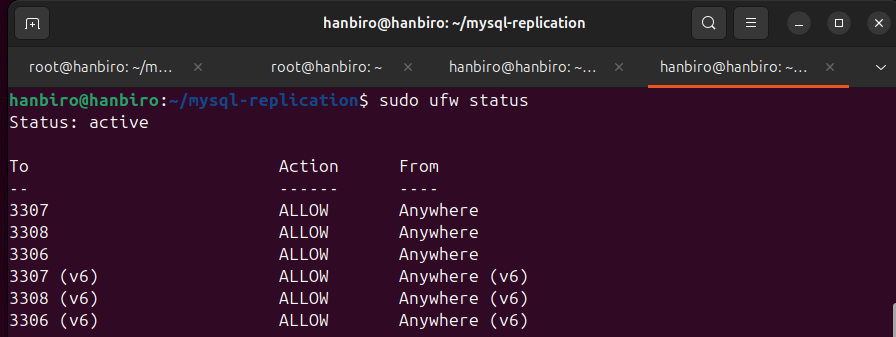


* Port 3308: This port is mapped from port 3306 in the MySQL Slave container. Port 3308 needs to be opened on the host server so that the Slave server can receive data from the Master server



sudo ufw allow 3307

sudo ufw allow 3308

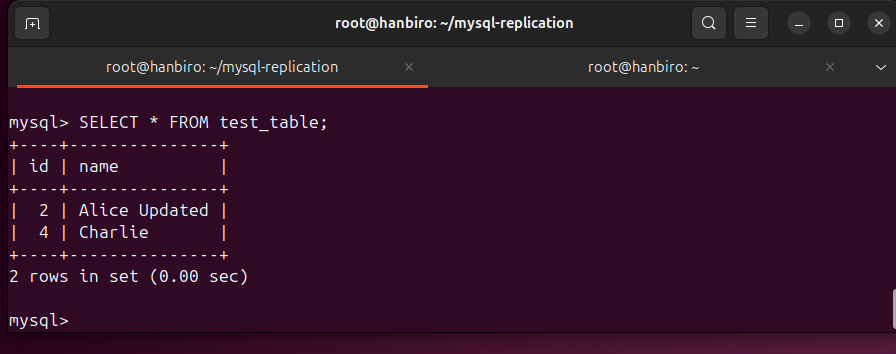


**What are the backup and restore procedures during operating?**

* Ensures data can be backed up and restored without affecting ongoing replication.
* Protects data in the event of data loss, application failure, or system failure.

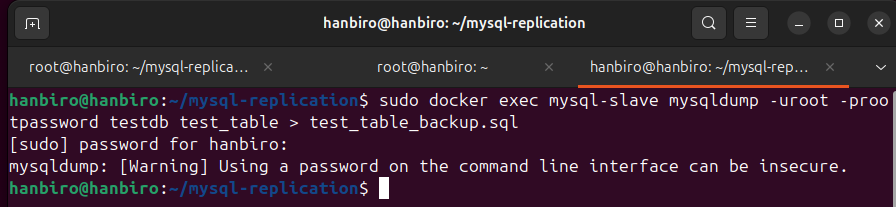
[ TEST ]

**sample data on Master**



* **create .sql backup file from Slave**: on the host machine

docker exec mysql-slave mysqldump -uroot -prootpassword testdb test\_table > test\_table\_backup.sql



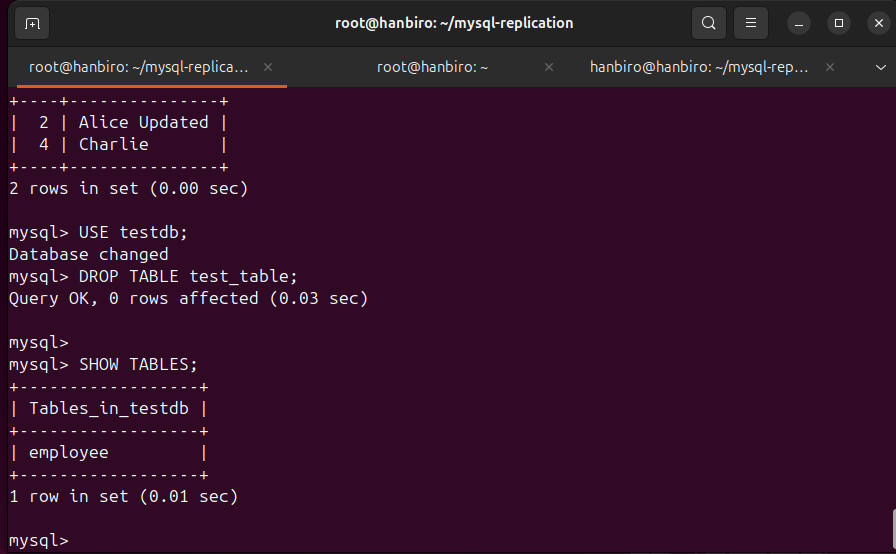
* simulate data loss on **Master (DROP table)**

**docker exec -it mysql-master mysql -uroot -p**

**USE testdb;**

**DROP TABLE test\_table;**

**SHOW TABLES;**

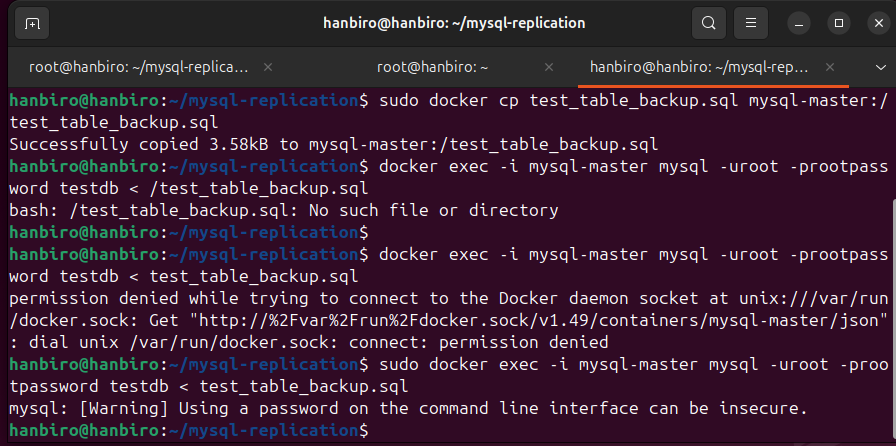
****

* restore from **.sql** backup file, on the local machine

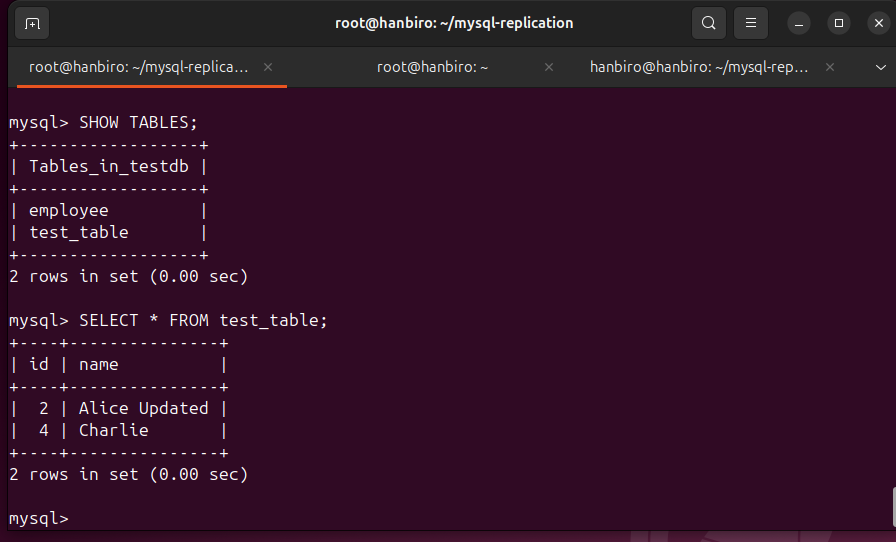
docker cp test\_table\_backup.sql mysql-master:/test\_table\_backup.sql

* restore:

docker exec -i mysql-master mysql -uroot -prootpassword testdb < test\_table\_backup.sql



* check in **Master:**



## 

## 

## 

## **What are the Pros of this Configuration?**

1. Clear division of roles between writing and reading

Master only performs data writing operations such as: INSERT, UPDATE, DELETE.

Slave only performs read queries (SELECT), helping to reduce I/O resources for Master.

The solution is suitable for systems with large reading volumes, such as: web backend, reporting systems, ERP, CMS, etc.

1. Increase performance and scalability

Multiple Slaves can be deployed to share the read query load.

Supports using a Load Balancer to distribute SELECT queries among Slaves.

Helps the system operate stably when there are hundreds/even thousands of concurrent connections.

1. Increase availability and redundancy

If the Master has a problem, the Slave can be quickly switched to the Master (failover).

Data always has a near-real-time copy on the Slave.

Creates a foundation for expansion to a Dual Master model or using HA tools such as MHA, Orchestrator, etc.

1. Easy backup and recovery

Can perform data backup from Slave without affecting Master performance.

Suitable for periodic backup tasks, data recovery testing (restore).

1. Separate data and log storage

According to the requirements of the assignment:

Data is stored at: /home/MYSQLDATA

Logs are stored at: /home/MYSQLLOGS

Makes system management easier, convenient for authorization, separate log backup, log monitoring, or applying logrotate.

1. Flexible in deployment (Docker or real machine)

Current configuration uses Docker, suitable for development environment, quick testing, CI/CD.

Can easily switch to the real environment (physical or virtual) with minimal effort.

Docker volumes are fixed according to the requirements of the problem → easy to backup & migrate.

## **What are the Cons of this Configuration?**

1. Single point of failure for Master

If the Master server goes down and there is no failover mechanism in place, write operations cannot be performed, as the Master is the only node that handles writes (INSERT, UPDATE, DELETE).

1. Replication delay

The Slave server may experience replication lag, meaning there could be a delay between when data is written on the Master and when it appears on the Slave. This may cause slight inconsistencies if read queries are served from the Slave.

1. Slave is read-only

The Slave is configured to be read-only, meaning no data modifications (INSERT, UPDATE, DELETE) can be done on it. While this helps maintain consistency, it can be restrictive for applications that need to write data to the Slave for backup or analytics purposes.

1. Scaling limitations

While adding more Slaves can improve read performance, the Master server remains a bottleneck for write-heavy applications. The configuration does not address scaling for write operations, which can become problematic with a high volume of writes.

1. Replication complexity with Dual Master setup

The setup may need additional configuration when considering a Dual Master setup (where both nodes handle both reads and writes). Handling conflict resolution, data consistency, and potential circular replication issues can add complexity.

1. Backup impact on Slave

Performing large data backups on the Slave server could potentially affect replication performance, especially if the backup process consumes significant system resources (CPU, memory, disk I/O).

1. Requires manual intervention for failover

If a failover occurs and the Slave needs to become the new Master, manual intervention may be required unless automatic failover tools like MHA, Orchestrator, or MySQL Group Replication are implemented.

1. Possible configuration and synchronization issues

Misconfigurations between Master and Slave (e.g., incorrect binlog positions, mismatched database schemas) can cause replication errors, requiring additional monitoring and troubleshooting efforts.

## **What are the Precautions for Usage?**

1. Monitor replication status – Check Slave’s replication regularly to detect issues early.
2. Handle replication lag – Watch for replication delays, especially under heavy writes.
3. Ensure failover mechanisms – Set up automatic failover or manual failover procedures.
4. Do not write to Slave – Avoid performing writes on Slave to prevent conflicts.
5. Regular backups – Backup data from the Slave regularly to minimize impact on the Master.
6. Consider data consistency – Be aware of replication lag that may cause data inconsistency.
7. Monitor disk space – Keep an eye on disk usage for logs and data files; implement log rotation.
8. Limit Slave connections – Avoid overloading the Slave with too many queries.
9. Follow security best practices – Use secure authentication and encrypted communication for replication.
10. Careful with MySQL upgrades – Test compatibility before upgrading MySQL versions.
11. Test disaster recovery – Regularly test recovery and failover processes.

## **Consider Which Service Is Suitable with This Configuration**

This Master-Slave MySQL configuration is well-suited for:

1. **Web Applications with High Read Load** e.g. news sites, blogs, e-commerce platforms – where users mostly view content.
2. **Reporting and Analytics Systems** e.g. BI dashboards, business reports – where data is frequently read but not written.
3. **ERP or CMS Systems** Systems with separated read/write operations benefit from load balancing on read queries.
4. **Backup and Archiving Systems** Use Slave servers for performing backups without interrupting the main database (Master).
5. **APIs with Read-Heavy Requests** REST or GraphQL APIs that handle thousands of read queries per second.

This configuration helps optimize performance, reduce Master load, and improve scalability and availability in these types of services.

## **Although There Will Be Any Problem During the Test, How Can You Solve This Problem When the Log Files Will Keep Increasing with the Actual Service?**

Set log auto delete time  
 Use expire\_logs\_days = 7 to automatically delete logs older than 7 days.

Manually clean old logs  
 Use this MySQL command:  
 PURGE BINARY LOGS BEFORE NOW() - INTERVAL 7 DAY;

Monitor log size and disk space  
 Use tools or scripts to watch log size and free space.

Use logrotate for Docker logs  
 Prevent Docker logs from using too much disk.

Separate log folder  
 Store logs in /home/MYSQLLOGS to make cleanup and backup easier.

Use external storage (optional)  
 Store logs on another disk or server to reduce local disk usage.