

Scaling Databases: Master Slave

Script

- Present a hypothetical case study on why a single node deployment is a bottleneck and why we need master-slave replication
 - Increased read throughput, load balance read queries
 - High availability and instant failover with no data loss
 - Backups and data distribution
- Discuss how replication works on MySQL
 - binlog, relay log, IO threads
- Setup a simple master-slave without any data
 - using Docker
 - using AWS
 - perform some DDL and DML operations and show that the replica picks up the changes
 - Show how to add existing data
- Sync vs Async Replication
 - Show AWS Multi-AZ deployment
- Statement-based vs Row-based replication
 - Differences
 - Examples

Extra things (not added right now) we can cover for advanced folks

- GTID
- Replication lag
 - Why replication lag happens,
 - Reading your Writes
- Load balancing read queries using master + 2 slave nodes
 - HAProxy
 - ProxySQL
- Custom implementation of binlog replication: MySQL master, Postgres slave -> Data pipeline/warehousing example
 - using Debezium + Kafka + Change Data Capture

Disclaimer: This tutorial assumes some familiarity with Docker and any application framework (like Django/ Spring/ Ruby on Rails/ Laravel) using a SQL database (MySQL). Check the Appendix for guides on how to set up Docker on your machine.

Introduction: Story

We are TunTun Ladoo, a leading online retailer and one-stop shop for ladoos. Like every year, this Diwali season we're planning a massive Dhamaka sale, where hundreds of thousands of visitors will shop ladoos on our website and app in a matter of hours.

Last year due to our growing popularity we went down during peak hours. Our production MySQL DB wasn't able to handle the load and took a downtime of about an hour. It was bad, we lost a bunch of revenue and in the worst case, the database could have been corrupted, taking our site down for days.

In the postmortem, we discovered that we have a bunch of expensive SQL queries which increases the overall CPU and Memory pressure of our MySQL instance. The majority of these queries were read-only, joining data across multiple large tables. For example, queries to check the stock, pricing, and inventory of Ladoos, analytical queries for checking sales and orders in real-time.

This year we're gearing up not to repeat the same, we have a plan to scale our MySQL. Our goals are pretty straightforward:

- Our read load during peak sales is higher than what a single machine can handle. We need to fix this bottleneck. A simple approach would be to buy a more powerful machine i.e. vertical scaling. But it's costly and after a certain point, we won't be able to scale anymore. Instead, we can potentially distribute the load across machines, i.e scaling horizontally.
- With multiple MySQL nodes, we get better redundancy. Even if one of these nodes goes down or crashes, another one can take over immediately.
- Our OLTP MySQL database is not meant for processing large analytical queries containing multiple joins and aggregations. An OLAP database is better suited for this purpose. We need to capture the changes in our production database and sync them to our data warehouse in real-time.

Let's assume that our dataset is static, i.e. it doesn't change over time. Scaling our reads is super easy in that case. We just have to copy the data to each node once and we are good

to go. But in the real world, as our tables are constantly changing, we need to handle the *changes* in replicated data and propagate them to each replica.

Replication

Luckily for us, MySQL's built-in replication solves the problem of keeping one server's data synchronized with another's. In this case, we need to mark one of the servers as a leader and MySQL will keep other nodes data in sync with the master copy.

More On Master Slave

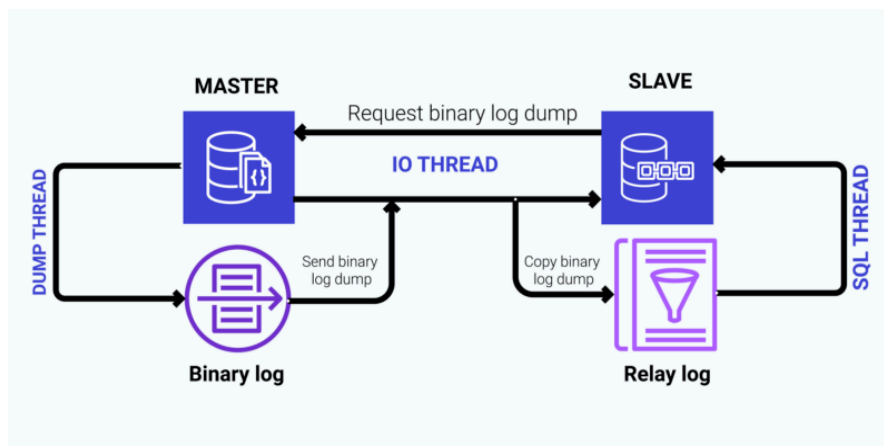
In this setup, we would have multiple MySQL nodes, one of which would be termed the leader/master/ primary. *All writes from clients need to be processed and persisted by the leader.* Other nodes are called followers/ replica/ slaves and are read-only from the client's viewpoint.

After persisting the data locally, the leader node sends the data change to followers using the *replication log/change stream*. Each follower takes the log and updates its local copy accordingly, making sure writes are applied in the same order as processed on the leader. When reading data, clients can query either leader or any of the followers, however, writes are only processed by the leader.

Setting up Replication on MySQL

At a higher level, MySQL replication consists of three different parts. Before committing every transaction that updates data, the master node records the changes in the binary log file in a serialized way (even if the transactions were running concurrently). During boot up, replica nodes start a worker thread called *IO slave thread* which maintains a connection to the master node and starts the *binlog dump* process. This binlog dump process reads data from the master's binary log and dumps it in its local *relay log*. It waits for signals from the master node for any future updates. The SQL slave thread reads and

replays events from the relay log, updating the replica's data to be in sync with the master. The process of fetching and replaying events is decoupled and runs asynchronously. But as



the replica uses two different threads for fetching and processing events, replication needs to be serialized. Transactions that might have run in parallel on the master are processed serially on the replica.

Setting up replication for a blank server is fairly simple. In the master nodes configuration we need to enable binlog by setting `log_bin = 1` and specify the logging format `binlog_format` as `ROW`. For servers used in replication, we need to specify a unique server ID for every node.

```
# master.conf
[mysqld]
server-id = 1
log_bin = 1
binlog_format = ROW
binlog_do_db = scalerdb
default_authentication_plugin = mysql_native_password
```

```
# slave.conf
[mysqld]
server-id = 2
log_bin = 1
binlog_do_db = scalerdb
default_authentication_plugin = mysql_native_password
```

```
→ docker run \
  --name masternode --rm \
  -e MYSQL_DATABASE=scalerdb \
  -e MYSQL_USER=master_user \
  -e MYSQL_PASSWORD=master_pwd \
  -e MYSQL_ROOT_PASSWORD=111 \
  -v $PWD/master.conf:/etc/mysql/conf.d/mysql.conf.cnf \
  mysql:8
```

```
→ docker run \
  --name slavenode --rm \
  -e MYSQL_DATABASE=scalerdb \
  -e MYSQL_USER=slave_user \
  -e MYSQL_PASSWORD=slave_pwd \
  -e MYSQL_ROOT_PASSWORD=111 \
  -v $PWD/slave.conf:/etc/mysql/conf.d/mysql.conf.cnf \
  mysql:8
```

The slavenode IO thread, which makes a TCP connection to masternode needs permissions to connect as a user and read the master's binary log file.

```
# Create a replication user, SSH into master node and get a local  
# MySQL CLI
```

```
→ docker exec -it masternode /bin/bash -c "mysql -u root --password=111"
```

```
mysql> CREATE USER "replication_user"@ "%" IDENTIFIED BY  
"replication_password";
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> GRANT REPLICATION SLAVE ON *.* TO "replication_user"@ "%";
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> FLUSH PRIVILEGES;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SHOW MASTER STATUS;
```

File	Position	Binlog_Do_DB	Binlog_Ignore_DB	Executed_Gtid_Set
1.000003	851	scalerdb		

```
1 row in set (0.00 sec)
```

```
# Get the masternode IP address and open slavenode CLI
```

```
→ docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
```

```
masternode
```

```
172.17.0.2
```

```
→ docker exec -it slavenode /bin/bash -c "mysql -u root --password=111"
```

```
mysql> SHOW SLAVE STATUS;
```

```
# Empty set, 1 warning (0.00 sec)
```

```
mysql> CHANGE MASTER TO
```

```
-> MASTER_HOST='172.17.0.2',  
-> MASTER_USER='replication_user',  
-> MASTER_PASSWORD='replication_password',  
-> MASTER_LOG_FILE='1.000003',  
-> MASTER_LOG_POS=851;
```

```
# Query OK, 0 rows affected, 8 warnings (0.03 sec)
```

```
mysql> START SLAVE;
```


```
mysql> CREATE TABLE customers (
  ->     id int NOT NULL AUTO_INCREMENT,
  ->     firstname varchar(255),
  ->     lastname varchar(255),
  ->     address varchar(255),
  ->     city varchar(255),
  ->     unique_id varchar(255),
  ->     CONSTRAINT pk_customers PRIMARY KEY (id)
  -> );
# Query OK, 0 rows affected (0.02 sec)

mysql>
mysql> INSERT INTO customers (firstname, lastname, address, city, unique_id)
  -> VALUES ('Joydeep', 'Mukherjee', 'Sakti Statesman, Bellandur',
'Bangalore', UUID());
# Query OK, 1 row affected (0.01 sec)

mysql> use scalerdb;
# Database changed
mysql>
mysql> select * from customers;
# Empty set (0.00 sec)

mysql> select * from customers;
+----+-----+-----+-----+-----+-----+
| id | firstname | lastname | address | city | unique_id |
+----+-----+-----+-----+-----+-----+
| 1 | Joydeep | Mukherjee | Sakti Statesman, Bellandur | Bangalore | 4f982cf4-2637-11ed-aec3 |
+----+-----+-----+-----+-----+-----+
# 1 row in set (0.00 sec)
```

AWS Example

 How do I create a read replica for an Amazon RDS database?

Adding a Node Later on

1. Take a consistent snapshot of the leaders' database at some point, without locking the whole database, and associate the snapshot with an exact position in binlog.
2. Copy the snapshot to a new replica and connect to the leader. It will request all the changes that have happened since the snapshot was taken (*binlog coordinates*)
3. Eventually, the follower will process the backlog of changes.

Replication Types

TODO: Replace graphics

When updating the price of Ladoos, the master node will receive an update request from our application servers at some point. Once the master updates its local copy, different replication strategies can be used to broadcast the changes to replicas. Replication between the master and slaves can happen synchronously or asynchronously.

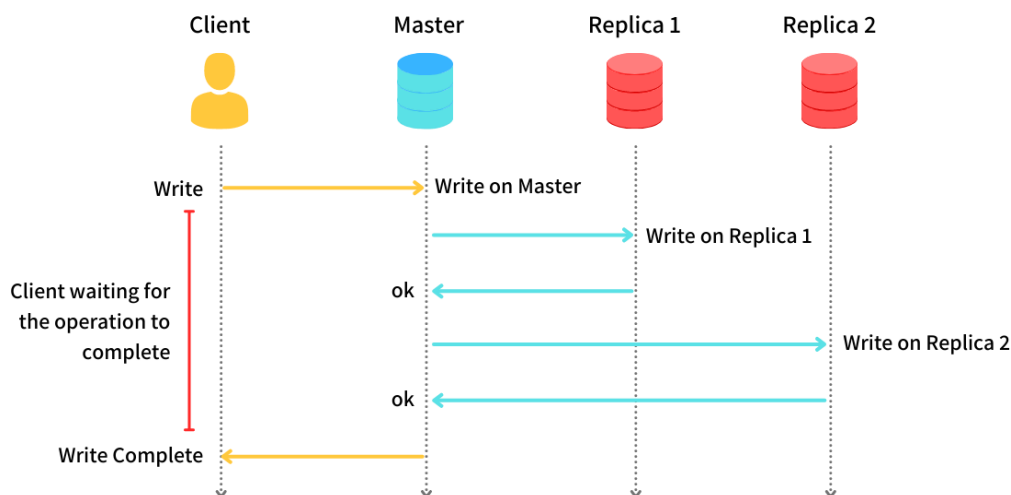
Synchronous Replication

In synchronous replication, the master node notifies all the replicas (sequentially/parallelly) and waits until all the slaves have confirmed the write operation to go through, before reporting success to the user and making the write visible to other queries.

Synchronous replication ensures that the replicas are always in sync with the master node. Even in case of a master node crash, replicas will still have consistently updated data making the system fault-tolerant by design. We can quickly promote any one of the slave nodes as the new master and continue to function as usual.

However, if a synchronous slave does not respond either due to a crash or network partition, write operations won't be processed. The master node needs to block all write operations till the replica comes back online.

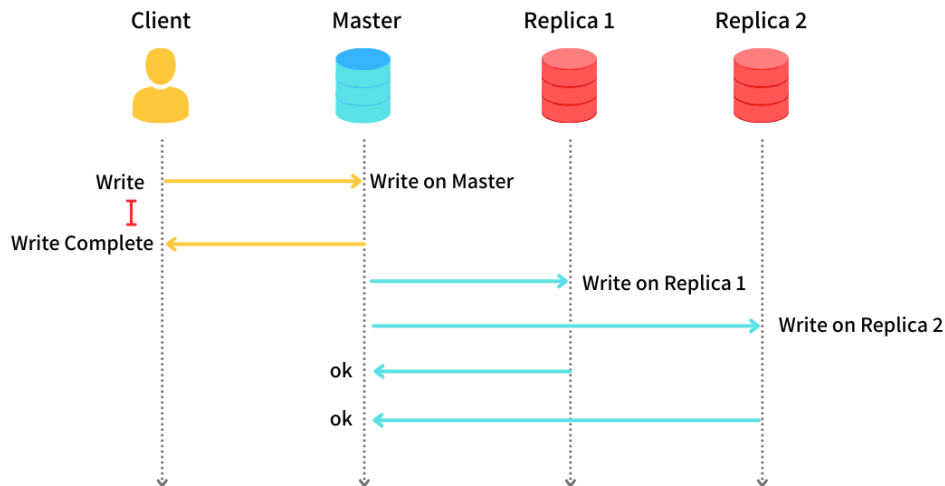
Ideally, it's recommended to have only one slave as a synchronous replica. This guarantees that we have an up-to-date copy of data on at least two nodes.



Synchronous Replication

Asynchronous Replication

In asynchronous replication, the master completes the write operation immediately and responds back to the client without waiting for the change to be propagated to the replicas. In this case, the clients will not be blocked for a long duration, increasing the overall throughput. If the master node crashes and is not recoverable, any writes that have not been propagated are lost. This means a write is not guaranteed to be durable even if it has been confirmed to the client.



Asynchronous Replication

AWS Example

Enable the Multi-AZ option for RDS and perform a failover (IP change)

Replication Variants

Statement based

Statement based replication works by recording queries that change data on the master. This means the leader logs every INSERT, UPDATE or DELETE statement in its binlog and sends them to replica. Replica instances reads the events from the relay log as actual SQL statements and reexecutes them as if it had been received from a client.

In theory, this will keep the replica in sync with the master and as we are using SQL statements, the binlog events tend to be compact and doesn't use a lot of bandwidth. (A query that updates gigabytes of data might add a few kb overhead in binlog). However there are several drawbacks:

- Any statement with a nondeterministic function like NOW(), RAND(), UUID(), is likely to generate different values on each node
- Statements with side-effects like triggers, stored procedures may result in different side-effects
- Statements depending on existing data in the database, must be executed in the same order on each replica. Thus all modifications must be serializable on replica, requiring more locking.

TODO: Add an example of a non-deterministic update using UUID() with statement based replication and show the inconsistency

Row-based

In case of row based replication, MySQL records the actual data changes in the binary log. This ensures every statement is replicated correctly. Row based replication can be extremely efficient and inefficient at the same time depending on the query.

```
INSERT INTO analytics(total)
SELECT sum(price) FROM very_large_invoice_table;
```

This query will scan many rows in the source table but will result in a single binlog event. In case of statement based replication, this query would have to be processed again on all the replica nodes.

```
UPDATE very_large_invoice_table SET discount = 0;
```

However, row based replication will be very expensive in this case as every row needs to be written to the binary log, making the binary log event extremely large.

MySQL can switch between statement based and row based replication dynamically if we set `binlog_format = MIXED`

Write Ahead Logs

TODO

Replication Lag

#TODO

Extra Examples

Example 3: Read Load Balancing: Multi node replicas

<https://shashwotrisal.medium.com/load-balancing-mysql-servers-using-proxysql-ff5ab6d1d4ea>

<https://towardsdatascience.com/high-availability-mysql-cluster-with-load-balancing-using-haproxy-and-heartbeat-40a16e134691>

Example 4: MySQL master, Postgres Slave: OLTP to OLAP

<https://medium.com/event-driven-utopia/a-visual-introduction-to-debezium-32563e23c6b8>

<https://medium.com/event-driven-utopia/configuring-debezium-to-capture-postgresql-changes-with-docker-compose-224742ca5372>

References


Kleppmann: Chapter 5

<https://scribe.rip/swlh/zero-downtime-master-slave-replication-4f2814138edf>

<https://datto.engineering/post/lossless-mysql-semi-sync-replication-and-automated-failover>

<https://severalnines.com/resources/whitepapers/mysql-replication-high-availability/>

<https://www.digitalocean.com/community/tutorials/how-to-set-up-replication-in-mysql>

 Turns out MySQL Statement-based Replication might not be a good idea, Lets discuss...

<https://engineering.fb.com/2021/07/22/data-infrastructure/mysql/>

<https://www.uber.com/en-IN/blog/postgres-to-mysql-migration/>

<https://arpitbhayani.me/blogs/replication-strategies>