Hands On Lab: Advanced Replication & High Availability

LAB 8.1: InnoDB Cluster Setup with 3 Nodes

Objective: Set up a production-grade InnoDB Cluster with automatic failover.

Prerequisites:

- Three MySQL Server 8.0+ instances
- Node 1: 192.168.1.100:3306
- Node 2: 192.168.1.101:3306
- Node 3: 192.168.1.102:3306
- MySQL Shell installed

Step-by-Step Instructions:

1. **Prepare MySQL Servers for InnoDB Cluster**

For each server, configure as follows:

```
sudo vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

Add under [mysqld]:

```
# InnoDB Cluster Configuration
server_id = 1  # Change to 2, 3 for other nodes
log_bin = /var/log/mysql/mysql-bin
binlog_format = ROW


# Group Replication
transaction_write_set_extraction = XXHASH64
loose_group_replication_group_name = "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee"
loose_group_replication_start_on_boot = OFF
loose_group_replication_local_address = "192.168.1.100:33061"  # Change IP
loose_group_replication_group_seeds =
"192.168.1.100:33061,192.168.1.101:33061,192.168.1.102:33061"
```

```
loose_group_replication_single_primary_mode = ON

# InnoDB Cluster
default_table_encryption = OFF
```

2. **Restart All MySQL Servers**

   ```
   sudo systemctl restart mysql
   ```

3. **Create InnoDB Cluster Admin User on Primary**

   ```
   CREATE USER 'icroot'@'%' IDENTIFIED BY 'IcPass123!';
   GRANT ALL ON *.* TO 'icroot'@'%' WITH GRANT OPTION;
   CREATE USER 'icroot'@'localhost' IDENTIFIED BY 'IcPass123!';
   GRANT ALL ON *.* TO 'icroot'@'localhost' WITH GRANT OPTION;
   FLUSH PRIVILEGES;
   ```

4. **Initialize InnoDB Cluster Using MySQL Shell**

   ```
   mysqlsh
   ```

In MySQL Shell:

```
// Connect to primary node
shell> \connect icroot@192.168.1.100:3306

// Create cluster
shell> var cluster = dba.createCluster("myCluster")

// Add members to cluster
shell> cluster.addInstance('icroot@192.168.1.101:3306')
shell> cluster.addInstance('icroot@192.168.1.102:3306')

// Get cluster status
shell> cluster.status()
```

```
// Exit MySQL Shell
shell> \quit
```

5. **Deploy MySQL Router for Connection Pooling**

```
# Install MySQL Router
sudo apt install -y mysql-router-community

# Configure router
sudo mysqlrouter --bootstrap icroot@192.168.1.100:3306 \
  --directory /etc/mysqlrouter

# Start router
sudo systemctl start mysqlrouter
sudo systemctl status mysqlrouter
```

6. **Connect Through MySQL Router**

```
# Connect to read-write port (6446)
mysql -h 127.0.0.1 -P 6446 -u icroot -p

# Connect to read-only port (6447)
mysql -h 127.0.0.1 -P 6447 -u icroot -p
```

7. **Test Automatic Failover**

a) **Create test data:**

```
CREATE DATABASE cluster_test;
CREATE TABLE cluster_test.data (
  id INT PRIMARY KEY AUTO_INCREMENT,
  value VARCHAR(255)
) ENGINE=InnoDB;
```

```
INSERT INTO cluster_test.data (value) VALUES ('Test data 1'), ('Test data 2');
```

b) **Simulate primary failure:**

```
# On primary node, stop MySQL
sudo systemctl stop mysql
```

c) **Verify automatic failover:**

```
# In MySQL Shell, check cluster status
mysqlsh
shell> \connect icroot@192.168.1.101:3306
shell> var cluster = dba.getCluster()
shell> cluster.status()
```

d) **Verify data availability:**

```
# Connect through router
mysql -h 127.0.0.1 -P 6446 -u icroot -p -e "SELECT * FROM cluster_test.data;"
```

8. **Monitor Cluster Health**

```
mysqlsh
shell> \connect icroot@192.168.1.101:3306
shell> var cluster = dba.getCluster()
shell> cluster.describe()
shell> cluster.status()
```

Hands-on Tasks:

- Set up 3-node InnoDB Cluster
- Configure MySQL Router
- Test automatic failover
- Monitor cluster status
- Document failover procedures

Solution:

1. Set up a 3-node MySQL InnoDB Cluster using MySQL Shell AdminAPI with one primary and two secondary instances.

1. Prerequisites and instance layout

Use three MySQL 8.x instances (ports are examples):

- Node1: localhost:3310 (seed / primary)
- Node2: localhost:3320
- Node3: localhost:3330

Ensure on all three:

```
[mysqld]
server_id=1/2/3            # unique per node
gtid_mode=ON
enforce_gtid_consistency=ON
binlog_format=ROW
log_bin=binlog
transaction_write_set_extraction=XXHASH64
loose-group_replication_bootstrap_group=OFF
loose-group_replication_start_on_boot=OFF
loose-group_replication_group_name="aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee"
loose-group_replication_local_address="127.0.0.1:33061/33062/33063"
loose-group_replication_group_seeds="127.0.0.1:33061,127.0.0.1:33062,127.0.0.1:33063"
loose-group_replication_single_primary_mode=ON
loose-group_replication_enforce_update_everywhere_checks=OFF
```

Restart all instances after editing config.

2. Create cluster admin user

On each node (or on Node1 and let AdminAPI create internal accounts):

```
CREATE USER 'cluster_admin'@'%' IDENTIFIED BY 'ClusterPass2026!';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'cluster_admin'@'%' WITH GRANT OPTION;
FLUSH PRIVILEGES;
```

3. Check and configure instances with MySQL Shell

Start MySQL Shell in JS mode:

```
mysqlsh --js
```

On Node1:

```
shell.connect('cluster_admin@localhost:3310');

dba.checkInstanceConfiguration('cluster_admin@localhost:3310');
dba.configureInstance('cluster_admin@localhost:3310');

dba.checkInstanceConfiguration('cluster_admin@localhost:3320');
dba.configureInstance('cluster_admin@localhost:3320');

dba.checkInstanceConfiguration('cluster_admin@localhost:3330');
dba.configureInstance('cluster_admin@localhost:3330');
```

Confirm checks report “instance configuration is suitable” or that required changes were applied.

4. Create the InnoDB Cluster

Still in MySQL Shell connected to Node1:

```
var cluster = dba.createCluster(
  'TrainingCluster',
  { localAddress: 'localhost:3310' }
);
```

This:

- Deploys cluster metadata on Node1
- Configures Group Replication and starts it on Node1

Check status:

```
cluster.status();
```

Node1 should show as PRIMARY with mode R/W.

5. Add the other two nodes

From the same shell:

```
cluster.addInstance('cluster_admin@localhost:3320', {
  localAddress: 'localhost:3320',
  recoveryMethod: 'clone'
});

cluster.addInstance('cluster_admin@localhost:3330', {
  localAddress: 'localhost:3330',
  recoveryMethod: 'clone'
});
```

Wait for each clone to finish. Then:

```
cluster.status({extended: true});
```

You should see:

- Node1: PRIMARY, ONLINE
- Node2: SECONDARY, ONLINE
- Node3: SECONDARY, ONLINE

6. Basic functional tests

6.1 Test writes on primary and reads on all nodes

From MySQL Shell, connect to primary:

```
shell.connect('cluster_admin@localhost:3310');

session.runSql("CREATE DATABASE IF NOT EXISTS cluster_lab;");

session.runSql("USE cluster_lab;");

session.runSql(`
  CREATE TABLE employees (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  )
`);

session.runSql("INSERT INTO employees (name) VALUES ('Node1-insert');");
```

On Node2 and Node3 (SQL mode):

```
shell.connect('cluster_admin@localhost:3320');

session.runSql("SELECT @@server_id, @@read_only, COUNT(*) FROM
cluster_lab.employees;");


shell.connect('cluster_admin@localhost:3330');

session.runSql("SELECT @@server_id, @@read_only, COUNT(*) FROM
cluster_lab.employees;");
```

Row counts should match and secondaries should be read_only = 1.

6.2 Test automatic primary election

Simulate failure of Node1:

```
# Stop mysqld on Node1

sudo systemctl stop mysql@3310
```

Back in MySQL Shell (connected to one of surviving nodes):

```
shell.connect('cluster_admin@localhost:3320');

cluster = dba.getCluster('TrainingCluster');

cluster.status();
```

One of Node2/Node3 should now be PRIMARY and writable.

7. (Optional) Sandbox deployment helper

For purely local hands-on, AdminAPI can create sandboxes for you:

```
mysqlsh --dba deploy-sandbox-instance 3310 --clusterAdmin=cluster_admin --
clusterAdminPassword=ClusterPass2026!

mysqlsh --dba deploy-sandbox-instance 3320 --clusterAdmin=cluster_admin --
clusterAdminPassword=ClusterPass2026!

mysqlsh --dba deploy-sandbox-instance 3330 --clusterAdmin=cluster_admin --
clusterAdminPassword=ClusterPass2026!
```

Then:

```
shell.connect('cluster_admin@localhost:3310');

var cluster = dba.createCluster('TrainingCluster');

cluster.addInstance('cluster_admin@localhost:3320');

cluster.addInstance('cluster_admin@localhost:3330');

cluster.status();
```

8. Verification checklist

- All 3 instances have GTID, binlog, and group_replication settings correct.
- dba.checkInstanceConfiguration() passes for each node.
- cluster.status() shows 1 PRIMARY, 2 SECONDARY in ONLINE state.
- Writes on primary replicate to both secondaries.
- Stopping primary triggers automatic primary election on a secondary.

This completes a full 3-node InnoDB Cluster lab setup using MySQL Shell AdminAPI.

2. Configure MySQL Router to front your 3-node InnoDB Cluster and expose stable R/W and R/O endpoints for applications.

1. Install MySQL Router

On the application (or separate) host:

```
# Ubuntu / Debian
sudo apt update
sudo apt install mysql-router

# Verify
mysqlrouter --version
```

2. Create router account in the cluster

In MySQL Shell connected to the cluster primary (e.g., localhost:3310):

```
mysqlsh --js
shell.connect('cluster_admin@localhost:3310');

cluster = dba.getCluster('TrainingCluster');

// Create a low-privilege router account
cluster.setupRouterAccount('router_user', {password: 'RouterPass2026!'});
```

This creates the account and grants only the privileges needed for metadata and health checks.

3. Bootstrap MySQL Router

On the router host (can be same as primary for a lab):

```
cd ~  # or /opt
mysqlrouter \
  --bootstrap router_user@localhost:3310 \
  --directory myrouter \
  --user $USER
```

You will be prompted for RouterPass2026!.

Bootstrap will:

- Connect to the InnoDB Cluster via the specified node.
- Read metadata and topology.
- Create myrouter/mysqlrouter.conf.
- Generate helper scripts start.sh, stop.sh (in sandbox mode).

4. Inspect generated configuration

```
cd myrouter
ls
# mysqlrouter.conf, start.sh, stop.sh, run/ logs/ etc.
```

Open mysqlrouter.conf:

```
[DEFAULT]
user = youruser
logging_folder = /home/youruser/myrouter/log

[logger]
level = INFO

[routing:primary_rw]
```

```
bind_address = 0.0.0.0

bind_port = 6446

destinations = metadata-cache://TrainingCluster/?role=PRIMARY

routing_strategy = first-available


[routing:secondary_ro]

bind_address = 0.0.0.0

bind_port = 6447

destinations = metadata-cache://TrainingCluster/?role=SECONDARY

routing_strategy = round-robin
```

Ports (default sandbox):

- 6446 – Classic protocol R/W
- 6447 – Classic protocol R/O
- 6448 – X protocol R/W
- 6449 – X protocol R/O

5. Start MySQL Router

For lab/sandbox mode:

```
cd ~/myrouter

./start.sh  # or: mysqlrouter -c mysqlrouter.conf &
```

Check it is listening:

```
ss -ltnp | grep 644

# or

netstat -ltnp | grep 644
```

6. Test routing behavior

6.1 Read-write endpoint

```
# Connect through router R/W port
mysql -h 127.0.0.1 -P 6446 -u app_user -p

SELECT @@hostname, @@port, @@global.read_only;
-- Should show current PRIMARY node, read_only = 0

USE cluster_lab;
INSERT INTO employees(name) VALUES('via_router_rw');
SELECT COUNT(*) FROM employees;
```

6.2 Read-only endpoint

```
mysql -h 127.0.0.1 -P 6447 -u app_user -p

SELECT @@hostname, @@port, @@global.read_only;
-- Should show a SECONDARY, read_only = 1

SELECT * FROM cluster_lab.employees ORDER BY id DESC LIMIT 5;
-- Read succeeds; writes fail:
INSERT INTO employees(name) VALUES('should_fail');
-- ERROR: read-only or access denied
```

7. Validate failover with Router

1. **Confirm current primary via router:**

```
mysql -h 127.0.0.1 -P 6446 -u app_user -p -e "SELECT @@hostname, @@port,
@@global.read_only;"
```

2. **Stop the current primary MySQL instance (e.g., 3310):**

```
sudo systemctl stop mysql@3310
```

3. **Wait for Group Replication to elect a new primary, then test again:**

```
mysql -h 127.0.0.1 -P 6446 -u app_user -p -e "SELECT @@hostname, @@port,
@@global.read_only;"
```

You should see a different host/port, with read_only = 0. Router transparently redirects R/W traffic to the new primary.

8. Optional: Router as a daemon/service

To run Router as a service using the generated config:

```
sudo mysqlrouter -c /home/youruser/myrouter/mysqlrouter.conf --user mysqlrouter_user --
daemon
```

Or create a systemd unit pointing to mysqlrouter -c /path/to/mysqlrouter.conf.

9. Lab verification checklist

- cluster.setupRouterAccount() executed successfully.
- mysqlrouter --bootstrap completed and mysqlrouter.conf created.
- R/W port 6446 routes to primary, writes succeed.
- R/O port 6447 routes to secondaries, writes blocked.
- After primary shutdown, Router redirects R/W connections to new primary.
- Logs (myrouter/log/mysqlrouter.log) show healthy metadata refresh and no fatal errors.

This completes a full MySQL Router configuration lab for your 3-node InnoDB Cluster.

3. Test automatic failover in your 3-node InnoDB Cluster and verify that MySQL Router transparently reconnects clients to the new primary.

1. Pre-check: cluster and router health

In MySQL Shell (JS), connect via Router R/W port:

```
mysqlsh --js root@localhost:6446
```

Check cluster status:

```
cluster = dba.getCluster('TrainingCluster');
cluster.status({extended: true});
```

Confirm:

- 3 members ONLINE
- One PRIMARY, two SECONDARY
- singlePrimaryMode: true

Note the current primary, e.g. localhost:3310.

2. Open a client session through Router

From a separate terminal, connect through the R/W port:

```
mysql -h 127.0.0.1 -P 6446 -u app_user -p
```

In that session:

```
SELECT @@hostname AS server, @@port AS port, @@global.read_only AS read_only;
-- Should show current PRIMARY with read_only = 0
```

Keep this client session open; you will use it to observe failover.

3. Start a continuous workload

Still on the client through Router:

```
USE cluster_lab;


-- Create simple table if not present
CREATE TABLE IF NOT EXISTS ha_test (
  id INT PRIMARY KEY AUTO_INCREMENT,
  msg VARCHAR(100),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB;


-- Simple loop (in bash) to generate writes via Router
\! while true; do \
    mysql -h 127.0.0.1 -P 6446 -u app_user -p'yourpass' -D cluster_lab \
      -e "INSERT INTO ha_test(msg) VALUES(CONCAT('write-', NOW()));"; \
    sleep 1; \
  done
```

This simulates a live application writing every second.

4. Trigger automatic failover by stopping the primary

From another terminal on the primary host (e.g. instance on 3310):

```
# Stop only the primary MySQL instance
sudo systemctl stop mysql@3310   # or appropriate service name
```

Alternatively (lab, not prod):

```
mysqladmin -h 127.0.0.1 -P 3310 -u root -p shutdown
```

Group Replication detects the loss and automatically elects a new primary from remaining members.

5. Observe behavior from the client

In the Router-connected client:

- You may see one transient error (connection lost or timeout) during election.
- Then inserts resume automatically once Router reconnects to the new primary.

Confirm which server you are now writing to:

```
SELECT @@hostname AS server, @@port AS port, @@global.read_only AS read_only;
-- Should now show one of 3320/3330 with read_only = 0
SELECT COUNT(*) FROM ha_test;
```

The count should continue increasing; there must be no manual change in connection string (still pointing to 127.0.0.1:6446).

6. Verify cluster sees the new primary

Back in MySQL Shell:

```
cluster = dba.getCluster('TrainingCluster');
cluster.status({extended: true});
```

You should see:

- Old primary (3310) as UNREACHABLE or OFFLINE
- A surviving member (3320 or 3330) promoted to PRIMARY
- Remaining node as SECONDARY

7. Optional: Bring back the old primary

Restart the stopped instance:

```
sudo systemctl start mysql@3310
```

Then rejoin to the cluster (if not automatic):

```
shell.connect('cluster_admin@localhost:3320');  // current primary
```

```
cluster = dba.getCluster('TrainingCluster');

cluster.rejoinInstance('cluster_admin@localhost:3310');

cluster.status();
```

Old primary should come back as SECONDARY.

8. Failover test checklist

- Before test: 3 nodes ONLINE, 1 PRIMARY, 2 SECONDARY.
- Router R/W port 6446 returns PRIMARY and accepts writes.
- After stopping primary, cluster.status() shows new PRIMARY elected.
- Router-connected client resumes writes without changing connection string.
- SELECT @@hostname via Router shows new PRIMARY.
- Bringing back old primary re-adds it as SECONDARY, not PRIMARY.

This lab confirms automatic failover of InnoDB Cluster and transparent reconnection through MySQL Router.

4. Monitor MySQL InnoDB Cluster status using MySQL Shell’s AdminAPI, including real-time watch and key health indicators.

1. Basic cluster status check

Connect MySQL Shell (JS) to any ONLINE instance (preferably the primary):

```
mysqlsh --js cluster_admin@localhost:3310
```

Get a cluster handle and show status:

```
var cluster = dba.getCluster('TrainingCluster');  // or dba.getCluster()
cluster.status();
```

Typical output (trimmed):

```
{
  "clusterName": "TrainingCluster",
  "defaultReplicaSet": {
    "primary": "localhost:3310",
    "status": "OK",
    "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
    "topology": {
      "localhost:3310": { "mode": "R/W", "status": "ONLINE" },
      "localhost:3320": { "mode": "R/O", "status": "ONLINE" },
      "localhost:3330": { "mode": "R/O", "status": "ONLINE" }
    }
  }
}
```

Key fields:

- status: OK, OK_NO_TOLERANCE, DEGRADED, UNREACHABLE, etc.

- statusText: human explanation (tolerance, failures).
- primary: current primary (single-primary mode).
- topology: each member’s status (ONLINE, OFFLINE, RECOVERING) and mode (R/W, R/O).

2. Extended status for deeper monitoring

Use extended mode for more detail:

```
cluster.status({extended: 1});
```

Higher levels (1–3) add:

- recovery information (clone or incremental, progress %).
- applierStatus and receiverStatus for Group Replication threads.
- GTID and replication lag indicators per instance.

For full detail:

```
cluster.status({extended: 3});
```

Use this during node recovery or troubleshooting.

3. Real-time monitoring with \watch

MySQL Shell can auto-refresh any command output:

```
\watch 5 cluster.status()
```

- Refreshes cluster status every 5 seconds.
- Use when simulating failures (stopping a node, testing failover) to observe topology changes live.

Example during a node outage:

- statusText changes to “Cluster is ONLINE but can only tolerate up to ZERO failures.”
- One member shows status: "UNREACHABLE" or OFFLINE.

4. Quick CLI health checks

For scripting from Bash (no interactive JS):

```
mysqlsh --js cluster_admin@localhost:3310 -e "dba.getCluster('TrainingCluster').status();" > cluster_status.json
```

Or a concise health summary:

```
mysqlsh --js cluster_admin@localhost:3310 -e "

  var c = dba.getCluster('TrainingCluster');

  var s = c.status();

  print('Status: ' + s.defaultReplicaSet.status);

  print('Primary: ' + s.defaultReplicaSet.primary);

"
```

Use this in cron or monitoring agents.

5. What to monitor (lab checklist)

Run regularly:

```
var s = cluster.status();

s.defaultReplicaSet.status;       // Should be "OK"

s.defaultReplicaSet.primary;      // Current primary

s.defaultReplicaSet.topology;     // Per-node status/mode
```

In your lab, verify:

- status is OK, statusText says cluster is ONLINE and can tolerate at least one failure.
- Exactly one node shows mode: "R/W" (primary), others R/O.
- When you stop one node, status reflects reduced tolerance and node status changes.
- When node is restarted and rejoined, all return to ONLINE.

This gives a complete, hands-on monitoring solution for your InnoDB Cluster using MySQL Shell.

LAB 8.2: Replication Monitoring and Troubleshooting

Objective: Monitor replication and troubleshoot common issues.

Step-by-Step Instructions:

1. **Monitor Replication Status**

   ```
   -- On Slave
   SHOW SLAVE STATUS\G


   -- Key fields to monitor:
   -- Slave_IO_Running: Yes
   -- Slave_SQL_Running: Yes
   -- Seconds_Behind_Master: 0 (or acceptable lag)
   -- Last_Errno: 0 (no errors)
   -- Last_Error: (should be empty)
   ```

2. **Check Master Binary Log Position**

   ```
   -- On Master
   SHOW MASTER STATUS;
   SHOW BINARY LOGS;
   ```

3. **Monitor Slave Replication Threads**

   ```
   -- On Slave
   SHOW PROCESSLIST;


   -- Look for:
   -- Slave I/O: Reading event from the master
   -- Slave SQL: Executing event received from master
   ```

4. **Check Replication Lag**

   ```
   -- Method 1: Using SHOW SLAVE STATUS
   SHOW SLAVE STATUS\G
   -- Look for: Seconds_Behind_Master
   ```

```
-- Method 2: Using Performance Schema
SELECT * FROM performance_schema.replication_applier_status\G
```

5. **Simulate Replication Error**

a) **On Slave, insert conflicting data:**

```
INSERT INTO replication_test.test_table (id, data)
VALUES (1, 'Conflicting data on slave');
```

b) **On Master, update same row:**

```
UPDATE replication_test.test_table SET data = 'Updated on master'
WHERE id = 1;
```

c) **Check for errors on Slave:**

```
SHOW SLAVE STATUS\G
-- Look for error in Last_Error field
```

6. **Skip Replication Error (if needed)**

```
-- Option 1: Skip specific number of events
SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1;
START SLAVE;

-- Option 2: Using GTID (preferred)
-- Don't use SQL_SLAVE_SKIP_COUNTER with GTID
-- Instead, use:
SET GTID_NEXT='current_master_gtid:next_number';
BEGIN;
COMMIT;
SET GTID_NEXT='AUTOMATIC';
```

7. **Use Percona Toolkit for Replication Monitoring**

```
# Install Percona Toolkit
sudo apt install -y percona-toolkit
```

```
# Check table consistency
pt-table-checksum --host=192.168.1.100 \
  --user=root --password=YourPassword \
  --databases=replication_test \
  --replicate=percona.checksums

# View results
mysql -u root -p -e "SELECT * FROM percona.checksums;"

# Fix differences
pt-table-sync --execute \
  h=192.168.1.100,u=root,p=YourPassword \
  --replicate=percona.checksums
```

8. **Monitor with Performance Schema**

```
-- View replication channels
SELECT * FROM performance_schema.replication_connection_configuration\G
SELECT * FROM performance_schema.replication_applier_configuration\G

-- View connection status
SELECT * FROM performance_schema.replication_connection_status\G
SELECT * FROM performance_schema.replication_applier_status\G

-- View worker status (for parallel replication)
SELECT * FROM performance_schema.replication_applier_status_by_worker\G
```

9. **Set up Automated Monitoring**

```
# Create monitoring script
cat > /usr/local/bin/monitor_replication.sh << 'EOF'
```

```
#!/bin/bash

SLAVE_HOST="192.168.1.101"
SLAVE_USER="root"
SLAVE_PASS="YourPassword"

SECONDS_BEHIND=$(mysql -h$SLAVE_HOST -u$SLAVE_USER -p$SLAVE_PASS \
  -sNe "SHOW SLAVE STATUS\G" | grep "Seconds_Behind_Master" | awk '{print
$NF}')

IO_RUNNING=$(mysql -h$SLAVE_HOST -u$SLAVE_USER -p$SLAVE_PASS \
  -sNe "SHOW SLAVE STATUS\G" | grep "Slave_IO_Running" | awk '{print $NF}')

SQL_RUNNING=$(mysql -h$SLAVE_HOST -u$SLAVE_USER -p$SLAVE_PASS \
  -sNe "SHOW SLAVE STATUS\G" | grep "Slave_SQL_Running" | awk '{print $NF}')

echo "$(date): IO=$IO_RUNNING, SQL=$SQL_RUNNING, Lag=${SECONDS_BEHIND}s"

if [[ "$IO_RUNNING" != "Yes" ]] || [[ "$SQL_RUNNING" != "Yes" ]]; then
  echo "ERROR: Replication issue detected!" | mail -s "Replication Alert"
admin@example.com
fi
EOF

chmod +x /usr/local/bin/monitor_replication.sh

# Add to crontab
crontab -e
```

*# Add: */5 * * * * /usr/local/bin/monitor_replication.sh >> /var/log/replication_monitor.log*

Hands-on Tasks:

- Monitor replication continuously
- Simulate and resolve replication errors
- Check data consistency between master and slave
- Document monitoring procedures

Solution:

1. Monitor MySQL replication continuously using a combination of SHOW REPLICA STATUS, Performance Schema, and a lightweight watch/alert script.

1. Simple continuous monitor using MySQL Shell

On the replica:

mysqlsh --sql root@replica_host:3306

Then:

\watch 5 **SHOW** REPLICA **STATUS**\G

This:

- Refreshes replication status every 5 seconds.
- Lets you watch Replica_IO_Running, Replica_SQL_Running, Seconds_Behind_Source live.

Key fields to keep an eye on:

- Replica_IO_Running / Slave_IO_Running = Yes
- Replica_SQL_Running / Slave_SQL_Running = Yes
- Seconds_Behind_Source / Seconds_Behind_Master ≈ 0

2. Performance Schema-based monitoring query

For more detailed, queryable metrics (MySQL 8+):

```sql
SELECT
  rcs.channel_name,
  rcs.service_state AS io_state,
  rcs.last_heartbeat_timestamp,
  ras.service_state AS sql_state,
  ras.last_error_number,
  ras.last_error_message
FROM performance_schema.replication_connection_status rcs
JOIN performance_schema.replication_applier_status ras
  ON rcs.channel_name = ras.channel_name;
```

- Run this periodically (every N seconds) to track whether I/O and SQL threads are running and see last error messages.

3. Lightweight Bash monitor loop

Create monitor_replication.sh on the replica:

```bash
#!/bin/bash
HOST="127.0.0.1"
USER="root"
PASS="yourpass"
INTERVAL=5
MAX_LAG=30  # seconds

while true; do
  STATUS=$(mysql -h"$HOST" -u"$USER" -p"$PASS" -e "SHOW REPLICA STATUS\G" 2>/dev/null)
  IO=$(echo "$STATUS" | grep -E 'Replica_IO_Running:|Slave_IO_Running:' | awk '{print $2}')
  SQL=$(echo "$STATUS" | grep -E 'Replica_SQL_Running:|Slave_SQL_Running:' | awk '{print $2}')
  LAG=$(echo "$STATUS" | grep -E 'Seconds_Behind_Source:|Seconds_Behind_Master:' | awk '{print $2}')

  TS=$(date '+%Y-%m-%d %H:%M:%S')
  echo "[$TS] IO=$IO SQL=$SQL LAG=${LAG}s"

  if [ "$IO" != "Yes" ] || [ "$SQL" != "Yes" ]; then
    echo "[$TS] CRITICAL: Replication threads stopped! IO=$IO SQL=$SQL" | logger -t mysql-repl
  elif [ "$LAG" = "NULL" ] || [ "$LAG" -gt "$MAX_LAG" ]; then
    echo "[$TS] WARNING: Replication lag high or unknown (LAG=$LAG)" | logger -t mysql-repl
  fi

  sleep "$INTERVAL"
done
```

```
chmod +x monitor_replication.sh
./monitor_replication.sh
```

This gives a continuous stream of replication health lines with basic alerting via syslog.

4. Lag-focused continuous monitor (Performance Schema)

For multi-threaded replicas, use Performance Schema for more accurate lag:

```
\watch 5
SELECT
  channel_name,
  COUNT_TRANSACTIONS_REMOTE_IN_APPLIER_QUEUE AS tx_in_queue
FROM performance_schema.replication_applier_status_by_coordinator;
```

- If tx_in_queue grows and stays high, your replica is falling behind even if Seconds_Behind_Source looks small.

5. Lab verification steps

In your lab, do this sequence:

1. Start monitor_replication.sh on the replica.
2. On the master, run a write loop:

```
while true; do
  mysql -u root -p -e "INSERT INTO test_backup_restore.rep_monitor(msg) VALUES(NOW());"
  sleep 1
done
```

3. Watch the monitor output:
   - IO/SQL Yes, lag stays at 0–1 seconds.
4. Temporarily stop SQL thread on replica:

```
STOP REPLICA SQL_THREAD;
```

5. Monitor should show:
   - SQL=No, lag becomes NULL, critical alert lines.

6. Start SQL thread again:

START REPLICA SQL_THREAD;

7. Monitor returns to healthy state.

That gives you a complete, hands-on continuous replication monitoring setup suitable for lab and easily adaptable to production.

2. Simulate common MySQL replication errors (duplicate key and missing row) and practice safe ways to diagnose and fix them without corrupting data.

1. Setup: simple replicated table

On master:

CREATE DATABASE IF NOT **EXISTS** repl_lab;

USE repl_lab;

CREATE TABLE accounts (

  id **INT PRIMARY KEY**,

  balance **INT** NOT NULL

) **ENGINE**=**InnoDB**;

INSERT INTO accounts **VALUES** (1,100), (2,200), (3,300);

Verify data on slave:

USE repl_lab;

SELECT * **FROM** accounts **ORDER BY** id;

2. Simulate error 1062: duplicate entry on slave

2.1 Manually create conflicting row on slave

On slave (bypass replication, simulate bad manual write):

USE repl_lab;

INSERT INTO accounts **VALUES** (4,400);   -- *Create row that master does NOT have*

2.2 Generate conflicting event from master

On master:

USE repl_lab;

INSERT INTO accounts **VALUES** (4,500);   -- *Replicates to slave and conflicts*

On slave, check status:

SHOW REPLICA **STATUS**\G   -- *or SHOW SLAVE STATUS\G*

You should see:

- Last_SQL_Errno: 1062
- Last_SQL_Error: Duplicate entry '4' for key 'PRIMARY' ...
- Replica_SQL_Running: No

3. Fix error 1062 safely (preferred)

Goal: make slave match master, then resume replication.

3.1 Inspect offending row and relay log position

On slave:

SHOW REPLICA **STATUS**\G

-- *Note: Relay_Master_Log_File, Exec_Master_Log_Pos, Last_SQL_Error*

SELECT * **FROM** repl_lab.accounts **WHERE** id = 4;

You now know slave has a wrong id=4 row.

3.2 Remove bad row and restart SQL thread

On slave:

```
STOP REPLICA SQL_THREAD;    -- or STOP SLAVE SQL_THREAD;


DELETE FROM repl_lab.accounts WHERE id = 4;


START REPLICA SQL_THREAD;   -- or START SLAVE SQL_THREAD;
```

Check status again:

```
SHOW REPLICA STATUS\G
-- Last_SQL_Errno: 0
-- Replica_SQL_Running: Yes
```

Verify:

```
SELECT * FROM repl_lab.accounts ORDER BY id;
-- Slave now has id=4 with balance=500, matching master
```

This is safer than skipping events because you fix the data and let the event replay correctly.

4. Simulate error 1032: “row not found” on slave

Now simulate the reverse: slave is missing a row master thinks exists.

4.1 Create row on master

On master:

```
USE repl_lab;
INSERT INTO accounts VALUES (5,500);
```

Wait for replication, then on slave:

```
SELECT * FROM accounts WHERE id=5;  -- Confirm present
```

4.2 Delete row only on slave

On slave:

DELETE FROM accounts **WHERE** id=5;

4.3 Update row on master

On master:

UPDATE accounts **SET** balance = balance + 50 **WHERE** id=5;

Replication tries to update id=5 on slave, which no longer exists.

On slave:

SHOW REPLICA **STATUS**\G

You should see:

- Last_SQL_Errno: 1032
- Last_SQL_Error: Could not execute Update_rows event ... Can't find record in 'accounts'
- Replica_SQL_Running: No

5. Fix error 1032 safely

5.1 Capture failing event information

On slave:

SHOW REPLICA **STATUS**\G

-- Note: Last_SQL_Error, Relay_Master_Log_File, Exec_Master_Log_Pos

5.2 Recreate missing row to match master

Check master for correct data:

-- On master

SELECT * **FROM** repl_lab.accounts **WHERE** id=5;

-- Suppose result is (5,550)

On slave:

```
STOP REPLICA SQL_THREAD;


INSERT INTO repl_lab.accounts VALUES (5,550);


START REPLICA SQL_THREAD;
```

Verify:

```
SHOW REPLICA STATUS\G
-- Last_SQL_Errno: 0, Replica_SQL_Running: Yes
SELECT * FROM repl_lab.accounts WHERE id=5;
```

6. When (and when NOT) to use sql_slave_skip_counter

You can also skip a single failing event, but this risks divergence if misused.

On slave:

```
STOP REPLICA SQL_THREAD;
SET GLOBAL sql_slave_skip_counter = 1;
START REPLICA SQL_THREAD;
```

Use this only when:

- You have confirmed the event is safe to ignore (e.g., duplicate insert that's already applied and data matches master).
- Ideally after validating the table with a tool like pt-table-checksum.

Do not use it blindly in loops or with slave-skip-errors in my.cnf in production.

7. Lab summary script

You can turn the above into a repeatable lab script for students:

1. Prepare table and initial data.

2. Run a shell script that:
    - Performs conflicting manual changes on slave.
    - Applies changes on master.
    - Shows SHOW REPLICA STATUS\G.
    - Guides them through:
        - STOP REPLICA SQL_THREAD
        - Fix data (DELETE / INSERT)
        - START REPLICA SQL_THREAD
3. End with a consistency check:

-- On both master and slave

SELECT COUNT(*) **AS rows**, SUM(balance) **AS** total_balance **FROM** repl_lab.accounts;

If counts and totals match, replication is healthy again.

This lab walks through both simulating and safely resolving the two most common replication errors: 1062 and 1032, without resorting to dangerous global skip settings.

3. Check data consistency between master and slave by comparing counts and checksums at SQL level, then validating with pt-table-checksum for full-table drift detection.

1. Quick SQL-level consistency checks

1.1 Row counts per table

On master:

USE repl_lab; *-- or your application DB*

SELECT 'master' **AS** server, table_name, table_rows

FROM information_schema.**tables**

WHERE table_schema = 'repl_lab'

ORDER BY table_name;

On slave:

USE repl_lab;

SELECT 'slave' **AS** server, table_name, table_rows

FROM information_schema.**tables**

WHERE table_schema = 'repl_lab'

ORDER BY table_name;

Compare outputs; row counts should match for each table. Small differences here mean definite inconsistency, but equality does not guarantee identical data.

1.2 CHECKSUM TABLE on key tables

Pick a few important tables (e.g., accounts, orders).

On master:

USE repl_lab;

CHECKSUM **TABLE** accounts, orders;

On slave:

USE repl_lab;

CHECKSUM **TABLE** accounts, orders;

Compare Checksum values per table; they must be identical.

2. Use Percona pt-table-checksum (recommended for full drift detection)

Install Percona Toolkit on the master host:

sudo apt update

sudo apt install percona-toolkit

pt-table-checksum --version

2.1 Run checksum from master (single command)

On master:

```
pt-table-checksum \
  --host=master_host \
  --user=root \
  --password=YOURPASS \
  --databases=repl_lab \
  --replicate=percona.checksums \
  --no-check-binlog-format \
  --empty-replicate-table
```

What this does:

- Runs chunked checksum queries on master for each table in repl_lab.
- Writes checksum for each chunk into percona.checksums on master.
- Those writes replicate to slaves, so percona.checksums is also created/updated there.

Sample output row (per table):

```
TS  ERRORS  DIFFS  ROWS  CHUNKS  SKIPPED  TIME  TABLE
12-27T11:10:01   0   0   1000     4      0     0.123 repl_lab.accounts
12-27T11:10:02   0   1   5000    10      0     0.456 repl_lab.orders
```

- DIFFS = 0 means no drift detected for that table.
- DIFFS > 0 means at least one chunk differs between master and slave(s).

2.2 Summarize differences per table

On master (or any slave) query the checksum table:

```
SELECT
  db,
  tbl,
  SUM(this_cnt) AS total_rows,
  SUM(master_cnt <> this_cnt OR master_crc <> this_crc
      OR ISNULL(master_crc) <> ISNULL(this_crc)) AS differing_chunks
FROM percona.checksums
GROUP BY db, tbl
HAVING differing_chunks > 0;
```

If zero rows returned, all tables are consistent for all replicas.

3. Drill into which slave is inconsistent

If you have multiple slaves, pt-table-checksum tracks per replica.

Show tables and replicas with differences:

```
pt-table-checksum \
  --replicate=percona.checksums \
  --replicate-check-only \
  --host=master_host \
  --user=root \
  --password=YOURPASS
```

This prints only tables with differences and identifies which slave(s) differ.

4. Lab steps to demonstrate consistency and drift

4.1 Baseline: clean, consistent replication

1. Ensure replication is running and healthy.

2. Run:

```
pt-table-checksum ... --databases=repl_lab
```

3. Confirm DIFFS=0 for all tables.

4.2 Simulate drift and detect it

On slave, make a bad manual change:

```
USE repl_lab;
UPDATE accounts SET balance = balance + 999 WHERE id = 1;
INSERT INTO accounts VALUES (999, 12345);
```

Run checksum again on master:

```
pt-table-checksum ... --databases=repl_lab
```

Now DIFFS for repl_lab.accounts should be 1, and the percona.checksums table will show differing chunks.

Check specific differences:

```
SELECT db, tbl, chunk, this_cnt, master_cnt, this_crc, master_crc
FROM percona.checksums
WHERE db='repl_lab' AND tbl='accounts'
  AND (master_cnt <> this_cnt OR master_crc <> this_crc);
```

5. (Optional) Auto-fix drift with pt-table-sync

Once you've identified drift, you can fix it by replaying changes from master to slave using pt-table-sync.

Example for one slave:

```
pt-table-sync \
  --execute \
  --replicate=percona.checksums \
  --databases=repl_lab \
  --host=master_host \
  --user=root \
  --password=YOURPASS
```

- Generates and executes INSERT/UPDATE/DELETE statements on master that replicate to the slave to fix the differences.
- Always test with --print first instead of --execute in real environments.

6. Minimal "no-tool" alternative (small lab)

If you don't want Percona Toolkit in a lab, you can do a simple checksum per table:

On master:

```
SELECT
  'accounts' AS tbl,
  COUNT(*) AS rows,
  SUM(CRC32(CONCAT_WS('#', id, balance))) AS checksum
FROM repl_lab.accounts;
```

On slave, run the same and compare rows and checksum. Repeat for other tables. Crude but works for teaching purposes.

7. Lab verification checklist

- Row counts match between master and slave for all tables.
- CHECKSUM TABLE values match for key tables.
- pt-table-checksum run from master completes with no errors.
- percona.checksums shows DIFFS=0 for all tables when replication is healthy.
- After intentional drift on slave, pt-table-checksum flags the affected table.
- (Optional) pt-table-sync returns tables to a consistent state.

This gives a complete, hands-on procedure to verify and understand data consistency between master and slave, from quick checks to full tooling.