Lab 6
Views and Roles

CSE 4308

DATABASE MANAGEMENT SYSTEM LAB

NOVEMBER 4, 2024

Contents

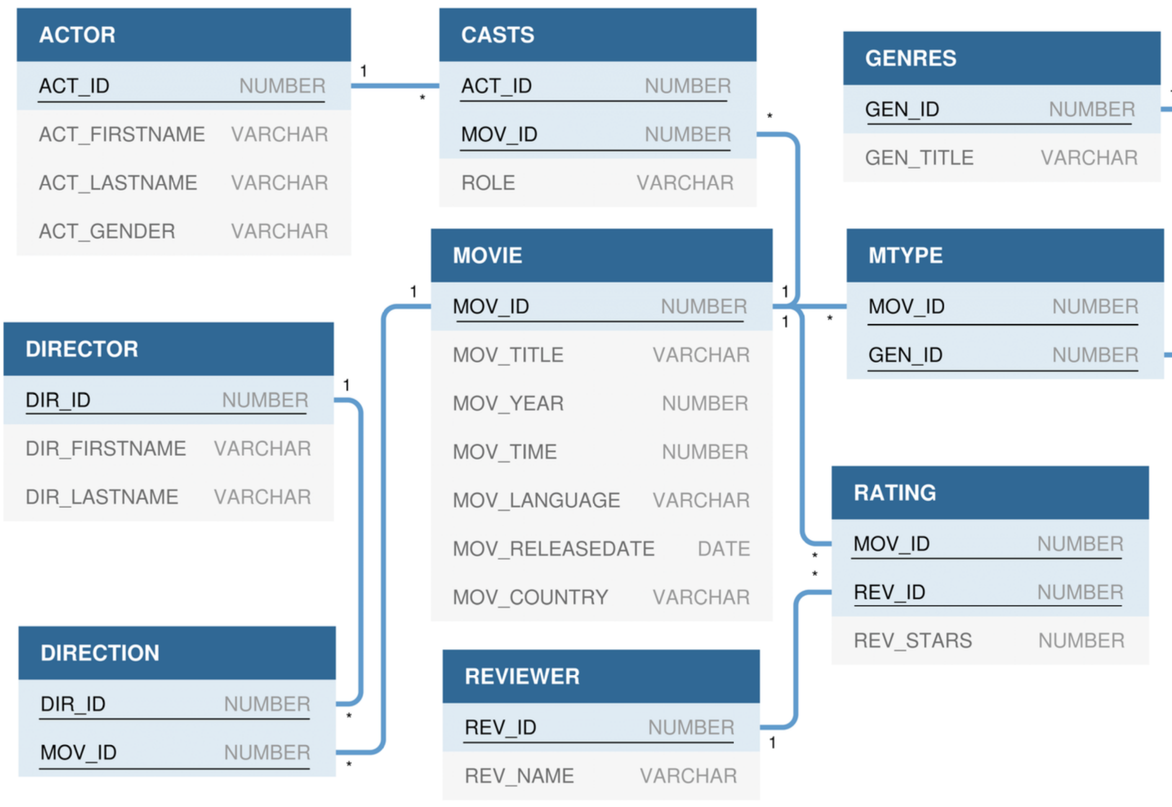
1 View 2
2 Roles 4
3 Lab Task 7

1 View

A MySQL VIEW, in essence, is a virtual table that does not physically store data. Rather, it is created by a query joining one or more tables. It derives its data from the tables on which it is based, allowing us to simplify complex queries and secure certain data from underlying tables. Views in MySQL can be treated just like any other table, but they do not occupy additional storage space. The syntax for creating a view in MySQL is:

```
CREATE [OR REPLACE] VIEW view_name AS
SELECT columns
FROM tables
[WHERE conditions];
```

Unlike a table, a view does not store any data. To be precise, a view only behaves like a table and acts as a stored query in the database. This is why views are often referred to as "named queries." Consider a sample movie database schema:



If we want to create a view showing only the ID, title, and language of the movies, we can create a view as follows:

```
CREATE OR REPLACE VIEW MOVIE_SUMMARY AS
SELECT MOV_ID, MOV_TITLE, MOV_LANGUAGE
FROM MOVIE;
```

Now we can query data from the view:

```
SELECT DISTINCT MOV_LANGUAGE
FROM MOVIE_SUMMARY;
```

Behind the scenes, MySQL finds the stored query associated with the name MOVIE_SUMMARY and executes it as defined.

```sql
SELECT DISTINCT MOV_LANGUAGE
FROM (
    SELECT MOV_ID, MOV_TITLE, MOV_LANGUAGE
    FROM MOVIE
) AS derived_table;
```

The result set returned from the view depends on the data of the underlying table. Any changes in the MOVIE table will be reflected in queries against the MOVIE_SUMMARY view.
There are two types of views in MySQL: Simple Views and Complex Views. Key differences between them are as follows:

Simple View	**Complex View**
Contains only one single base table or is created from only one table.	Contains more than one base table or is created from more than one table.
Does not contain aggregated functions, GROUP BY, DISTINCT, pseudo-columns like ROWNUM, or columns defined by expressions.	Can contain aggregated functions, GROUP BY, DISTINCT, pseudo-columns like ROWNUM, and columns defined by expressions.
Does not include NOT NULL columns from base tables.	NOT NULL columns that are not selected by the simple view can be included in a complex view.

In MySQL, simple views can be updated if they meet specific criteria (e.g., no JOINs, aggregations, or subqueries). For example, MOVIE_SUMMARY is a simple view, so we can modify it:

```sql
INSERT INTO MOVIE_SUMMARY (MOV_ID, MOV_TITLE, MOV_LANGUAGE)
VALUES ('935', 'Emily the Criminal', 'English');

UPDATE MOVIE_SUMMARY
SET MOV_ID = '934'
WHERE MOV_ID = '935';

DELETE FROM MOVIE_SUMMARY
WHERE MOV_TITLE = 'Emily the Criminal';
```

Views can be dropped using the following syntax:

```sql
DROP VIEW view_name;
```

Complex views cannot be updated or modified directly, whereas simple views in MySQL can sometimes be updated if they follow certain criteria (e.g., they do not include joins or aggregations).
A major use case for views, particularly complex views, is simplifying data retrieval. For example, we can determine the genres where male actors are more prevalent than female actors using the following query:

```sql
CREATE OR REPLACE VIEW SEXIST_GENRES AS
SELECT G.GEN_TITLE
FROM GENRES G
JOIN MTYPE M ON G.GEN_ID = M.GEN_ID
JOIN CASTS C ON M.MOV_ID = C.MOV_ID
JOIN ACTOR A ON C.ACT_ID = A.ACT_ID
GROUP BY G.GEN_TITLE
HAVING SUM(A.ACT_GENDER = 'M') > SUM(A.ACT_GENDER = 'F');
```

- This query finds all genres where male actors are cast more frequently than female actors. By grouping by genre and counting male and female actors separately within each genre, we're able to determine if male actors are in the majority for that genre.
- The final result set includes only the genres where male actors outnumber female actors, giving us a list of "male-preferred" genres according to the current dataset.
- Adding the line `CREATE OR REPLACE VIEW SEXIST_GENRES AS` before the query will create a view, which can simplify other queries that utilize this result.

Views can also provide an additional security layer by hiding certain columns and rows from underlying tables and exposing only necessary data to specific users. For instance, in a `STUDENT` table, instructors might only need access to student IDs and departments.

2 Roles

Role-Based Access Control (RBAC) Example

RBAC enables us to manage user permissions efficiently by assigning roles instead of direct permissions. Each role can be granted specific permissions, and users inherit those permissions when assigned to the role. Below, we'll define roles based on access to specific tables in the movie database.

Syntax for Creating and Managing Roles

```
CREATE ROLE role_name;
DROP ROLE role_name;
```

To grant privileges to a role on a table:

```
GRANT privilege_name ON table_name TO role_name [WITH GRANT
   OPTION];
```

We can also assign one role to another, effectively passing all privileges:

```
GRANT role_name_1 TO role_name_2;
```

Assigning a role to a user:

```
GRANT role_name TO user_name;
```

Example Roles for the Movie Database

We'll create three roles for different types of users in the movie database:

- **Viewer** - Can view movies and their ratings.
- **Reviewer** - Can view and rate movies.
- **Admin** - Has full access to add, update, or delete movie data.

Step 1: Define Roles and Grant Privileges

1. Viewer Role

This role is for users who only need read access to movies and ratings.

```
CREATE ROLE Viewer;
GRANT SELECT ON MOVIE TO Viewer;
GRANT SELECT ON RATING TO Viewer;
```

2. Reviewer Role

This role includes Viewer permissions and allows users to insert new ratings.

```
CREATE ROLE Reviewer;
GRANT Viewer TO Reviewer;
GRANT INSERT ON RATING TO Reviewer;
```

3. Admin Role

This role has full access to the movie database, including the ability to modify movie records and assign privileges.

```
CREATE ROLE Admin;
GRANT SELECT, INSERT, UPDATE, DELETE ON MOVIE TO Admin;
GRANT SELECT, INSERT, UPDATE, DELETE ON ACTOR TO Admin;
GRANT SELECT, INSERT, UPDATE, DELETE ON RATING TO Admin;
GRANT ALL ON MOVIE TO Admin WITH GRANT OPTION;
```

Step 2: Create Users and Assign Roles

```
DROP USER IF EXISTS ViewerUser;
DROP USER IF EXISTS ReviewerUser;
DROP USER IF EXISTS AdminUser;

CREATE USER ViewerUser IDENTIFIED BY 'password123';
CREATE USER ReviewerUser IDENTIFIED BY 'password123';
CREATE USER AdminUser IDENTIFIED BY 'password123';

GRANT Viewer TO ViewerUser;
GRANT Reviewer TO ReviewerUser;
GRANT Admin TO AdminUser;
```

Users inherit the permissions granted to their roles. We can assign multiple roles to a user if needed:

```
GRANT Viewer, Reviewer TO AdminUser;
```

Step 3: Testing Role-Based Access

To test access, connect as a specific user and attempt operations within their role scope:

```
-- Test SELECT permission on MOVIE
SELECT * FROM MOVIE;

-- Test SELECT permission on RATING
SELECT * FROM RATING;

-- Attempt to insert a record in RATING (should fail)
INSERT INTO RATING (movie_id, rating) VALUES (1, 5);

-- Attempt to update a record in MOVIE (should fail)
```

```
UPDATE MOVIE SET title = 'New Title' WHERE movie_id = 1;

-- Attempt to delete a record in MOVIE (should fail)
DELETE FROM MOVIE WHERE movie_id = 1;
```

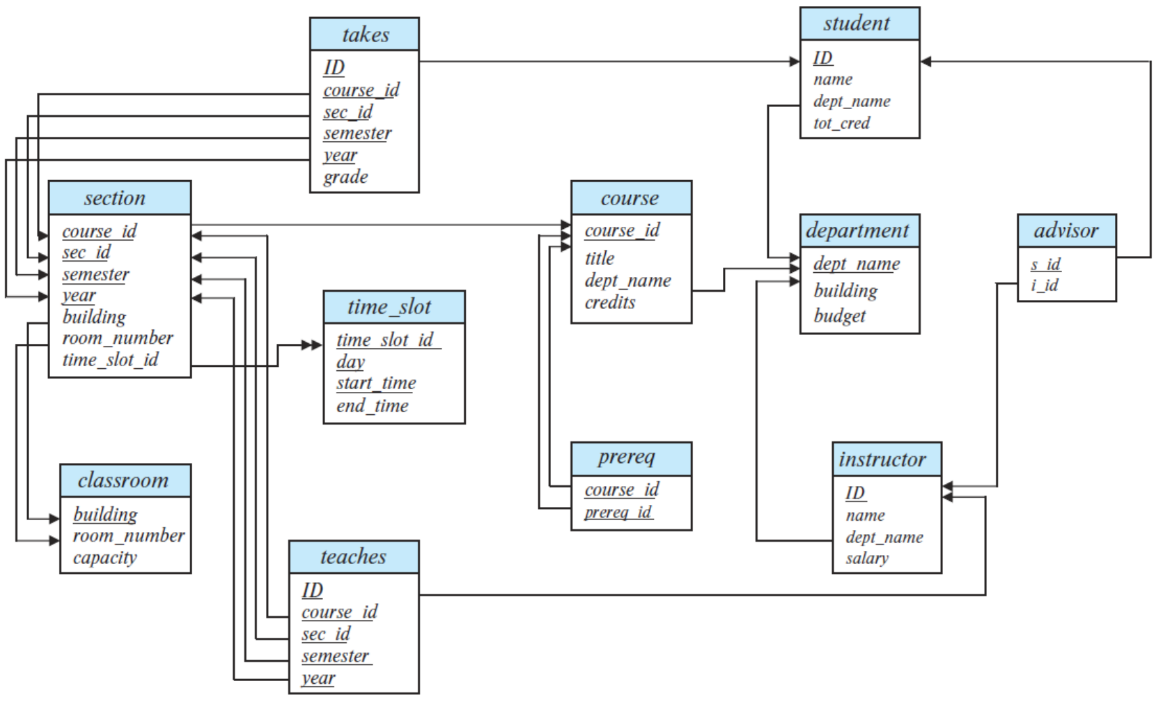
To revoke all roles;

```
-- Revoke roles from users before dropping them
REVOKE Viewer FROM ViewerUser;
REVOKE Reviewer FROM ReviewerUser;
REVOKE Admin FROM AdminUser;

-- Drop roles in reverse order to remove dependencies
DROP ROLE IF EXISTS Admin;
DROP ROLE IF EXISTS Reviewer;
DROP ROLE IF EXISTS Viewer;
```

3 Lab Task

Execute the `DDL+drop.sql` and `smallRelationsInsertFile.sql` scripts using command. It creates a set of tables along with values that maintain the following schema: Here, the underlined attributes denote the primary keys and the arcs denote the foreign



key relationships. In this lab, you have to write all SQL statements in an editor first and save them with .sql extension. Then execute the SQL script.

1. Create a view named `Advisor_Selection` that shows the ID, name, and department name of instructors.
2. Create another view named `Student_Count` using `Advisor_Selection` and `advisor` to show the name of the instructor and the number of students assigned to them.
3. Four categories of users have been identified in the database:
   - Students should be able to view information regarding advisors and courses.
   - Course teachers should be able to view information about students and courses.
   - Heads of Departments should have all the privileges of course teachers, and additionally be able to add new instructors.
   - Administrators should be able to view department and instructor information and update the department budget.
4. Create roles for these categories, grant them the appropriate privileges, create users under these roles, and write SQL queries to demonstrate access control.

Submission Guidelines

You are required to submit a report that includes your code, a detailed explanation, and the corresponding output. Rename your report as `<StudentID_Lab_1.pdf>`.

- Your lab report must be generated in LaTeX. Recommended tool: Overleaf. You can use some of the available templates in Overleaf.
- Include all code/pseudo code relevant to the lab tasks in the lab report.
- Please provide citations for any claims that are not self-explanatory or commonly understood.
- It is recommended to use vector graphics (e.g., `.pdf`, `.svg`) for all visualizations.
- In cases of high similarities between two reports, both reports will be discarded.