**Database Management Assignment: Sakila Database in MySQL Workbench**

# Objective

This assignment aims to enhance your understanding of database management by performing various operations on the **Sakila** database using **MySQL Workbench**. You will modify, insert, delete, and update records, design complex queries, work with transactions, and ensure data integrity and consistency.

# Assignment Tasks

## 1. Identifying Tools and Statements for Modifying Database Content

- Research and document different **SQL statements** used to modify database content (INSERT, UPDATE, DELETE, ALTER, etc.).

### 1. INSERT Statement

- **Purpose**: Adds new records to a table.
- **Syntax**:
  INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);
- **Example**:
  INSERT INTO employees (id, name, position) VALUES (1, 'Alice', 'Developer');

### 2. UPDATE Statement

- **Purpose**: Modifies existing records in a table.
- **Syntax**:
  UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition;
- **Example**:
  UPDATE employees SET position = 'Senior Developer' WHERE id = 1;

### 3. DELETE Statement

- **Purpose**: Removes records from a table.
- **Syntax**:
  DELETE FROM table_name WHERE condition;
- **Example**:
  DELETE FROM employees WHERE id = 1;

### 4. ALTER Statement

- **Purpose**: Modifies the structure of a table (e.g., adding or removing columns).
- **Syntax**:
  ALTER TABLE table_name ADD COLUMN column_name datatype;

- **Example**:
  ALTER TABLE employees ADD COLUMN salary DECIMAL(10,2);

● Describe the functionalities of MySQL Workbench tools such as **SQL Editor, Schema Inspector, and Query Builder**:

### 1. SQL Editor

- A graphical interface for writing and executing SQL queries.
- Features include syntax highlighting, query history, and error detection.

### 2. Schema Inspector

- Provides detailed information about database schemas, including table structures, indexes, and constraints.
- Allows users to analyze table sizes, column types, and optimize performance.

### 3. Query Builder

- A visual tool for creating SQL queries without writing code manually.
- Users can drag and drop tables, define relationships, and generate SQL commands automatically.

● **Deliverable:** A section in the final report summarizing SQL statements and MySQL Workbench tools

This section provides an overview of essential SQL statements for modifying database content and explores key tools in MySQL Workbench. Understanding these elements is crucial for efficient database management and manipulation.

## 2. Data Insertion, Deletion, and Update

● Using the **actor** table, insert a new record with fictitious data.

```
mysql> insert into actor (actor_id, first_name, last_name, last_update)
    -> values (201, 'Mati','Peluso', now());
Query OK, 1 row affected (0.01 sec)
```

```
|      200 | THORA       | TEMPLE       | 2006-02-15 04:34:33 |
|      201 | Mati        | Peluso       | 2025-03-10 13:26:45 |
+----------+-------------+--------------+---------------------+
201 rows in set (0.00 sec)
```

- Update an existing record by changing the last name of an actor.

```
mysql> update actor set first_name='Nati' where actor_id=201;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
|      201 | Nati        | Peluso       | 2025-03-10 13:33:02 |
+----------+-------------+--------------+---------------------+
201 rows in set (0.00 sec)
```

- Delete an actor from the table.

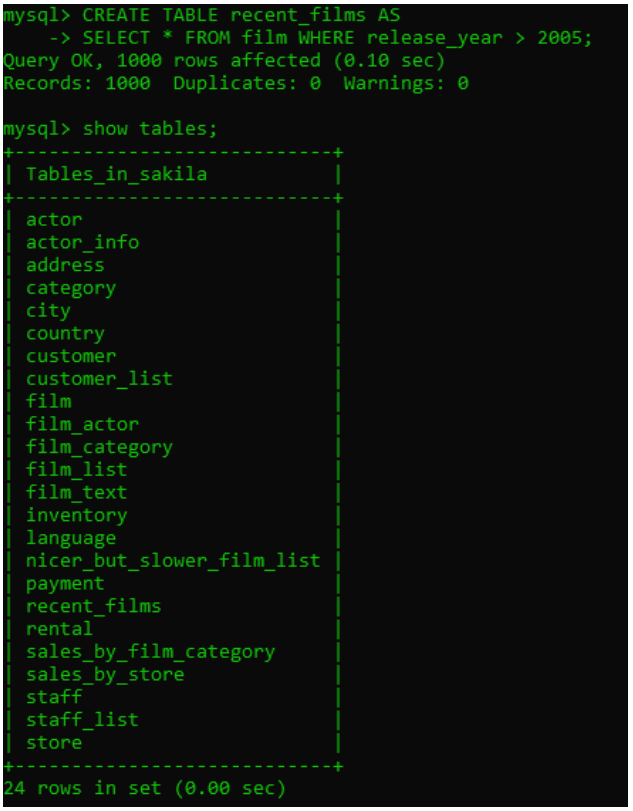
```
mysql> delete from actor where actor_id=201;
Query OK, 1 row affected (0.01 sec)
```

```
|      200 | THORA       | TEMPLE       | 2006-02-15 04:34:33 |
+----------+-------------+--------------+---------------------+
200 rows in set (0.00 sec)
```

- **Deliverable:** Provide the executed SQL statements and their results.
  - **SQL File:** `02_modify_actor.sql`
  - **Screenshots:** `02_modify_actor_screenshots/`

## 3. Creating a Table from a Query Result

- Execute a query to retrieve all movies released after 2005 from the **film** table.



```
mysql> CREATE TABLE recent_films AS
    -> SELECT * FROM film WHERE release_year > 2005;
Query OK, 1000 rows affected (0.10 sec)
Records: 1000  Duplicates: 0  Warnings: 0

mysql> show tables;
+---------------------------+
| Tables_in_sakila          |
+---------------------------+
| actor                     |
| actor_info                |
| address                   |
| category                  |
| city                      |
| country                   |
| customer                  |
| customer_list             |
| film                      |
| film_actor                |
| film_category             |
| film_list                 |
| film_text                 |
| inventory                 |
| language                  |
| nicer_but_slower_film_list |
| payment                   |
| recent_films              |
| rental                    |
| sales_by_film_category    |
| sales_by_store            |
| staff                     |
| staff_list                |
| store                     |
+---------------------------+
24 rows in set (0.00 sec)
```

- Store the result in a new table called **recent_films**.

```
mysql> select * from recent_films;
+---------+----------------------------+--------------------------------------------------------------------------------
--------------------------------------------------------------+--------------+-------------+----------------------+--------
---------+-------------+--------+-----------------+--------+-----------------------------------------------------------
+---------------------+
| film_id | title                      | description
                                                | release_year | language_id | original_language_id | rental_
duration | rental_rate | length | replacement_cost | rating | special_features
| last_update         |
+---------+----------------------------+--------------------------------------------------------------------------------
--------------------------------------------------------------+--------------+-------------+----------------------+--------
---------+-------------+--------+-----------------+--------+-----------------------------------------------------------
+---------------------+
|       1 | ACADEMY DINOSAUR           | A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in
 The Canadian Rockies                                       |         2006 |           1 |                 NULL |
       6 |        0.99 |     86 |            20.99 | PG     | Deleted Scenes,Behind the Scenes
| 2006-02-15 05:03:42 |
|       2 | ACE GOLDFINGER             | A Astounding Epistle of a Database Administrator And a Explorer who must Fi
nd a Car in Ancient China                                   |         2006 |           1 |                 NULL |
       3 |        4.99 |     48 |            12.99 | G      | Trailers,Deleted Scenes
| 2006-02-15 05:03:42 |
|       3 | ADAPTATION HOLES           | A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberjac
k in A Baloon Factory                                       |         2006 |           1 |                 NULL |
       7 |        2.99 |     50 |            18.99 | NC-17  | Trailers,Deleted Scenes
| 2006-02-15 05:03:42 |
|       4 | AFFAIR PREJUDICE           | A Fanciful Documentary of a Frisbee And a Lumberjack who must Chase a Monke
y in A Shark Tank                                           |         2006 |           1 |                 NULL |
       5 |        2.99 |    117 |            26.99 | G      | Commentaries,Behind the Scenes
| 2006-02-15 05:03:42 |
|       5 | AFRICAN EGG                | A Fast-Paced Documentary of a Pastry Chef And a Dentist who must Pursue a F
orensic Psychologist in The Gulf of Mexico                  |         2006 |           1 |                 NULL |
       6 |        2.99 |    130 |            22.99 | G      | Deleted Scenes
| 2006-02-15 05:03:42 |
|       6 | AGENT TRUMAN               | A Intrepid Panorama of a Robot And a Boy who must Escape a Sumo Wrestler in
 Ancient China                                              |         2006 |           1 |                 NULL |
       3 |        2.99 |    169 |            17.99 | PG     | Deleted Scenes
| 2006-02-15 05:03:42 |
|       7 | AIRPLANE SIERRA            | A Touching Saga of a Hunter And a Butler who must Discover a Butler in A Je
t Boat                                                      |         2006 |           1 |                 NULL |
       6 |        4.99 |     62 |            28.99 | PG-13  | Trailers,Deleted Scenes
| 2006-02-15 05:03:42 |
|       8 | AIRPORT POLLOCK            | A Epic Tale of a Moose And a Girl who must Confront a Monkey in Ancient Ind
ia                                                          |         2006 |           1 |                 NULL |
       6 |        4.99 |     54 |            15.99 | R      | Trailers
| 2006-02-15 05:03:42 |
|       9 | ALABAMA DEVIL              | A Thoughtful Panorama of a Database Administrator And a Mad Scientist who m
ust Outgun a Mad Scientist in A Jet Boat                    |         2006 |           1 |                 NULL |
       3 |        2.99 |    114 |            21.99 | PG-13  | Trailers,Deleted Scenes
```

- **Deliverable:** The SQL script used and a screenshot of the newly created table.
    - **SQL File:** `03_create_recent_films.sql`
    - **Screenshot:** `03_recent_films_screenshot.png`

## 4. Designing Complex SQL Scripts

- Write an SQL script that:
    - Lists all customers who have rented a film in the last 30 days.

```
mysql> SELECT DISTINCT customer.customer_id, customer.first_name, customer.last_name
    -> FROM customer
    -> JOIN rental ON customer.customer_id = rental.customer_id
    -> WHERE rental_date >= CURDATE() - INTERVAL 30 DAY;
Empty set (0.02 sec)
```

    - Identifies the most rented film in the database.

```
mysql> SELECT film.film_id, film.title, COUNT(rental.rental_id) AS rental_count
    -> FROM film
    -> JOIN inventory ON film.film_id = inventory.film_id
    -> JOIN rental ON inventory.inventory_id = rental.inventory_id
    -> GROUP BY film.film_id, film.title
    -> ORDER BY rental_count DESC
    -> LIMIT 1;
+---------+--------------------+--------------+
| film_id | title              | rental_count |
+---------+--------------------+--------------+
|     103 | BUCKET BROTHERHOOD |           34 |
+---------+--------------------+--------------+
1 row in set (0.05 sec)
```

- Displays the total revenue generated per store.

```
mysql> SELECT store.store_id, SUM(payment.amount) AS total_revenue
    -> FROM store
    -> JOIN staff ON store.store_id = staff.store_id
    -> JOIN payment ON staff.staff_id = payment.staff_id
    -> GROUP BY store.store_id;
+----------+---------------+
| store_id | total_revenue |
+----------+---------------+
|        1 |      33482.50 |
|        2 |      33924.06 |
+----------+---------------+
2 rows in set (0.07 sec)
```

- **Deliverable:** The SQL script with comments explaining each query.
  - **SQL File:** `04_complex_queries.sql`

## 5. Understanding Transactions

- Explain transactions and their importance in database management.
- Using MySQL Workbench, perform a transaction that:
  - Inserts a new rental record.
  - Updates the inventory to reflect the rental.
  - Commits the transaction.

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO rental (rental_date, inventory_id, customer_id, return_date, staff_id, last_update)VALUES (NOW(), 5, 1, NULL, 2, NOW());
Query OK, 1 row affected (0.01 sec)

mysql> UPDATE inventory SET last_update = NOW() WHERE inventory_id = 5;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)
```

- **Deliverable:**
  - **SQL File:** `05_transaction_example.sql`
  - **Section in the final report covering transactions.**

## 6. Rolling Back Transactions

- Demonstrate a scenario where a transaction is partially executed but later rolled back

due to an error (e.g., an attempt to rent a movie that is out of stock).

```
mysql> -- Verificar si el inventory_id existe y está disponible
mysql> SELECT COUNT(*) FROM inventory WHERE inventory_id = 100;
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO rental (rental_date, inventory_id, customer_id, return_date, staff_id, last_update)
    -> VALUES (NOW(), 100, 1, NULL, 2, NOW());
Query OK, 1 row affected (0.00 sec)

mysql> SELECT COUNT(*) INTO @inventory_exists FROM inventory WHERE inventory_id = 100;
Query OK, 1 row affected (0.00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.01 sec)
```

- **Deliverable:**
  - **SQL File:** `06_rollback_example.sql`
  - **Section in the final report covering rollback transactions.**

## 7. Understanding Record Locking Policies

- Research and document different types of record-locking mechanisms (pessimistic vs. optimistic locking).

Record locking is a mechanism used in database management systems to ensure data consistency when multiple transactions try to access the same data simultaneously. MySQL supports two main types of record locking:

**a) Pessimistic Locking**

- Assumes conflicts will occur and prevents them by locking records preemptively.
- A transaction locks a record, preventing other transactions from modifying it until the lock is released.
- Used in high-concurrency environments where data integrity is critical.
-

**b) Optimistic Locking**

- Assumes conflicts are rare and allows multiple transactions to read data.
- Before committing, a transaction checks whether the data has changed.
- Commonly used with a versioning system or timestamp.

- Test a scenario where two users attempt to update the same record simultaneously.

**User 1:**

START TRANSACTION;
UPDATE inventory SET stock = stock - 1 WHERE inventory_id = 100;
-- The row is now locked until you execute COMMIT or ROLLBACK.

**User 2:**

START TRANSACTION;
UPDATE inventory SET stock = stock - 1 WHERE inventory_id = 100;
-- This query will remain blocked until User 1 executes COMMIT or ROLLBACK.

**User 1:**

COMMIT; -- Locks are released.

-- Now Session 2 will be able to execute its UPDATE without issues.

- **Deliverable:**
  - **Section in the final report covering record locking policies.**

| | |
|---|---|
| **Pessimistic** | High contention, ensuring data integrity is crucial (e.g., financial transactions). |
| **Optimistic** | Low contention, allows better performance (e.g., user profile updates). |

## 8. Ensuring Data Integrity and Consistency

- Identify potential data integrity issues in the **Sakila** database.

1. Check for Orphaned Records

This query ensures there are no rental records without a matching customer or inventory entry:

```
mysql> SELECT r.rental_id
    -> FROM rental r
    -> LEFT JOIN customer c ON r.customer_id = c.customer_id
    -> LEFT JOIN inventory i ON r.inventory_id = i.inventory_id
    -> WHERE c.customer_id IS NULL OR i.inventory_id IS NULL;
Empty set (0.12 sec)
```

2. Verify Foreign Key Constraints

We check if foreign key constraints are properly set in tables like rental and payment:

```
mysql> SELECT table_name, constraint_name, referenced_table_name
    -> FROM information_schema.key_column_usage
    -> WHERE table_schema = 'sakila' AND referenced_table_name IS NOT NULL;
+---------------+--------------------------+-----------------------+
| TABLE_NAME    | CONSTRAINT_NAME          | REFERENCED_TABLE_NAME |
+---------------+--------------------------+-----------------------+
| address       | fk_address_city          | city                  |
| city          | fk_city_country          | country               |
| customer      | fk_customer_address      | address               |
| customer      | fk_customer_store        | store                 |
| film          | fk_film_language         | language              |
| film          | fk_film_language_original | language             |
| film_actor    | fk_film_actor_actor      | actor                 |
| film_actor    | fk_film_actor_film       | film                  |
| film_category | fk_film_category_category | category             |
| film_category | fk_film_category_film    | film                  |
| inventory     | fk_inventory_film        | film                  |
| inventory     | fk_inventory_store       | store                 |
| payment       | fk_payment_customer      | customer              |
| payment       | fk_payment_rental        | rental                |
| payment       | fk_payment_staff         | staff                 |
| rental        | fk_rental_customer       | customer              |
| rental        | fk_rental_inventory      | inventory             |
| rental        | fk_rental_staff          | staff                 |
| staff         | fk_staff_address         | address               |
| staff         | fk_staff_store           | store                 |
| store         | fk_store_address         | address               |
| store         | fk_store_staff           | staff                 |
+---------------+--------------------------+-----------------------+
22 rows in set (0.01 sec)
```

3. Check for Duplicate Records

This ensures there are no duplicate customers with the same name and email:

```
mysql> SELECT first_name, last_name, email, COUNT(*)
    -> FROM customer
    -> GROUP BY first_name, last_name, email
    -> HAVING COUNT(*) > 1;
Empty set (0.01 sec)
```

4. Verify Data Format Consistency

For example, checking if all rental dates are valid (not in the future):
● Implement foreign key constraints and triggers to maintain data consistency.

- **Create a trigger** that checks if the rental date is in the future before inserting any rental record.

This trigger will fire before any new record is inserted into the rental table, and it will check if the rental_date is in the future. If it is, the trigger will prevent the insertion and raise an error.

```
mysql> DELIMITER $$
mysql>
mysql> CREATE TRIGGER validate_rental_date BEFORE INSERT ON rental
    -> FOR EACH ROW
    -> BEGIN
    ->   -- Check if rental date is in the future
    ->   IF NEW.rental_date > CURDATE() THEN
    ->     -- Raise an error if rental date is in the future
    ->     SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Rental date cannot be in the future';
    ->   END IF;
    -> END $$
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> DELIMITER ;
```

- **Implement Foreign Key Constraints**: Foreign key constraints are used to ensure that every rental record links to valid records in the inventory and customer tables.

This constraint ensures that each rental must reference a valid inventory record (like a valid film).

```
mysql> ALTER TABLE rental
    -> ADD CONSTRAINT fk_inventory FOREIGN KEY (inventory_id) REFERENCES inventory (inventory_id)
    -> ON DELETE CASCADE;
Query OK, 16049 rows affected (0.65 sec)
Records: 16049  Duplicates: 0  Warnings: 0
```

This constraint ensures that each rental must reference a valid customer record.

```
mysql> ALTER TABLE rental
    -> ADD CONSTRAINT fk_customer FOREIGN KEY (customer_id) REFERENCES customer (customer_id)
    -> ON DELETE CASCADE;
Query OK, 16049 rows affected (0.50 sec)
Records: 16049  Duplicates: 0  Warnings: 0
```

- After implementing the constraints and trigger, test the system by trying to insert records into the rental table. By inserting a rental with a future date we verify that the trigger prevents the insertion.

```
mysql> INSERT INTO rental (rental_date, inventory_id, customer_id)
    -> VALUES ('2025-03-12', 1, 1);
ERROR 1644 (45000): Rental date cannot be in the future
```

- **Deliverable:**
  - **SQL File:** `08_data_integrity.sql`
  - **Section in the final report covering data integrity.**

# Submission Requirements

- **Folder Structure:**
  - `Deliverables/`
    - `SQL_Scripts/` (Contains all `.sql` files)
    - `Screenshots/` (Contains screenshots of query executions, if

applicable)
        - `Final_Report/` (Contains a single consolidated report in PDF format)
- **Final Report:** All previously separate reports should now be consolidated into a single PDF file:
    - **Filename:** `Surname_Name_Final_Report_Sakila.pdf`
    - **Folder:** `Deliverables/Final_Report/`
    - The report must contain all research, explanations, query results, and screenshots.

## Notes:

- Ensure that all SQL statements are tested before submission.
- The final report should include reflections on challenges faced and how they were overcome.
- Cite any external references used in your research.