Hands On Lab: Installation & Configuration

LAB 2.1: MySQL Configuration & Parameter Tuning

Objective: Configure MySQL settings for optimal performance and monitor configuration changes.

Prerequisites:

- MySQL Server installed and running

Step-by-Step Instructions:

1. **Locate MySQL Configuration File**

   sudo find / -name "my.cnf" -o -name "mysqld.cnf" 2>/dev/null

Typical locations:

   - /etc/mysql/my.cnf
   - /etc/mysql/mysql.conf.d/mysqld.cnf
   - /etc/my.cnf

2. **Backup Original Configuration**

   sudo cp /etc/mysql/mysql.conf.d/mysqld.cnf /etc/mysql/mysql.conf.d/mysqld.cnf.backup

3. **View Current MySQL Configuration**

   sudo cat /etc/mysql/mysql.conf.d/mysqld.cnf

4. **Connect to MySQL and Check Current Variables**

   SHOW VARIABLES LIKE '%max_connections%';

   SHOW VARIABLES LIKE '%buffer_pool%';

   SHOW VARIABLES LIKE '%log_error%';

   SHOW VARIABLES LIKE '%datadir%';

5. **Edit Configuration File**

   sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf

6. **Add/Modify Configuration Parameters**

   [mysqld]

   # Connection and Memory Settings

```
max_connections = 200
default_storage_engine = InnoDB

# InnoDB Settings
innodb_buffer_pool_size = 1G
innodb_log_file_size = 256M
innodb_flush_log_at_trx_commit = 2
innodb_flush_method = O_DIRECT

# Query and Sort Buffer
sort_buffer_size = 2M
bulk_insert_buffer_size = 16M
tmp_table_size = 32M
max_allowed_packet = 256M

# Logging Configuration
slow_query_log = 1
slow_query_log_file = /var/log/mysql/slow-query.log
long_query_time = 2
log_error = /var/log/mysql/error.log
general_log = 0
general_log_file = /var/log/mysql/general.log

# Binary Logging for Replication
server_id = 1
log_bin = /var/log/mysql/mysql-bin
binlog_format = ROW
binlog_expire_logs_seconds = 864000
```

7. **Save and Exit (Ctrl+O, Enter, Ctrl+X)**
8. **Restart MySQL Service**

   sudo systemctl restart mysql
9. **Verify Configuration Changes**

   SHOW VARIABLES LIKE 'max_connections';

   SHOW VARIABLES LIKE 'innodb_buffer_pool_size';

   SHOW VARIABLES LIKE 'slow_query_log';

LAB 2.2: Configuring and Monitoring MySQL Logs

Objective: Enable and monitor different MySQL logs (error, slow query, general).

Step-by-Step Instructions:

1. **Check Error Log**

   sudo tail -f /var/log/mysql/error.log

2. **Create Slow Query Log File (if not exists)**

   sudo touch /var/log/mysql/slow-query.log

   sudo chown mysql:mysql /var/log/mysql/slow-query.log

   sudo chmod 640 /var/log/mysql/slow-query.log

3. **Enable Slow Query Logging**

   SET GLOBAL slow_query_log = 'ON';

   SET GLOBAL long_query_time = 2;

4. **Create Slow Query for Testing**

   SELECT SLEEP(3); -- *This will be logged as slow*

5. **View Slow Query Log**

   sudo tail -20 /var/log/mysql/slow-query.log

6. **Enable General Query Log (Temporary - for debugging only)**

   SET GLOBAL general_log = 'ON';

   SET GLOBAL log_output = 'TABLE'; -- *Log to mysql.general_log table*

7. **Execute Sample Queries**

   SELECT * FROM mysql.user LIMIT 1;

   INSERT INTO test_table VALUES (1, 'test');

   DELETE FROM test_table WHERE id = 1;

8. **View General Query Log**

   SELECT * FROM mysql.general_log ORDER BY event_time DESC LIMIT 10\G

9. **Disable General Query Log (to save resources)**

   SET GLOBAL general_log = 'OFF';

10. **Check Binary Log Status**

    SHOW MASTER STATUS;

    SHOW BINARY LOGS;

Hands-on Tasks:

- Create multiple slow queries and observe them in the log
- Parse slow query log to identify problematic queries
- Set up log rotation using logrotate: sudo vim /etc/logrotate.d/mysql-server
- Monitor log file sizes: du -h /var/log/mysql/

Solution:

To generate slow queries for the lab, first enable the slow query log with a low threshold, then run deliberately slow statements such as SELECT SLEEP() or heavy BENCHMARK() calls and review them in the slow log file or via tools like mysqldumpslow.

1. Enable slow query logging for the lab

Run these as a privileged user (e.g., root) in your lab server.

-- Check if slow query log is enabled

SHOW VARIABLES LIKE 'slow_query_log';

SHOW VARIABLES LIKE 'slow_query_log_file';

SHOW VARIABLES LIKE 'long_query_time';

-- Enable slow query log (session-wide)

SET GLOBAL slow_query_log = 'ON';

*-- Log *all* queries slower than 1 second (for the lab)*

SET GLOBAL long_query_time = 1;

Notes for explanation to learners:

- slow_query_log turns logging on or off.
- long_query_time is the threshold in seconds; anything longer is logged.

On some systems you may need to also set the log file path in my.cnf (persistent):

```
[mysqld]
slow_query_log      = 1
slow_query_log_file = /var/log/mysql/slow.log
long_query_time     = 1
```

2. Create a simple test table (optional)

This lets you demonstrate slow queries that scan or compute on data.

```
CREATE DATABASE IF NOT EXISTS perf_lab;
USE perf_lab;

CREATE TABLE big_table (
  id INT PRIMARY KEY AUTO_INCREMENT,
  pad VARCHAR(200) NOT NULL
) ENGINE = InnoDB;

-- Insert some rows for testing (repeat this block or use a loop to grow the table)
INSERT INTO big_table (pad)
SELECT REPEAT('x', 200)
FROM (
  SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL
  SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL
  SELECT 9 UNION ALL SELECT 10
) t;
```

Run the INSERT several times to reach tens of thousands of rows if needed.

3. Generate slow queries on purpose

a) Using SLEEP to create guaranteed slow queries

These do no work except waiting, but they are excellent for demonstrating logging.

-- Each of these will take about 6–8 seconds and should be logged

SELECT SLEEP(6);

SELECT SLEEP(7);

SELECT SLEEP(8);

SELECT SLEEP(7);

SELECT SLEEP(7);

SLEEP(n) pauses execution for n seconds and will appear in the slow log because it exceeds long_query_time.

b) Using BENCHMARK to burn CPU

This produces slow queries by running a function many times.

-- Heavy CPU test

SELECT BENCHMARK(99999999, MD5('creating a slow query on purpose'));

BENCHMARK(count, expr) executes expr repeatedly and is a standard pattern to simulate CPU-heavy queries for testing.

c) Using an intentionally bad query on a large table

Once big_table is large enough, run:

-- Full table scan with a function on every row

SELECT COUNT(*)

FROM big_table

WHERE MD5(pad) = 'abcd';

Because the function prevents index use and has to run on each row, this can take longer and appear in the slow log on a larger dataset.

4. View slow queries in the log

On Linux, the slow log is often in /var/log/mysql/ or similar; confirm via:

SHOW VARIABLES LIKE 'slow_query_log_file';

Then, from the OS shell:

 # View raw slow log

tail -n 50 /var/log/mysql/slow.log

 # Summarize slow queries (top patterns)

mysqldumpslow -s c -t 10 /var/log/mysql/slow.log

- mysqldumpslow -s c -t 10 groups identical queries and shows the 10 most frequent slow query patterns.

5. Clean up or reset (optional)

For a clean server after the lab:

SET GLOBAL slow_query_log = 'OFF';

-- Optionally reset long_query_time to a higher default, e.g.

SET GLOBAL long_query_time = 10;

This completes the hands-on task: multiple intentionally slow queries are generated, logged, and then inspected via the slow query log and tools.

LAB 2.3: SQL Modes and System Variables

Objective: Understand SQL modes and their impact on query behavior.

Step-by-Step Instructions:

1. **Check Current SQL Mode**

   SELECT @@sql_mode;

   SHOW VARIABLES LIKE 'sql_mode';

2. **Test Strict Mode Behavior**

a) Create a test table:

```
CREATE DATABASE sql_mode_test;

USE sql_mode_test;


CREATE TABLE test_table (
   id INT PRIMARY KEY AUTO_INCREMENT,
   age INT,
   email VARCHAR(50) NOT NULL
);
```

b) Try inserting invalid data:

```
-- This will fail in STRICT mode
INSERT INTO test_table (age, email) VALUES ('not a number', 'test@example.com');
INSERT INTO test_table (id, age) VALUES (1, 25);  -- Missing NOT NULL column
```

3. **Set SQL Mode**

   SET GLOBAL sql_mode='STRICT_TRANS_TABLES,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';

   SET SESSION sql_mode='STRICT_TRANS_TABLES,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';

4. **Observe Different Behaviors**

   -- Test with strict mode enabled

INSERT INTO test_table (age, email) VALUES (25, 'test@example.com');

INSERT INTO test_table (age, email) VALUES ('abc', 'test2@example.com'); -- *Will fail*

5. **Check System Variables Status**

   SHOW STATUS LIKE 'Threads%';

   SHOW STATUS LIKE 'Questions';

   SHOW STATUS LIKE 'Slow_queries';

   SHOW STATUS LIKE 'Connections';

Hands-on Tasks:

- Test INSERT/UPDATE behavior in strict vs non-strict modes
- Identify the differences in error handling
- Document the impact of different SQL_MODE settings
- Create a comparison chart of different SQL modes and their uses

Solution:

Use the same table and DML in two sessions, toggling sql_mode between strict and non-strict so learners can see errors vs silent adjustments or warnings on invalid/missing data.

1. Prepare a test table

Use a clean schema and a table with constraints that are easy to violate.

```
CREATE DATABASE IF NOT EXISTS sqlmode_lab;
USE sqlmode_lab;

DROP TABLE IF EXISTS strict_test;

CREATE TABLE strict_test (
 id    INT UNSIGNED NOT NULL AUTO_INCREMENT,
 qty   INT      NOT NULL,
 price  DECIMAL(5,2) NOT NULL,
```

```
  created_on DATE       NOT NULL,
  PRIMARY KEY (id)
) ENGINE = InnoDB;
```

- INT UNSIGNED lets you show out-of-range negatives.
- NOT NULL columns with no default allow missing-value behavior to differ.
- DECIMAL(5,2) and DATE allow invalid formats/ranges to be tested.

2. Non-strict mode: observe silent adjustments

Turn strict mode off for the current session and show how MySQL “fixes” bad data.

```
-- Session A: non-strict mode
SET SESSION sql_mode = '';

SELECT @@SESSION.sql_mode;

-- 2.1 Missing NOT NULL columns (qty, price, created_on)
INSERT INTO strict_test (qty)
VALUES (10);

-- 2.2 Out-of-range value for UNSIGNED (negative qty)
INSERT INTO strict_test (qty, price, created_on)
VALUES (-5, 100.00, '2024-01-01');

-- 2.3 Invalid date and invalid decimal
INSERT INTO strict_test (qty, price, created_on)
VALUES (1, 9999.999, '2024-02-31');

-- Inspect stored data
SELECT * FROM strict_test;
```

SHOW WARNINGS;

Discuss observations:

- Missing NOT NULL columns get filled with “adjusted” defaults (typically 0 or 0.00 or 0000-00-00) and only warnings are produced.
- Negative UNSIGNED values may be clipped to 0, again with a warning.
- Invalid DATE or out-of-range DECIMAL values are either coerced or stored as 0000-00-00 / truncated values in non-strict mode.

Have learners capture SHOW WARNINGS output after each INSERT.

3. Strict mode: observe hard errors

Now enable strict mode and rerun the same statements to show errors instead of silent fixes.

-- Session B: strict mode enabled

SET SESSION sql_mode = 'STRICT_TRANS_TABLES';

SELECT @@SESSION.sql_mode;

-- 3.1 Missing NOT NULL columns

INSERT INTO strict_test (qty)

VALUES (10);

-- 3.2 Out-of-range UNSIGNED

INSERT INTO strict_test (qty, price, created_on)

VALUES (-5, 100.00, '2024-01-01');

-- 3.3 Invalid date / out-of-range decimal

INSERT INTO strict_test (qty, price, created_on)

VALUES (1, 9999.999, '2024-02-31');

In strict mode:

- Each invalid or missing value causes an error and the statement is rejected.
- No row is inserted or updated for the failing statement (for transactional tables like InnoDB).
- Error messages clearly report which column/value is invalid or missing.

Learners should confirm with:

```
SELECT * FROM strict_test;
```

Rows from step 2 remain, but no new rows from the strict-session attempts appear.

4. UPDATE differences in strict vs non-strict

Use existing rows to demonstrate UPDATE behavior.

```
-- Still in strict mode session
-- Try to set invalid values
UPDATE strict_test
SET qty = -10
WHERE id = 1;

UPDATE strict_test
SET created_on = '2024-13-01'
WHERE id = 1;
```

In strict mode, these UPDATEs fail with errors and the row is not modified.

Switch to non-strict mode and retry:

```
SET SESSION sql_mode = '';

UPDATE strict_test
```

SET qty = -10

WHERE id = 1;

UPDATE strict_test

SET created_on = '2024-13-01'

WHERE id = 1;

SELECT * **FROM** strict_test **WHERE** id = 1;

SHOW WARNINGS;

Non-strict mode typically:

- Adjusts out-of-range values (e.g., negative UNSIGNED to 0).
- Stores invalid dates as 0000-00-00 if date-related modes allow it.
- Only emits warnings while modifying the row.

5. Optional: TRADITIONAL mode and INSERT IGNORE

-- Traditional mode (very strict)

SET SESSION sql_mode = 'TRADITIONAL';

INSERT INTO strict_test (qty, price, created_on)

VALUES (-1, 100.00, '2024-01-01'); *-- should error*

-- Same statement with IGNORE: executes but adjusts value

INSERT IGNORE INTO strict_test (qty, price, created_on)

VALUES (-1, 100.00, '2024-01-01');

SELECT * **FROM** strict_test **ORDER BY** id **DESC LIMIT** 3;

SHOW WARNINGS;