Revision of SQL for MySQL RDBMS

1. SQL Query Structures

Understanding SQL query structures, including joins and subqueries, is essential for effective data retrieval and manipulation. This section provides intermediate queries with explanations, covering various SQL constructs used in Data Query Language (DQL).

1.1. Basic SELECT Statement

Query:

```sql
SELECT first_name, last_name
FROM actor;
```

Explanation:

- **Purpose:** Retrieves the first and last names of all actors from the `actor` table.
- **Output:** A list of actors with their respective first and last names.

1.2. WHERE Clause

Query:

```sql
SELECT first_name, last_name
FROM actor
WHERE last_name = 'KILMER';
```

Explanation:

- **Purpose:** Retrieves actors whose last name is ‘Smith’.
- **Output:** A subset of actors with the last name ‘Smith’.

1.3. ORDER BY Clause

Query:

```sql
SELECT first_name, last_name
FROM actor
ORDER BY last_name ASC, first_name DESC;
```

Explanation:

- **Purpose:** Retrieves all actors ordered by last name in ascending order and first name in descending order.
- **Output:** Sorted list of actors based on specified criteria.

1.4. JOIN Operations

Joins combine rows from two or more tables based on related columns.

1.4.1. INNER JOIN

Query:

```
SELECT a.first_name, a.last_name, f.title
FROM actor a
INNER JOIN film_actor fa ON a.actor_id = fa.actor_id
INNER JOIN film f ON fa.film_id = f.film_id;
```

Explanation:

- **Purpose:** Retrieves actors and the titles of films they have acted in.
- **Output:** Rows containing actor names and corresponding film titles where there is a matching record in both `actor` and `film_actor` tables.

1.4.2. LEFT JOIN

Query:

```
SELECT
    f.film_id,
    f.title,
    r.rental_id
FROM
    film f
LEFT JOIN
    inventory i ON f.film_id = i.film_id
LEFT JOIN
    rental r ON i.inventory_id = r.inventory_id
WHERE
    r.rental_id IS NULL;
```

Explanation:

- **Purpose:** Retrieves all films, including those that have never been rented. For films without any rentals, the `rental_id` column will have `NULL` .
- **Output:** A list of films and their rental IDs. Films with no rentals will show `NULL` in the `rental_id` column.

1.4.3. RIGHT JOIN

Query:

```
SELECT
    i.inventory_id,
    f.film_id,
    f.title
FROM
    inventory i
RIGHT JOIN
    film f ON i.film_id = f.film_id
WHERE
    i.inventory_id IS NULL;
```

Explanation:

- **Purpose:** Retrieves all films, including those without any associated inventory items. For films without inventory, the `inventory_id` column will have `NULL` .
- **Output:** A list of films and their inventory IDs. Films without inventory will show `NULL` in the `inventory_id` column.

1.4.4. CROSS JOIN

A **CROSS JOIN** often results in a Cartesian product, which can be overwhelming if used carelessly. However, it makes sense when combining data sets where all possible combinations are relevant for analysis. A practical use case in the **Sakila** database could be generating all possible **film rental recommendations** for customers.

Query

```
SELECT
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    f.title AS recommended_film
FROM
    customer c
CROSS JOIN
    film f
LIMIT 20;
```

Explanation

- **Purpose:** Generates a list of all possible film recommendations for each customer.
- **CROSS JOIN:** Creates a Cartesian product of all customers and films.
- **LIMIT 20:** Limits the result set to make it manageable for review.

Practical Use Case:

Can be used for:

- Recommending all available films to customers.
- Generating a matrix of all customer-film combinations for targeted marketing.
- **CROSS JOIN Usage:**
  This makes sense when analyzing or simulating all possible relationships between two datasets, like customers and films.

1.5. Subqueries

Subqueries are nested queries within a main query, used to perform operations that require multiple steps.

1.5.1. Scalar Subquery

Query:

```
SELECT first_name, last_name,
       (SELECT COUNT(*) FROM film_actor fa WHERE fa.actor_id = a.actor_id) AS film_count
FROM actor a;
```

Explanation:

- **Purpose:** Retrieves each actor's name along with the count of films they have acted in.
- **Output:** Actor names with their respective film counts.

1.5.2. IN Subquery

List the names of customers who rented films between **'2005-01-01'** and **'2006-12-31'**.

```
SELECT
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name
FROM
    customer c
WHERE
    c.customer_id IN (
        SELECT DISTINCT r.customer_id
        FROM rental r
        WHERE r.rental_date BETWEEN '2005-01-01' AND '2006-12-31'
    );
```

1.5.3. EXISTS Subquery

Query:

```
SELECT first_name, last_name
FROM actor a
WHERE EXISTS (
    SELECT 1
    FROM film_actor fa
    WHERE fa.actor_id = a.actor_id AND fa.film_id = 1
);
```

Explanation:

- **Purpose:** Similar to the IN subquery, it retrieves actors who have acted in the film with `film_id = 1`.
- **Output:** List of actors associated with the specified film.

1.6. Aggregation Functions

Aggregation functions perform calculations on multiple rows of a table's column and return a single value.

Query:

```
SELECT c.first_name, c.last_name, COUNT(r.rental_id) AS rental_count
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
GROUP BY c.customer_id
HAVING rental_count > 5
ORDER BY rental_count DESC;
```

Explanation:

- **Purpose:** Retrieves customers who have made more than five rentals, along with their rental counts.
- **Output:** List of customers with rental counts exceeding five, ordered from highest to lowest.

1.7. GROUP BY and HAVING Clauses

Query:

```
SELECT ci.city, COUNT(c.customer_id) AS customer_count
FROM customer c
JOIN address a ON c.address_id = a.address_id
JOIN city ci ON a.city_id = ci.city_id
GROUP BY ci.city_id
having customer_count > 1
ORDER BY customer_count DESC;
```

Explanation:

- **Purpose:** Retrieves cities with more than 1 customers, along with the number of customers in each city.

- **Output:** List of cities meeting the customer count criteria, ordered by the number of customers.

1.8. UNION and UNION ALL

Query:

```
SELECT first_name, last_name FROM actor
WHERE last_name LIKE 'A%'
UNION
SELECT first_name, last_name FROM customer
WHERE last_name LIKE 'A%';
```

Explanation:

- **Purpose:** Combines the list of actors and customers whose last names start with ‘A’, removing duplicates.
- **Output:** Unified list of unique first and last names from both actors and customers.

Query with UNION ALL:

```
SELECT first_name, last_name FROM actor
WHERE last_name LIKE 'A%'
UNION ALL
SELECT first_name, last_name FROM customer
WHERE last_name LIKE 'A%';
```

Explanation:

- **Purpose:** Similar to the previous query but includes duplicate entries.
- **Output:** Combined list of first and last names from both actors and customers, including duplicates.

2. Practice Questions and Solutions for Sakila Database

Enhance your understanding of MySQL by practicing with the Sakila database. Below are 10 questions along with their solutions.

Question 1: List All Films Released in 2006

Query:

```
SELECT title, release_year
FROM film
WHERE release_year = 2006;
```

Solution:

- **Explanation:** This query selects the title and release year of films that were released in the year 2006.
- **Output:** A list of films with their titles and the year 2006.

Question 2: Find Customers from a Specific City

Query:

```
SELECT c.first_name, c.last_name, ci.city
FROM customer c
JOIN address a ON c.address_id = a.address_id
JOIN city ci ON a.city_id = ci.city_id
WHERE ci.city = 'London';
```

Solution:

- **Explanation:** Retrieves the first and last names of customers residing in London.
- **Output:** List of customers located in London.

Question 3: Retrieve Top 5 Actors with Most Film Appearances

Query:

```
SELECT a.first_name, a.last_name, COUNT(fa.film_id) AS film_count
FROM actor a
JOIN film_actor fa ON a.actor_id = fa.actor_id
GROUP BY a.actor_id
ORDER BY film_count DESC
LIMIT 5;
```

Solution:

- **Explanation:** Counts the number of films each actor has appeared in and selects the top five actors with the highest counts.
- **Output:** Top five actors with the most film appearances.

Question 4: List All Active Rentals

Query:

```
SELECT r.rental_id, c.first_name, c.last_name, f.title, r.rental_date
FROM rental r
JOIN customer c ON r.customer_id = c.customer_id
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
WHERE r.return_date IS NULL;
```

Solution:

- **Explanation:** Retrieves rentals that have not been returned yet (`return_date` is `NULL`), along with customer and film details.
- **Output:** List of active (unreturned) rentals.

Question 5: Calculate Total Revenue per Store

Query:

```
SELECT
    s.store_id,
    CONCAT(st.first_name, ' ', st.last_name) AS manager,
    SUM(p.amount) AS total_revenue
FROM
    store s
JOIN
    staff st ON s.manager_staff_id = st.staff_id
JOIN
    inventory i ON s.store_id = i.store_id
JOIN
    rental r ON i.inventory_id = r.inventory_id
JOIN
    payment p ON r.rental_id = p.rental_id
GROUP BY
    s.store_id, st.staff_id
ORDER BY
    total_revenue DESC
LIMIT 10;
```

Solution:

- **Explanation:** Calculates the total payment amount received per store, along with the store manager's name.
- **Output:** List of stores with their managers and total revenue, ordered by highest revenue.

Question 6: Find Films with No Rentals

Query:

```
SELECT f.title, f.release_year
FROM film f
LEFT JOIN inventory i ON f.film_id = i.film_id
LEFT JOIN rental r ON i.inventory_id = r.inventory_id
WHERE r.rental_id IS NULL;
```

Solution:

- **Explanation:** Identifies films that have never been rented by checking for `NULL` in the `rental_id` after left joins.
- **Output:** List of films with no rental records.

Question 7: List Customers Who Have Made the Least Payments bottom 10

Query:

```
SELECT
    c.first_name,
    c.last_name,
    c.email,
    COALESCE(SUM(p.amount), 0) AS total_payments
FROM
    customer c
LEFT JOIN payment p ON c.customer_id = p.customer_id
GROUP BY
    c.customer_id
ORDER BY
    total_payments ASC
LIMIT 10;
```

Solution:

- **Explanation**:
    - **`COALESCE(SUM(p.amount), 0)`** : Ensures that customers with no payments show a `0` total.
    - **`GROUP BY c.customer_id`** : Groups payments by each customer.
    - **`ORDER BY total_payments ASC`** : Sorts customers in ascending order of their total payments.
    - **`LIMIT 10`** : Returns the top 10 customers with the least payments.
- **Output**: A list of customers who have contributed the least revenue.

Question 8: Retrieve the Average Rental Duration per Category

Query:

```
SELECT cat.name AS category, AVG(f.rental_duration) AS avg_rental_duration
FROM category cat
JOIN film_category fc ON cat.category_id = fc.category_id
JOIN film f ON fc.film_id = f.film_id
GROUP BY cat.category_id
ORDER BY avg_rental_duration DESC;
```

Solution:

- **Explanation:** Calculates the average rental duration for each film category.
- **Output:** List of categories with their average rental durations.

Question 9: Find the Most Recent Rental for Each Customer

Query:

```
SELECT c.first_name, c.last_name, r.rental_date, f.title
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
WHERE r.rental_date = (
    SELECT MAX(r2.rental_date)
    FROM rental r2
    WHERE r2.customer_id = c.customer_id
)
ORDER BY r.rental_date DESC;
```

Solution:

- **Explanation:** Retrieves the most recent rental record for each customer by comparing rental dates.
- **Output:** Latest rental details for each customer.

Question 10: List Films Available for Rental in Each Store

Query:

```
SELECT s.store_id, CONCAT(st.first_name, ' ', st.last_name) AS manager, f.title, COUNT(i.inve
FROM store s
JOIN staff st ON s.manager_staff_id = st.staff_id
JOIN inventory i ON s.store_id = i.store_id
JOIN film f ON i.film_id = f.film_id
LEFT JOIN rental r ON i.inventory_id = r.inventory_id AND r.return_date IS NULL
WHERE r.rental_id IS NULL
GROUP BY s.store_id, f.film_id
ORDER BY s.store_id, f.title;
```

Solution:

- **Explanation:** Lists each film available for rental in each store by counting inventory items not currently rented out.
- **Output:** For each store, the films available along with the number of available copies.

3. Order of Execution in SQL Queries

Understanding the logical order in which SQL processes query clauses helps in writing optimized and accurate queries. Below is the typical order of execution:

General Query Structure

1. **FROM**: Specify the tables to retrieve data from.
2. **JOIN**: Combine rows from multiple tables based on related columns.
3. **WHERE**: Filter rows based on specific conditions.
4. **GROUP BY**: Group rows based on one or more columns.
5. **HAVING**: Filter grouped rows based on aggregated conditions.
6. **SELECT**: Specify the columns to retrieve, including aggregate functions.
7. **DISTINCT**: Remove duplicate rows (if needed).
8. **ORDER BY**: Sort rows based on one or more columns.
9. **LIMIT**: Restrict the number of rows returned.

Illustrative Example:

```
SELECT c.first_name, c.last_name, COUNT(r.rental_id) AS rental_count
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
WHERE c.active = 1
GROUP BY c.customer_id
HAVING rental_count > 5
ORDER BY rental_count DESC
LIMIT 10;
```

Execution Order:

1. **FROM customer c JOIN rental r**:
   - Specify the tables (`customer` and `rental`) to retrieve data from.
   - Combine rows from these tables using a `JOIN` based on `customer_id` .
2. **WHERE c.active = 1**:
   - Filter rows to include only active customers (`c.active = 1`).
3. **GROUP BY c.customer_id**:
   - Group the results by each customer.
   - This step ensures that aggregations like `COUNT(r.rental_id)` apply to each customer separately.
4. **HAVING rental_count > 5**:
   - Filter the grouped rows to include only those customers with more than five rentals.
5. **SELECT c.first_name, c.last_name, COUNT(r.rental_id) AS rental_count**:
   - Specify the columns to retrieve (`first_name` , `last_name` , and the count of rentals).

6. **ORDER BY rental_count DESC**:
   - Sort the results by the `rental_count` in descending order, showing the most frequent renters first.
7. **LIMIT 10**:
   - Restrict the output to the top 10 customers based on rental count.

4. Best Practices for Writing Efficient SQL Queries

- **Use Explicit JOINs:** Prefer `JOIN` clauses over implicit joins in the `WHERE` clause for better readability and maintenance.
- **Indexing:** Ensure that columns used in `JOIN`, `WHERE`, and `ORDER BY` clauses are properly indexed to improve query performance.
- **Avoid SELECT ***: Specify only the necessary columns to reduce the amount of data processed and transferred.
- **Use Aliases:** Employ table aliases to make queries more readable, especially when dealing with multiple tables.
- **Filter Early:** Apply `WHERE` clauses as early as possible to reduce the dataset size before performing joins and aggregations.
- **Limit Results:** Use `LIMIT` to restrict the number of rows returned when testing queries to enhance performance.
- **Understand Execution Plans:** Use `EXPLAIN` to analyze how MySQL executes a query and optimize accordingly.
- **Normalize Data:** Ensure the database schema is normalized to reduce redundancy and improve data integrity.
- **Handle NULLs Appropriately:** Be mindful of `NULL` values in data and handle them using functions like `COALESCE` or by setting default values.
- **Maintain Consistent Formatting:** Write queries with consistent indentation and formatting for better readability and maintenance.