Views in MySQL

Definition and Explanation

A **view** in MySQL is a virtual table that represents the result of a stored query. It does not store data physically but provides a way to simplify complex queries by encapsulating them into a single table-like structure. Views can be queried like regular tables, allowing for more straightforward data retrieval and improved security by restricting access to specific data.

Key Characteristics

- **Virtual Table**:
  Views do not hold data themselves; they display data stored in other tables.
- **Simplification**:
  They simplify complex SQL queries by encapsulating them into a single virtual table.
- **Security**:
  Views can restrict user access to specific rows or columns, enhancing data security.
- **Updatability**:
  Some views are updatable, meaning you can perform `INSERT`, `UPDATE`, or `DELETE` operations on them, which will affect the underlying base tables.

Use Cases

1. **Simplifying Complex Queries**:
   Encapsulate complex joins and calculations into a view for easier data retrieval.

2. **Data Security**:
   Provide users with access to specific data without exposing the entire table.

3. **Data Abstraction**:
   Present data in a different structure without altering the underlying tables.

Examples

Example 1: Creating a View for Actor Information

The Sakila database includes a view named `actor_info` that provides detailed information about actors and the films they have participated in.

```
CREATE VIEW actor_info AS
SELECT
    a.actor_id,
    a.first_name,
    a.last_name,
    GROUP_CONCAT(f.title ORDER BY f.title) AS filmography
FROM
    actor a
JOIN
    film_actor fa ON a.actor_id = fa.actor_id
JOIN
    film f ON fa.film_id = f.film_id
GROUP BY
    a.actor_id, a.first_name, a.last_name;
```

Example 2: Creating a View for Customer List

The `customer_list` view provides a formatted list of customers, including their names, contact information, and active status.

```
CREATE VIEW customer_list AS
SELECT
    cu.customer_id AS ID,
    CONCAT(cu.first_name, ' ', cu.last_name) AS name,
```

```
    a.address AS address,
    a.phone AS phone,
    cu.active AS active
FROM
    customer cu
JOIN
    address a ON cu.address_id = a.address_id;
```

Example 3: Creating a View for Sales by Film Category

The `sales_by_film_category` view summarizes sales revenue grouped by film category.

```
CREATE VIEW sales_by_film_category AS
SELECT
    c.name AS category,
    SUM(p.amount) AS total_sales
FROM
    payment p
JOIN
    rental r ON p.rental_id = r.rental_id
JOIN
    inventory i ON r.inventory_id = i.inventory_id
JOIN
    film f ON i.film_id = f.film_id
JOIN
    film_category fc ON f.film_id = fc.film_id
JOIN
    category c ON fc.category_id = c.category_id
GROUP BY
    c.name
ORDER BY
    total_sales DESC;
```

Benefits of Using Views

---

1. **Code Reusability**:
   Write the query once and reuse it multiple times.

2. **Simplified Querying**:
   Users can query a simple view instead of writing complex SQL joins.

3. **Security**:
   Restrict access to sensitive data by exposing only specific columns or rows.

4. **Maintainability**:
   If the underlying tables change, updating the view can abstract those changes from the end-user.

Updatable Views:

A view is considered **updatable** if it allows `INSERT`, `UPDATE`, or `DELETE` operations that directly affect the underlying base tables. However, not all views possess this capability.

Conditions for Updatable Views:

For a view to be updatable, it must meet certain criteria:

- **Single-Table Reference:** The view should reference only one base table.
- **No Aggregate Functions:** Functions like `SUM()`, `MIN()`, `MAX()`, `COUNT()`, etc., should be absent.
- **No `DISTINCT`, `GROUP BY`, `HAVING`, `UNION`, or `UNION ALL`:** These clauses render a view non-updatable.
- **No Subqueries in the `SELECT` or `WHERE` Clause:** Subqueries referring to the table in the `FROM` clause can prevent updatability.
- **No `JOIN`s:** Views involving joins are generally non-updatable, though there are exceptions with certain inner joins.
- **No `TEMPTABLE` Algorithm:** Views created with the `TEMPTABLE` algorithm are not updatable.

Checking View Updatability:

MySQL sets an **updatability flag** when a view is created. You can verify if a view is updatable by querying the `INFORMATION_SCHEMA.VIEWS` table:

```
SELECT TABLE_NAME, IS_UPDATABLE
FROM INFORMATION_SCHEMA.VIEWS
WHERE TABLE_SCHEMA = 'sakila';
```

Predefined views in the `Sakila` database.

1. `actor_info` View

This view provides detailed information about actors and their filmography.

Query Example

```
SELECT actor_id, first_name, last_name, film_info
FROM actor_info
WHERE film_info LIKE '%Action%';
```

- **Purpose**: Retrieve all actors who have acted in films belonging to the “Action” genre.

---

2. `customer_list` View

This view lists customers with their contact information and active status.

Query Example

```
SELECT ID, name, address, phone, notes
FROM customer_list
```

```
WHERE notes = 'active';
```

- **Purpose**: Get a list of all active customers, including their contact details.

---

3. `film_list` View

This view provides details about films, including their category, rental rate, and availability.

Query Example

```
SELECT FID, title, category, price
FROM film_list
WHERE category = 'Comedy' AND price < 3.00;
```

- **Purpose**: List all comedy films with a rental price under $3.00.

---

4. `nicer_but_slower_film_list` View

This view is similar to `film_list` but may execute more slowly due to additional processing.

Query Example

```
SELECT FID, title, description
FROM nicer_but_slower_film_list
WHERE title LIKE '%Love%';
```

- **Purpose**: Find all films with “Love” in the title.

---

5. `sales_by_film_category` View

This view summarizes total sales by film category.

Query Example

```
SELECT category, total_sales
FROM sales_by_film_category
ORDER BY total_sales DESC;
```

- **Purpose**: Display total sales for each film category, sorted by the highest sales.

---

6. `sales_by_store` View

This view shows total sales for each store, along with the store manager's details.

Query Example

```
SELECT store, manager, total_sales
FROM sales_by_store
ORDER BY total_sales DESC;
```

- **Purpose**: Compare sales performance between different stores.

---

7. `staff_list` View

This view lists staff members with their contact information and the store they work in.

Query Example

```
SELECT ID, name, SID AS store, address, phone
FROM staff_list
WHERE SID = 1;
```

- **Purpose**: Get a list of all staff members working at “Store #1”.

Combining Multiple Views

Example: Retrieve Customer Rentals and Staff Information

```
SELECT
    cl.name AS customer_name,
    fl.title AS film_title,
    sl.name AS staff_name,
    sl.SID AS store_id
FROM
    customer_list cl
JOIN
    rental r ON cl.ID = r.customer_id
JOIN
    inventory i ON r.inventory_id = i.inventory_id
JOIN
    film_list fl ON i.film_id = fl.FID
JOIN
    staff_list sl ON r.staff_id = sl.ID
LIMIT 10;
```

- **Purpose**: Display a list of customers, the films they rented, and the staff members who processed the rentals.

Stored Procedures in MySQL

Definition and Explanation

A **stored procedure** in MySQL is a set of SQL statements that are stored in the database and can be executed as a single unit. Stored procedures allow for modular programming by encapsulating business logic, making code reusable, and reducing redundancy.

Stored procedures can accept parameters, perform complex operations, and return results. They are especially useful for automating repetitive tasks and improving performance by reducing the amount of communication between applications and the database server.

Key Characteristics

- **Modularity**:
  Encapsulate a sequence of SQL statements into a single callable unit.
- **Performance Improvement**:
  Reduces network traffic by executing multiple SQL statements on the server with a single call.
- **Reusability**:
  Once created, stored procedures can be reused across different applications and users.
- **Security**:
  Permissions can be granted to execute a stored procedure without granting direct access to the underlying tables.
- **Parameter Support**:
  Stored procedures can accept `IN`, `OUT`, and `INOUT` parameters.

Syntax for Creating a Stored Procedure

```
DELIMITER //

CREATE PROCEDURE procedure_name (parameters)
BEGIN
    -- SQL statements
END //
```

```
DELIMITER ;
```

- **`DELIMITER`** : Changes the default delimiter `;` to `//` to allow defining multi-statement procedures.
- **`procedure_name`** : The name of the stored procedure.
- **`parameters`** : Input (`IN`), output (`OUT`), or input-output (`INOUT`) parameters.
- **`BEGIN...END`** : Contains the SQL statements that make up the procedure.

Parameter Types

1. **`IN` Parameter**: Passes values into the procedure.
2. **`OUT` Parameter**: Returns values from the procedure.
3. **`INOUT` Parameter**: Passes values into the procedure and returns updated values.

Use Cases

1. **Automating Business Logic**:
   Encapsulate complex business rules for consistency.

2. **Reducing Redundancy**:
   Reuse the same logic in multiple parts of an application.

3. **Data Validation**:
   Ensure data integrity by validating inputs before performing operations.

4. **Complex Transactions**:
   Execute multiple SQL statements within a single transaction.

Examples

Example 1: Simple Stored Procedure

This procedure retrieves a customer's email based on their ID.

```
DELIMITER //

CREATE PROCEDURE GetCustomerEmail(IN customerID INT)
BEGIN
    SELECT email
    FROM customer
    WHERE customer_id = customerID;
END //

DELIMITER ;
```

Usage:

```
CALL GetCustomerEmail(1);
```

Example 2: Stored Procedure with `OUT` Parameter

This procedure calculates the total payments made by a customer and returns the result.

```
DELIMITER //

CREATE PROCEDURE GetCustomerTotalPayments(IN customerID INT, OUT totalPayme
BEGIN
    SELECT SUM(amount) INTO totalPayments
    FROM payment
    WHERE customer_id = customerID;
END //

DELIMITER ;
```

Usage:

```
CALL GetCustomerTotalPayments(1, @totalPayments);
SELECT @totalPayments;
```

Example 3: Stored Procedure with `INOUT` Parameter

This procedure updates a customer's rental count and returns the new count.

```
DELIMITER //

CREATE PROCEDURE UpdateRentalCount(INOUT customerID INT)
BEGIN
    UPDATE customer
    SET rental_count = rental_count + 1
    WHERE customer_id = customerID;
END //

DELIMITER ;
```

Usage:

```
SET @customerID = 1;
CALL UpdateRentalCount(@customerID);
SELECT @customerID;
```

Advantages of Stored Procedures

- **Performance Optimization**: Reduces the number of database requests by executing multiple statements in a single call.
- **Reusability**: Centralize and reuse logic across applications.
- **Security**: Control access to data by exposing procedures instead of raw tables.
- **Maintainability**: Easier to maintain and update logic in one place.

Disadvantages of Stored Procedures

- **Debugging Complexity**: Debugging stored procedures can be more difficult compared to application code.
- **Portability Issues**: Stored procedures may not be easily portable across different database systems.

- **Performance Overhead**: Excessive use of complex procedures may lead to performance issues.