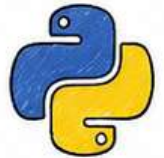


Chapter 32: MySQL Connectivity

32.1 MySQL with Python

- * MySQL is a popular open-source **Relational Database Management System** (RDBMS).
- * Python can be used to **connect** to a **MySQL database** and perform various database operations.
- * This allows Python applications to **store, retrieve, update** and **delete** data in a MySQL database.
- * Python acts as the front-end and MySQL acts as the back-end database.



Key Points :

- ✓ MySQL stores data in tables (rows and columns).
- ✓ Python is used to connect and interact with MySQL.
- ✓ MySQL with Python is widely used in web applications, data analysis and enterprise systems.



Working :

Python Application
(Front-End)



MySQL Connector
(Communication Layer)



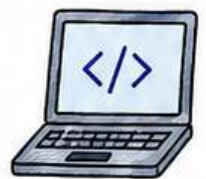
MySQL Database
(Back-End)



Python sends SQL queries using Connector → MySQL processes the queries → Results are returned to Python.

32.2 MySQL Connector/Python

- * MySQL Connector/Python is the **official MySQL driver** for Python.
- * It allows Python programs to **connect** to **MySQL Server**.
- * It supports **Python 3.x** and provides a full set of features to interact with MySQL.



Features :

- * Easy to install and use.
- * Supports prepared statements.
- * Provides connection pooling.
- * Supports SSL connections.
- * High performance and reliable.



Basic Syntax to Connect :

```
import mysql.connector
conn = mysql.connector.connect(
    host = "localhost",
    user = "root",
    password = "your_password",
    database = "your_database"
)
```

Make sure
MySQL Server
is running
before
connecting.

Key Takeaway :



MySQL Connector/Python makes it **easy** for Python applications to connect to MySQL and perform database operations **efficiently**.



32.3 Connection Management

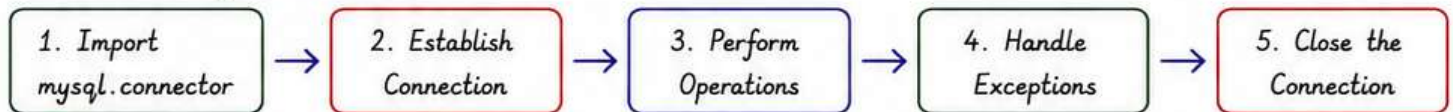
- * A connection must be **established** before interacting with the MySQL database.
- * Always **close the connection** after the work is done to free resources.
- * Use **try...except** blocks to handle connection errors.
- * Connection parameters include **host**, **user**, **password** and **database**.

Key Points :

- ✓ Open a connection before performing any database operation.
- ✓ Close the connection when the task is completed.
- ✓ Use error handling to manage connection related issues.



Steps to Manage Connection :



Example :

```
import mysql.connector
try:
    conn = mysql.connector.connect(
        host = "localhost",
        user = "root",
        password = "your_password",
        database = "your_database")
    print("Connection Successful!")
except mysql.connector.Error as err:
    print("Error:", err)
finally:
    if 'conn' in locals() and conn.is_connected():
        conn.close()
    print("Connection Closed.")
```

Connection Parameters :

- **host** - Database server address (default: localhost)
- **user** - MySQL username
- **password** - MySQL password
- **database** - Database name to connect
- **port** - Port number (default: 3306)



Using **try...except** ensures that your program does not crash if connection fails.

32.4 Executing Queries from Python

- * Once the connection is established, we can execute SQL queries using a **cursor object**.
- * Queries can be **SELECT, INSERT, UPDATE, DELETE**, etc.
- * After executing queries that modify data, use **commit()** to save changes.



Common Methods :

Method	Description
<code>cursor()</code>	Creates a cursor object to execute queries.
<code>execute(query)</code>	Executes a single SQL query.
<code>executemany(query, values)</code>	Executes a query for multiple sets of values.
<code>fetchone()</code>	Fetches the next row of a query result.
<code>fetchall()</code>	Fetches all rows of a query result.
<code>commit()</code>	Saves the changes made to the database.
<code>close()</code>	Closes the cursor or connection.

Example :

```
import mysql.connector
conn = mysql.connector.connect(
    host="localhost", user="root",
    password="your_password",
    database="your_database")
cur = conn.cursor()
cur.execute("SELECT * FROM students;")
rows = cur.fetchall()
for row in rows:
    print(row)
cur.close()
conn.close()
```



Key Takeaway :



Proper **connection management** and correct **execution of queries** are essential for interacting with MySQL databases **efficiently** from Python.



32.5 Parameterized Queries

- * Parameterized queries help to **prevent SQL injection attacks**.
- * Instead of directly inserting values into the query, **placeholders** are used.
- * Values are passed as parameters to the **execute()** method.
- * This makes the code safer and more efficient.

Example :

```
import mysql.connector
conn = mysql.connector.connect(
    host = "localhost", user = "root",
    password = "your_password", database = "your_database")
cur = conn.cursor()
# Using parameterized query
sql = "INSERT INTO students(name, age, course) VALUES(%s, %s, %s)"
values = ("Bhavitha", 22, "CSE")
cur.execute(sql, values)
conn.commit()
cur.close()
conn.close()
```



Key Points :

- ✓ Use %s as placeholders.
- ✓ Values are passed as a tuple.
- ✓ Prevents SQL injection.
- ✓ Improves performance.



Never use string concatenation to build SQL queries.
Always use parameterized queries.

32.6 CRUD using Python

- * CRUD stands for **Create, Read, Update and Delete** operations.



Examples :

Operation	SQL Query	Python Example (using execute())	Description	
Create (INSERT)	INSERT INTO students (name, age, course) VALUES(%s, %s, %s)	cur.execute("INSERT INTO students (name, age, course) VALUES(%s, %s, %s)", ("Bhavitha", 22, "CSE")) conn.commit()	Inserts a new record into the table.	
Read (SELECT)	SELECT * FROM students WHERE course = %s	cur.execute("SELECT * FROM students WHERE course = %s", ("CSE",)) rows = cur.fetchall() for row in rows: print(row)	Retrieves records from the table.	
Update (UPDATE)	UPDATE students SET age = %s WHERE id = %s	cur.execute("UPDATE students SET age = %s WHERE id = %s", (23, 1)) conn.commit()	Updates existing records.	
Delete (DELETE)	DELETE FROM students WHERE id = %s	cur.execute("DELETE FROM students WHERE id = %s", (1,)) conn.commit()	Deletes records from the table.	

Key Takeaway :

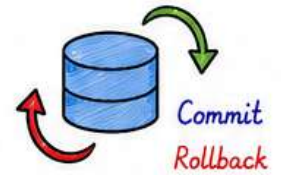


Using MySQL with Python, we can perform all **CRUD** operations efficiently.
Always use **parameterized queries** and proper connection management for **secure**
and **reliable** database interaction.



32.7 Transactions using Python

- * A transaction is a set of SQL operations performed as a **single unit of work**.
- * Transactions ensure data **consistency** and **reliability**.
- * Python MySQL Connector supports transactions using **commit()** and **rollback()**.
- * If an error occurs, **rollback()** undoes all changes.



Example :

```
import mysql.connector
conn = mysql.connector.connect(
    host = "localhost", user = "root",
    password = "your_password", database = "your_database")
try:
    cur = conn.cursor()
    # Start transaction
    cur.execute("UPDATE accounts SET balance = balance - %s
                WHERE id = %s", (500, 1))
    cur.execute("UPDATE accounts SET balance = balance + %s
                WHERE id = %s", (500, 2))
    # If both queries succeed, commit the transaction
    conn.commit()
    print("Transaction committed successfully.")
except mysql.connector.Error as err:
    conn.rollback()
    print("Error occurred. Transaction rolled back.")
finally:
    cur.close()
    conn.close()
```

Key Points :

- ✓ Use **commit()** to save all changes.
- ✓ Use **rollback()** to undo changes in case of error.
- ✓ Transactions maintain data integrity.
- ✓ Always close the connection after the transaction.



Use transactions when performing multiple operations that depend on each other. Either **all** succeed or **none** should be applied.

32.8 MySQL with Java

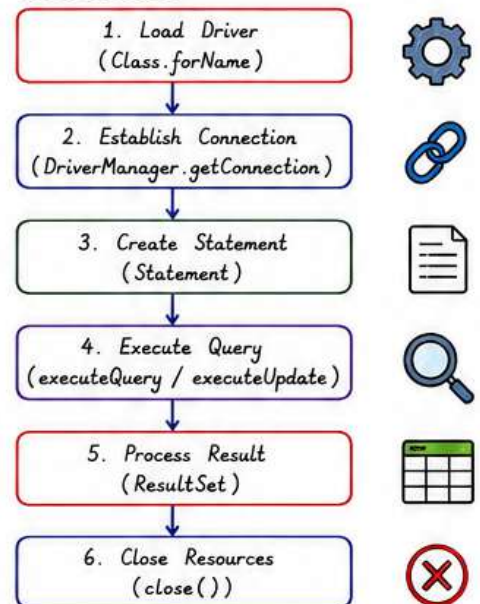
- * Java can connect to MySQL using **JDBC (Java Database Connectivity)**.
- * JDBC provides drivers and APIs to interact with databases.
- * We use **MySQL Connector/J** (JDBC Driver) to connect Java with MySQL.
- * Steps: Load Driver → Establish Connection → Create Statement → Execute Query → Process Result.



Example (Java) :

```
import java.sql.*;
public class MySQLExample {
    public static void main(String[] args) {
        try {
            // 1. Load Driver
            Class.forName("com.mysql.cj.jdbc.Driver");
            // 2. Establish Connection
            Connection conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/your_database", "root", "your_password");
            // 3. Create Statement
            Statement stmt = conn.createStatement();
            // 4. Execute Query
            ResultSet rs = stmt.executeQuery("SELECT * FROM students");
            // 5. Process Result
            while(rs.next()) {
                System.out.println(rs.getInt("id") + " " +
                    rs.getString("name") + " " + rs.getInt("age"));
            }
            // Close resources
            rs.close(); stmt.close(); conn.close();
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

JDBC Steps :



Key Takeaway :

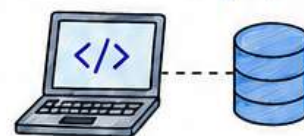


Python provides **simple** and **powerful** ways to connect and manage MySQL databases. Java uses **JDBC** to connect with MySQL and perform database operations efficiently.



32.9 JDBC Connectivity

- * JDBC (**Java Database Connectivity**) is an API to connect and execute queries with databases.
- * JDBC works with **any database** that supports JDBC drivers (like MySQL).
- * For MySQL, we use **MySQL Connector/J** driver.
- * Connection is established using **DriverManager.getConnection()**.



JDBC URL for MySQL :

```
jdbc:mysql://host:port/database_name
```

Example :

```
jdbc:mysql://localhost:3306/college
```

Example (JDBC Connection) :

```
import java.sql.*;
public class JDBCTest {
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // 1. Load Driver
            Class.forName("com.mysql.cj.jdbc.Driver");
            // 2. Establish Connection
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/college", "root", "your_password");
            // 3. Create Statement
            stmt = conn.createStatement();
            // 4. Execute Query
            rs = stmt.executeQuery("SELECT * FROM students");
            // 5. Process Result
            while(rs.next()) {
                System.out.println(rs.getInt("id") + " " +
                    rs.getString("name") + " " + rs.getInt("age"));
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            try {
                if(rs != null) rs.close();
                if(stmt != null) stmt.close();
                if(conn != null) conn.close();
            } catch (Exception e) { }
        }
    }
}
```

Common JDBC Classes :

Class / Interface	Description
DriverManager	Manages the list of database drivers.
Connection	Represents a connection to the database.
Statement	Used to execute static SQL statements.
PreparedStatement	Used to execute parameterized queries.
ResultSet	Stores the result of a query.

Key Takeaway :



JDBC provides a **standard** way for Java applications to connect to MySQL.

Always load the driver, establish connection, execute queries and **close resources**.



32.10 MySQL with PHP

- * PHP can connect to MySQL using **MySQLi** (Improved extension) or **PDO** (PHP Data Objects).
- * MySQLi provides **object-oriented** and **procedural** interfaces.
- * It supports prepared statements, transactions and multiple queries.
- * Steps: Open Connection → Execute Query → Process Result → Close Connection.



Example (MySQLi in PHP) :

```
<?php
// 1. Connect to MySQL
$conn = new mysqli("localhost", "root", "your_password", "college");

// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

// 2. Execute Query
$sql = "SELECT id, name, age FROM students";
$result = $conn->query($sql);

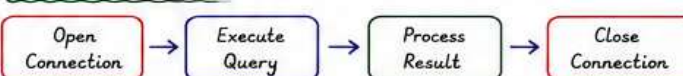
// 3. Process Result
if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        echo $row["id"] . " " . $row["name"] . " " . $row["age"];
        echo "<br>";
    }
} else {
    echo "No records found";
}

// 4. Close Connection
$result->free();
$conn->close();
?>
```

MySQLi Functions :

Function	Description
new mysqli()	Creates a new connection.
query(\$sql)	Executes a SQL query.
num_rows	Returns number of rows in result.
fetch_assoc()	Fetches a row as an associative array.
close()	Closes the connection.

PHP-MySQLi Workflow :



Key Points :

- ✓ Use MySQLi or PDO to connect PHP with MySQL.
- ✓ Always check connection before executing queries.
- ✓ Use prepared statements to prevent SQL injection.
- ✓ Close the connection to free resources.



Key Takeaway :



PHP with MySQL makes web applications **dynamic** and **data-driven**.

MySQLi provides **security**, **performance** and **ease of use** for database operations in PHP.

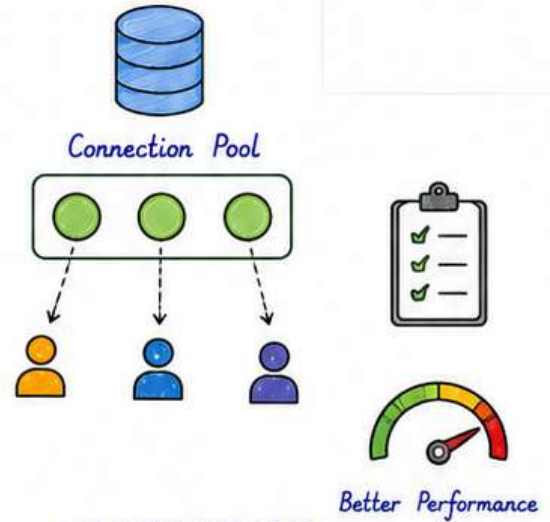


32.11 Connection Pooling

- * Creating a new connection to the database is **expensive**.
- * Connection pooling **reuses existing connections** to improve performance.
- * A **pool of database connections** is created and managed.
- * When a connection is needed, one is **taken** from the pool.
- * After use, the connection is **returned** to the pool instead of being closed.

Key Points :

- ✓ Improves application performance.
- ✓ Reduces the overhead of creating and closing connections.
- ✓ Manages a fixed number of connections efficiently.
- ✓ Helps in handling high concurrency.
- ✓ Widely used in web applications and enterprise systems.



How it Works :

1. Create a pool of connections
2. Get connection from the pool
3. Use connection for operations
4. Return connection to the pool
5. Reuse connection for future requests

Example (Python - mysql.connector.pooling):

```
import mysql.connector
from mysql.connector import pooling

# Create a connection pool
mydbpool = pooling.MySQLConnectionPool(
    pool_name = "mypool",
    pool_size = 5,
    host = "localhost",
    user = "root",
    password = "your_password",
    database = "college"
)

# Get connection from pool
conn = mydbpool.get_connection()
cur = conn.cursor()
cur.execute("SELECT * FROM students")
rows = cur.fetchall()
for row in rows:
    print(row)

# Return connection to pool
cur.close()
conn.close() # returns connection to the pool
```

Common Pooling Parameters :

Parameter	Description
pool_name	Name of the connection pool.
pool_size	Maximum number of connections in the pool.
pool_reset_session	Resets session when connection is returned to pool (default: True).
pool_pre_ping	Checks connection liveness before using (default: False).

Benefits of Connection Pooling :

- * Reduces connection creation time.
- * Increases application performance.
- * Efficiently handles high loads.
- * Optimizes database resources.



Where it is Used :

- Web applications
- Enterprise applications
- Microservices
- High-traffic systems
- Any application with frequent database access



Key Takeaway :



Connection pooling is an effective technique to **reuse** database connections, improve **performance**, handle **more users** and **reduce resource consumption**.

